Available online at www.sciencedirect.com
SCIENCE DIRECT®
Pervasive and Mobile Computing 1 (2005) 425–445
pervasive and mobile computing
www.elsevier.com/locate/pmc

Sizzle: A standards-based end-to-end security architecture for the embedded Internet☆

Vipul Gupta*, Michael Wurm, Yu Zhu, Matthew Millard, Stephen Fung, Nils Gura, Hans Eberle, Sheueling Chang Shantz

Sun Microsystems Laboratories, 16 Network Circle, UMPK16 160, Menlo Park, CA 94025, USA

Received 4 June 2005; accepted 22 August 2005
Available online 10 October 2005

Abstract

According to popular perception, public-key cryptography is beyond the capabilities of highly constrained, "mote"-like, embedded devices. We show that elliptic curve cryptography not only makes public-key cryptography feasible on these devices, it allows one to create a complete secure web server stack that runs efficiently within very tight resource constraints. Our small-footprint HTTPS stack, nicknamed Sizzle, has been implemented on multiple generations of the Berkeley/Crossbow motes where it runs in less than 4 KB of RAM, completes a full SSL handshake in 1 s (session reuse takes 0.5 s) and transfers 1 KB of application data over SSL in 0.4 s. Sizzle is the world's smallest secure web server and can be embedded inside home appliances, personal medical devices, etc., allowing them to be monitored and controlled remotely via a web browser without sacrificing end-to-end security.

Keywords: Sensor network security; Elliptic Curve Cryptography; Smallest secure webserver

☆ Expanded version of a paper that received the Mark Weiser Best Paper Award at PerCom 2005.

* Corresponding author. Tel.: +1 650 786 7551; fax: +1 650 786 6013.
E-mail addresses: vipul.gupta@sun.com (V. Gupta), mwurm@sime.com (M. Wurm), davidyuzhu@hotmail.com (Y. Zhu), mmillard@kos.net (M. Millard), stephen.fung@gmail.com (S. Fung), nils.gura@sun.com (N. Gura), hans.eberle@sun.com (H. Eberle), sheueling.chang@sun.com (S. Chang Shantz).

1574-1192/$ - see front matter
doi:10.1016/j.pmcj.2005.08.005

1. Introduction

In the last several years, the Internet has grown rapidly beyond servers, desktops and laptops to include handheld devices like PDAs and smart phones. There is now a growing realization that this trend will continue as increasing numbers of even simpler, more constrained devices (sensors, home appliances, personal medical devices) get connected to the Internet. The term "embedded Internet" is often used to refer to the part of the Internet that is invisibly and tightly woven into our daily lives. Embedded devices with sensing and communication capabilities will enable the application of computing technologies in settings where they are unusual today: habitat monitoring [25], medical emergency response [31], battlefield management and home automation.

Many of these applications have security requirements. For example, health information must only be made available to authorized personnel (authentication) and be protected from modification (data integrity) or disclosure (confidentiality) in transit. Even seemingly innocuous data such as temperature and pressure readings may need to be secured. Consider the case of a chemical plant where sensors are used to continuously monitor the reactions used in manufacturing the final product. Without adequate security, an attacker could feed highly abnormal readings into the monitoring system and trigger catastrophic reactions.

Secure Sockets Layer (SSL)[1] [10] is the most commonly used security protocol on the Internet today. It is built into many popular applications, including all well-known web browsers, and is widely trusted to secure sensitive transactions including on-line banking, stock trading and e-commerce. This paper describes our investigation into using the same protocol to secure the embedded Internet.

SSL combines public-key cryptography for key-distribution and authentication with symmetric-key cryptography for data encryption and integrity. Public-key cryptography is widely believed to be beyond the capabilities of embedded devices. This perception is primarily driven by earlier experiments involving C-language implementations of RSA, today's dominant public-key cryptosystem [30].

Recent work in our research group has shown that optimized, assembly-language implementations of RSA perform much better and the use of Elliptic Curve Cryptography (ECC) provides an additional order of magnitude performance improvement on 8-bit CPUs [15]. First proposed by Victor Miller [20] in 1985, and independently by Neal Koblitz [18], ECC is emerging as an attractive alternative to RSA for resource-constrained environments.

We have built a small-footprint secure web server stack (including HTTP and SSL), called Sizzle[2] on top of our ECC and RSA implementations. Sizzle runs efficiently under tight resource constraints and interoperates with browsers like Mozilla, Internet Explorer and Safari. It can take advantage of the higher performance of ECC when communicating with an ECC-enabled version of Mozilla [11].

The main contributions of this paper are:

[1] Throughout this paper, we use SSL to refer to all versions of this protocol including version 3.1 *aka* Transport Layer Security (TLSv1.0) [7].

[2] This name derives from "Slim SSL" (SSSL).

- We describe the first fully implemented, end-to-end security architecture for highly constrained embedded devices.
- We describe the challenges posed by tight resource constraints on these devices and design choices we made to overcome them.
- We evaluate the performance and resource utilization of various subcomponents as well as the complete system and show that they are reasonable for their intended application scenarios.
- We compare several generations of mote devices to deduce the impact of technology trends like faster CPUs vs. faster networking on different aspects of our architecture.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 provides an overview of Elliptic Curve Cryptography. Section 4 discusses the SSL protocol and its use of ECC. Section 5 describes Sizzle, including its main features and the overall architecture. In Section 6, we present performance results and resource consumption statistics for Sizzle. Finally, Section 7 summarizes our conclusions.

2. Related work

Secure web servers for small devices have been built by PeerSec Networks [22] and Zingg [32]. The Mini Web Server with SSL [32] targets the IPC@CHIP platform which has a 20 MHz, 16-bit Intel 80186 processor, 512 KB of Flash, 512 KB of RAM and a built-in Ethernet connection. The SSL code size is around 100 KB. MatrixSSL [22] has a smaller footprint (around 50–70 KB), but still targets 32-bit and 64-bit CPUs, e.g., ARM7, MIPS, PowerPC, i386 and x86-64. Both only support RSA-based key exchange, and on the IPC@CHIP platform, RSA decryption takes nearly 45 s [32].

Neither implementation described above is suitable for highly constrained platforms like the Berkeley/Crossbow "motes" [6]: wireless, battery-powered devices with much weaker CPUs and a mere 4–10 KB of RAM (see Fig. 1 for details). The motes are particularly interesting because they are emerging as the preferred platform for much of sensor-related research in academia and industry [27]. Prior security proposals for such devices have deemed public-key cryptography to be "too expensive" and "impractical" and relied on symmetric-key algorithms with manual pre-distribution of keys [8,17,23,24]. Unfortunately, these schemes do not scale, offer only link-level (rather than end-to-end) security and risk the security of the entire network on an attacker's ability to compromise a few devices. Sizzle addresses these shortcomings by utilizing highly optimized, assembly-language implementations of public-key cryptography. Sizzle was first introduced in [13], and this paper extends that work by evaluating its performance on a wider array of devices, comparing the use of ECC against RSA and studying the impact of different technology trends (e.g., faster CPU vs. faster communication) and optimizations (e.g., persistent-HTTPS).

3. Elliptic Curve Cryptography

At the foundation of every public-key cryptosystem is a hard mathematical problem that is computationally intractable. The relative difficulty of solving that problem determines the security strength of the corresponding system. Since the best known algorithms to

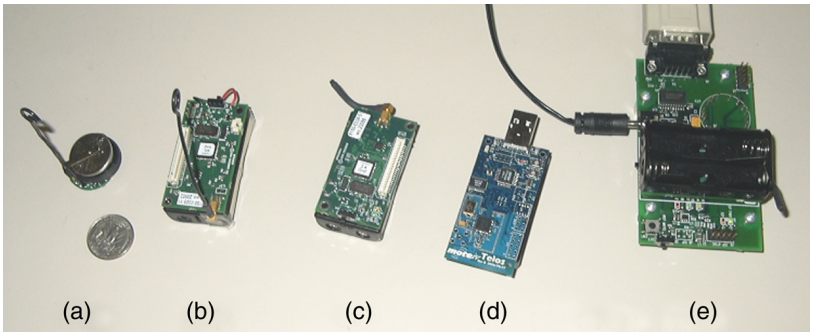

Fig. 1. The Berkeley/Crossbow family of "mote" devices: (a) Mica2dot, (b) Mica2, (c) MicaZ, (d) Telos (Rev B), and (e) development board. Traditional public-key cryptography is perceived to be beyond the capabilities of these devices. The first three motes have an 8-bit Atmel ATmega processor, 128 KB instruction memory (FLASH) and 4 KB RAM. The CPU is clocked at 4 MHz on the Mica2dot and at 7.37 MHz on the Mica2 and MicaZ. The Telos has an 8 MHz, 16-bit TI MSP430 processor, 48 KB FLASH and 10 KB RAM. The Mica2dot and Mica2 use a proprietary radio, Chipcon CC1000, rated at 20 kbps, while the MicaZ and Telos use an IEEE 802.15.4 radio rated at 250 kbps. Each Mica mote can be plugged into the development board to create a base station for communicating with other motes using the same radio. A Telos mote can act as a base station when plugged into the USB port of a PC or laptop.

attack ECC have fully exponential run times but the best known algorithms to attack RSA have sub-exponential run times [29], ECC can offer equivalent security with substantially smaller key sizes [19,21], e.g., a 160-bit ECC key provides the same level of security as a 1024-bit RSA key, and 224-bit ECC is equivalent to 2048-bit RSA. Smaller keys result in faster computations, and lower power consumption as well as memory and bandwidth savings, making ECC especially appealing for resource-constrained environments. More importantly, the performance advantage of ECC over RSA increases as security needs increase over time. According to Gura et al. [15], 160-bit ECC key-exchange operations are 13 times faster than 1024-bit RSA decryption operations on the mote, but 224-bit ECC operations are almost 38 times faster than 2048-bit RSA decryption operations.

ECC operates on a group of points on an elliptic curve defined over a finite field. Its main cryptographic operation is *scalar point multiplication*, which computes $Q = kP$ (a point P on an elliptic curve multiplied by an integer k resulting in another point Q on the curve). Scalar multiplication is performed through a combination of *point additions* and *point doublings*. For example, $11P$ can be expressed as $11P = (2((2(2P)) + P)) + P$. The security of ECC relies on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP), which states that, given P and $Q = kP$, it is hard to find k. Besides the curve equation, an important elliptic curve parameter is the *base point*, G, which is fixed for each curve. In ECC, a large random integer k acts as a private key, while the result of multiplying the private key k with the curve's base point G serves as the corresponding public key.

The Elliptic Curve Diffie Hellman (ECDH) key exchange [1] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [2] are elliptic curve counterparts of the well-known Diffie-Hellman and DSA algorithms, respectively. ECC was recently approved by the National Institute of Standards and Technology (NIST) for US Government use [28],

and several standards organizations, including the American National Standards Institute (ANSI), the Institute of Electrical & Electronic Engineers (IEEE), the Open Mobile Alliance (OMA) and the Internet Engineering Task Force (IETF), have ongoing efforts to include it as a required or recommended security mechanism.

4. Overview of the SSL protocol

SSL offers encryption, source authentication and integrity protection for data and is flexible enough to accommodate different cryptographic algorithms for key agreement, encryption and hashing. Particular combinations of these algorithms are called *cipher suites*; for example, the cipher suite *TLS_RSA_WITH_RC4_128_SHA* uses RSA for key exchange, 128-bit RC4 for bulk encryption, and SHA for hashing.

The two main components of SSL are the Handshake protocol and the Record Layer protocol. The Handshake protocol allows an SSL client and server to negotiate a common cipher suite, authenticate each other, and establish a shared *master secret* using public-key algorithms. The Record Layer derives symmetric keys from the master secret and uses them with faster symmetric-key algorithms for bulk encryption and authentication of application data.

Since public-key operations are computationally expensive, the protocol's designers added the ability for a client and server to reuse a previously established master secret. This feature is also known as "session resumption", "session reuse" or "session caching". The resulting abbreviated handshake does not involve any public-key cryptography, requires fewer, shorter messages, and can be completed more quickly.

4.1. ECC-based full handshake

Fig. 2 shows the general operation of an ECDH–ECDSA handshake, as specified in [12]. The client and server first exchange random values (for replay protection) and negotiate a cipher suite using the *Client Hello* and *ServerHello* messages. The *ServerCertificate* message contains the server's ECDH public key signed by a certificate authority using ECDSA. After validating the ECDSA signature, the client conveys its ECDH public key to the server in the *ClientKeyExchange* message. Next, each entity uses its own ECDH private key and the other's public key to perform an ECDH operation and arrive at a shared premaster secret. Both end points then use the premaster secret to create a master secret which, along with previously exchanged random values, is used to derive the cipher keys, initialization vectors and MAC (Message Authentication Code) keys for bulk encryption and authentication by the Record Layer.

The use of ECDH and ECDSA in SSL mimics the use of DH and DSA, respectively. The SSL specification already defines cipher suites based on DH and DSA, so the incorporation of ECC is not a large change. Our research team has added support for ECC cipher suites in both OpenSSL and Mozilla [14].

4.2. RSA-based full handshake

An RSA-based handshake is similar to the ECC-based handshake, but there are a few differences. The *Certificate* message contains the server's RSA public key signed using

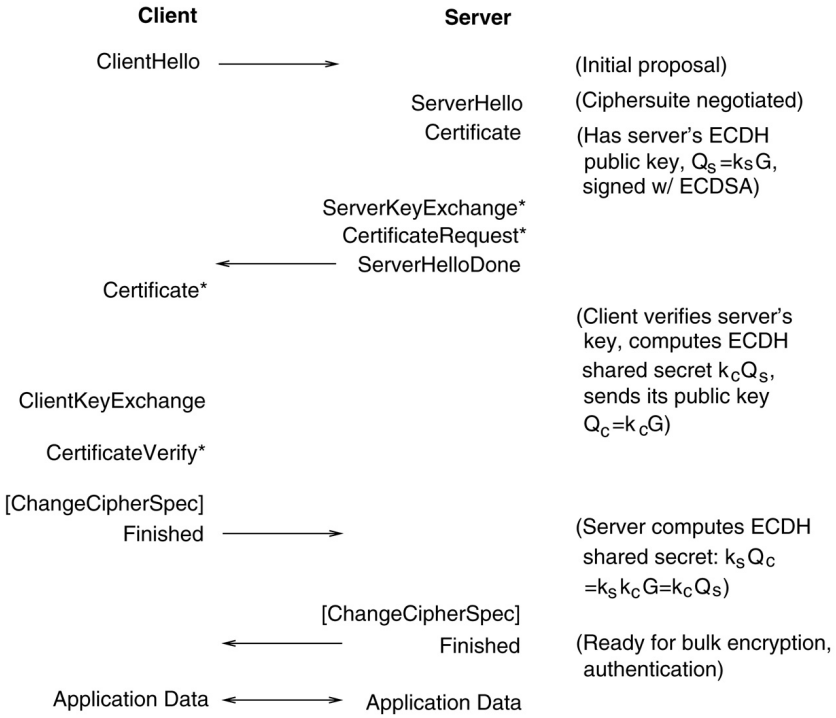

Fig. 2. An ECDH-based SSL handshake. The most computationally intensive part on the server is ECC point multiplication. Messages marked with an asterisk are optional and only sent in certain protocol variants.

RSA. The *ClientKeyExchange* message contains a random value encrypted with that public key. The server uses its corresponding private key to decrypt the random value which serves as a premaster secret. RSA decryption is a much slower compared to an equivalent ECDH operation.

4.3. Abbreviated handshake

The abbreviated handshake is shown in Fig. 3. Here, the *ClientHello* message includes the non-zero ID of a previously negotiated session. If the server still has that session information cached and is willing to reuse the corresponding master secret, it echoes the session ID in the *ServerHello* message.[3] Otherwise, it returns a new session ID, thereby signalling the client to engage in a full handshake. The derivation of symmetric keys from the master secret is identical to the full handshake scenario.

4.4. SSL and HTTP

The most common usage of SSL is in securing HTTP, the main transport protocol of the World Wide Web. HTTP is a simple request-and-response protocol and its use

[3] The likelihood of a cache hit depends on the server's configuration and its current workload.

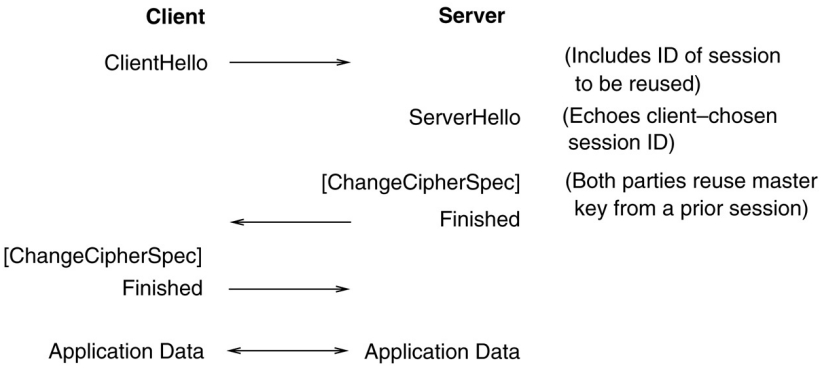

Fig. 3. An abbreviated SSL handshake does not involve any public-key operation and uses fewer messages. It is, therefore, much faster than a full handshake.

over SSL is referred to as HTTPS. In the original version of HTTP, each request and its corresponding response was sent over a new TCP connection. A subsequent improvement, called persistent HTTP [9], keeps the TCP connection open for a configurable duration so other requests arriving within that period can be serviced over the same connection. Opening and closing fewer TCP connections saves CPU time and memory, reduces network congestion, and improves response time.

When SSL is used to protect HTTP, each new TCP connection requires a fresh handshake: either full or abbreviated, depending on the session cache at each end-point. The use of persistent HTTP with SSL reduces the cost of security significantly by amortizing the computational and communication cost of a handshake across multiple request–response exchanges (see Fig. 4).

5. Sizzle

While SSL has been identified as a good solution for securing Internet communications, it has been considered too “heavy-weight” for highly constrained embedded devices like the motes due to its reliance on public-key cryptography [17]. We have demonstrated that efficient implementations of public-key cryptography are feasible on these devices, and Sizzle is our small footprint implementation of an HTTPS stack that brings the well-established security properties of SSL to the embedded Internet.

Sizzle allows one to embed a secure web server in tiny devices (e.g., utility meters) for the purpose of remote monitoring and control. Fig. 5 shows the architecture we have implemented that is applicable to these scenarios. The introduction of a gateway between the monitoring/control station and the devices being monitored/controlled provides several benefits:

- The gateway serves as a bridge between the embedded devices and the rest of the Internet. It connects to the Internet using a high-speed link (e.g., Ethernet) and communicates with one or more embedded devices using a lower-speed wireless link (such as IEEE 802.15.4) optimized for power consumption.

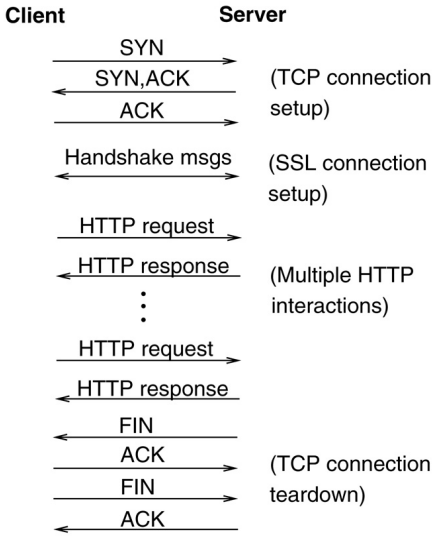

Fig. 4. Message flow in persistent HTTPS. Notice how the cost of an SSL handshake can be amortized across multiple request–response exchanges.

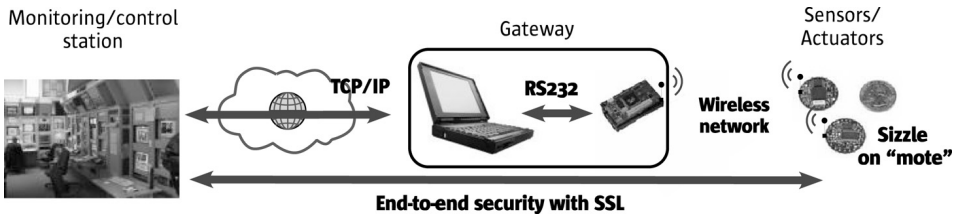

Fig. 5. Gateway-based architecture for making embedded devices accessible across the Internet. The gateway acts as a bridge between a fast TCP/IP network and a collection of sensor devices connected via a slower, energy efficient, wireless network. Even though the TCP connection terminates at the gateway, the security provided by SSL extends end-to-end.

- The gateway provides a single choke point for controlling access to the embedded devices. It can implement various mechanisms including address-based filtering to enforce selective access from across the Internet. The gateway is also the ideal place to log all interactions with the embedded devices.
- The gateway can serve as a performance-enhancing proxy. In particular, the TCP protocol over which much of the Internet traffic (including both HTTP and HTTPS) flows interprets packet loss as an indication of congestion. This causes TCP to perform poorly when the connection involves a wireless hop [4]. A well-known approach for alleviating this performance degradation splits the end-to-end path at the wireless link boundary [3]; precisely where the gateway is situated. Terminating TCP at the gateway and using a special purpose reliable protocol between the gateway and the embedded devices has several benefits:

- the special-purpose protocol can be made simpler because it need not support end-to-end congestion control across multiple, possibly heterogeneous, links;
- it can be tailored for the special loss characteristics of the single link (e.g., a NACK-based scheme may be used if most of the packets sent are delivered successfully); and
- local packet loss recovery improves overall performance.

There is an important difference between this security architecture and other gateway-based architectures for connecting small devices, most notably WAP 1.0 [26], where the gateway sees all traffic in the clear—decrypting incoming data and re-encrypting it before passing it along. In those architectures, the gateway needs to be trusted and compromising it compromises all connections passing through it. With our approach, the security provided by SSL extends end-to-end. All data stays encrypted as it crosses the gateway, i.e., even if the latter is compromised, an attacker is unable to view or alter any data.

SSL is a versatile protocol supporting many cryptographic algorithms and several variations in authenticating the entities involved. We chose a subset of these features to meet tight resource constraints while addressing the security needs for a wide array of usage scenarios. Here we list some of the bandwidth, memory and computation saving features of Sizzle.

(1) Sizzle implements a relatively small set of cryptographic algorithms. The MD5 and SHA1 hashing algorithms are mandatory for SSL since they are used in the derivation of the master secret and symmetric keys. We selected RC4 for bulk encryption because of its speed, compact code size and widespread support in existing SSL deployments. We chose ECDH–ECDSA (using the 160-bit curve secp160r1) for key exchange due to its resource efficiency and speed, but also implemented RSA (using 1024-bit keys) for performance comparisons and compatibility with a wide range of web browsers. Compile time flags can be used to build versions that support only ECC or only RSA.
(2) The cipher suites enabled in Sizzle do not entail sending the *ServerKeyExchange* message and the server's ECC or RSA public key is sent in a certificate. We deliberately chose the subject and issuer names in certificates to be brief and omitted optional extensions. The resulting ECC certificate has 222 bytes and the corresponding RSA certificate has 413 bytes.
(3) Sizzle uses 4-byte session identifier values rather than the 32-byte values used in Apache and other servers with much larger scalability requirements. Tight memory constraints preclude storing information for more than a few connections in the session cache and a 4-byte identifier is adequate to identify each cached session uniquely. Besides implementing session reuse, Sizzle also implements persistent HTTP(S) to further reduce the overhead of public-key cryptography.
(4) Sizzle does not send out the *CertificateRequest* message which automatically eliminates the client's *Certificate*/*CertificateVerify* messages and obviates the need for any certificate parsing code in Sizzle. Clients can still be authenticated using passwords (one-time or otherwise) over SSL as is commonly done for on-line banking, stock trading and e-commerce.[4]

[4] We have implemented this feature so certain operations, like device reconfiguration, require authentication.

Fig. 6. ECC-enabled wireless devices built at Sun Labs: (a) thermostat, (b) health-monitoring watch (c) magnetic stripe reader.

(5) Static information such as the server's private key, corresponding certificate and static portions of application-specific web pages, are stored in program memory, rather than data memory, to reduce RAM usage. Both the Atmel ATmega and MSP430 allow a programmer to specify certain memory areas as sensitive and deny general read access to them.
(6) Sizzle maintains state for only one SSL connection at a time. If a new connection attempt is made while a prior connection is still open (for persistent HTTPS) but idle, that old connection is terminated gracefully before the new connection is established.
(7) Another bandwidth-saving feature we experimented with is the use of reduced HTTP headers. Unlike other features on this list, this one is implemented on the web browser rather than the Sizzle server. The Mozilla browser, for example, sends over 400 bytes of headers in each HTTP request. This represents a large communication overhead in many situations where the HTTP response sent by Sizzle is small. We added a new checkbox in Mozilla's HTTP Networking configuration panel to omit sending certain headers, e.g., Accept, Accept-Language, Accept-Encoding, etc., that are ignored by Sizzle. Enabling this feature reduces the size of a typical HTTP request to around 100 bytes.

We have implemented the architecture shown in Fig. 5 using mote devices for sensors/actuators and a web browser as the monitoring/controlling application. Multiple devices can be connected via a single gateway and the secure web server within each device is mapped to a distinct TCP port at the gateway. Fig. 6 shows some of the wireless devices built in our laboratories. We embedded a mote with the Tiny OS operating system [27] running Sizzle inside a wireless thermostat whose settings can be read and modified using a browser. Fig. 7 shows a screenshot of an ECC-enabled Mozilla browser communicating with the thermostat. The closed lock icon in the browser's lower right corner indicates that the communication is protected by SSL and the connection details reveal an ECC public key in the server's certificate.

The SSL protocol was designed to be layered atop a reliable, bi-directional byte-stream service, typically provided by TCP. Unfortunately, Tiny OS does not include a reliable data transfer mechanism and a significant portion of our effort went into implementing such a

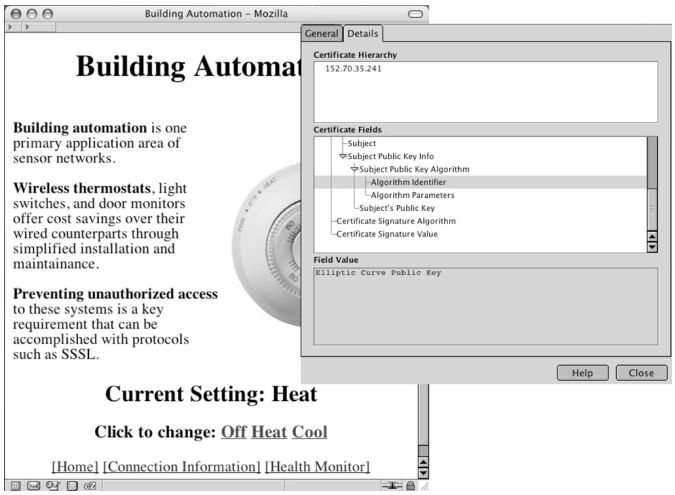

Fig. 7. A screenshot of an ECC-enabled Mozilla browser communicating with Sizzle embedded inside a wireless thermostat. The browser can be used to monitor and change the thermostat settings securely across a TCP/IP network. Images linked to the HTML page are served from another location, i.e., to conserve memory, these images are not stored inside Sizzle.

mechanism. Our first prototype used a simple stop-and-go ACK-based protocol. However, since Tiny OS uses fixed-size packets, sending an ACK for each data packet halved the effective throughput. Our current prototype transmits the first and last packets in a data block with explicit ACKs. Other packets in between use a NACK-based scheme where only lost packets are explicitly signaled. In addition, three other control packets are used: (i) NEWCONNECTION indicates that the gateway has accepted a new TCP connection, (ii) DISCONNECT tells the gateway to terminate a TCP connection, and (iii) READY indicates a mote's readiness to receive the next message from the gateway. All three are sent with explicit ACKs.

Application messages are fragmented into Tiny OS packets and sent across the wireless link without the overhead of TCP or IP headers. Each Tiny OS packet has a payload of 28 bytes, but our reliable transmission scheme uses 6 of those bytes for housekeeping chores like sequence numbers and return addresses.[5] Therefore, an exemplary 286-byte record containing the *ServerHello*, *Certificate* and *ServerHelloDone* messages is fragmented into $\lceil 286/22 \rceil = 13$ Tiny OS packets. However, when this record is transmitted on the wired link, it is sent along with a 20-byte TCP header (TCP options increase the header size) and a 20-byte IP header.

6. Experimental results

All measurements reported in this section were made with Tiny OS 1.1.10 which supports multiple kinds of motes. Versions prior to 1.1.7 did not support the IEEE 802.15.4 radio found on the MicaZ and Telos.

[5] There is considerable room for optimization here.

Table 1
Memory (in bytes) consumed by a wireless thermostat application embedding Sizzle

	FLASH	RAM (static)	RAM (max. stack size)	
			w/ ECC	w/ RSA
Mica2dot	48,882	3094	656	950
Mica2	49,116	3094	656	950
MicaZ	48,516	3040	629	925
TelosB	40,988	2780	651	853

Total RAM usage with RSA runs very close to the 4096-byte limit on the Mica family of devices, making ECC a superior alternative.

6.1. Memory usage

We used the objdump utility to determine static memory usage of Sizzle on various mote devices. Run time usage was determined by filling the memory region allocated to program stack with specific byte values and checking how much of it was overwritten during program execution.

Table 1 presents our measurements for a wireless thermostat application embedding Sizzle. These numbers include resources consumed by TinyOS, our reliable communication layer, cryptographic algorithms, the SSL and HTTP protocol handlers and multiple application-specific web pages.[6] The application-specific portion of the code is roughly 8 KB independent of the device. There is plenty of room in FLASH memory to implement even more complex applications. However, on devices other than Telos, the total RAM usage with RSA is close to the 4 KB limit. So from this perspective alone (ignoring performance for now), ECC is a superior alternative.

The MSP430 processor's ability to operate on 16-bit operands compared to the 8-bit Atmel ATmega processor results in more compact cryptographic code but the maximum available code memory on the MSP platform is less than a third of that on the Atmel platforms. The percentage of total FLASH memory devoted to cryptography is around 15% (17–20 KB) on the Atmel platforms and about 35% (12–14 KB) for the MSP (see Table 2).

6.2. Handshake, data transfer speed

We measured the speed of an SSL handshake separately from that of bulk data transfer because the latter is application specific and dependent upon the amount of data being sent. There are four built-in counters on the ATmega and two on the MSP430. We used these to count the number of CPU cycles spent in individual cryptographic algorithms. Transmission time of individual messages was measured at the gateway by noting the delay between sending (receiving) of the first packet and receiving (sending) of the last acknowledgment. Delays for all messages were summed up to arrive at the overall time spent in data transfer. The total time spent in a handshake or bulk transfer was

[6] There were web pages for learning about the thermostat, monitoring/controlling its settings and authenticating users.

Table 2
Bytes of FLASH memory devoted to cryptographic code in Sizzle for the Atmel ATmega and MSP430 processors

	Atmel ATmega (128 KB max)	MSP430 (40 KB max)
RSA	6444 + 1349	3180 + 1350
ECC	5348 + 268	3328 + 268
SHA1	2046	1486
MD5	2442	1742
RC4	228	216
SSL	7556	5848

Numbers after the plus sign indicate additional memory used to store certificates, keys and other related constants.

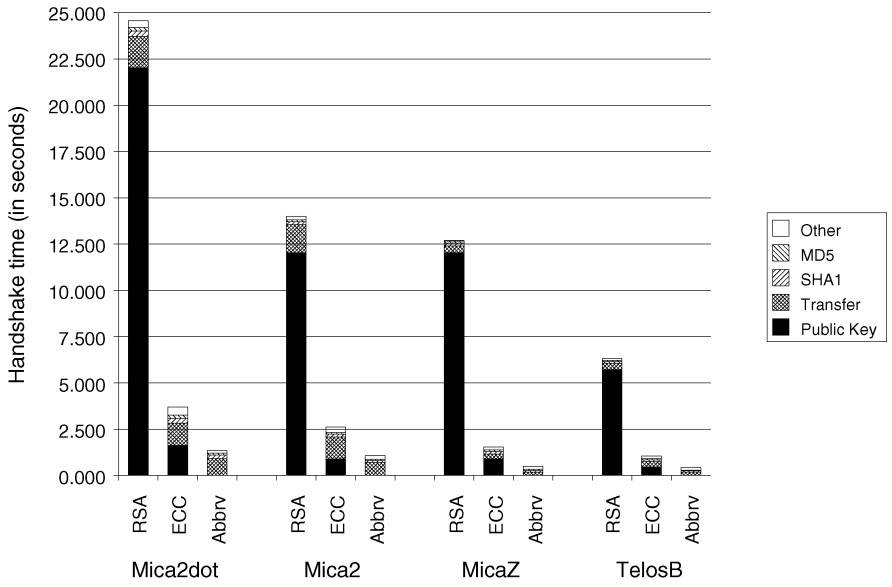

Fig. 8. Time taken for three kinds of SSL handshake on different motes: (i) RSA-based full handshake, (ii) ECC-based full handshake, and (iii) abbreviated handshake. Note that the RSA handshake benefits significantly from an improvement in CPU capabilities—doubling the CPU speed (from Mica2dot to Mica2), or data path (from MicaZ to Telos), nearly halves the total time. For the abbreviated handshake, speeding up data transmission (from the Mica2/Mica2dot to the MicaZ/Telos) has a bigger impact.

also measured at the gateway by noting the delay between sending the first packet in a *ClientHello* or HTTP request and receiving the last packet in a *Finished* message or HTTP response.

Our handshake measurements are presented in Fig. 8. As expected, an RSA-based handshake is the slowest and an abbreviated handshake the fastest. An RSA handshake takes 6–25 s depending on the platform (of which the RSA decryption contributes 5.7–22 s); an ECC handshake takes 1–4 s (with ECC point multiplication contributing 0.5–1.6 s); and

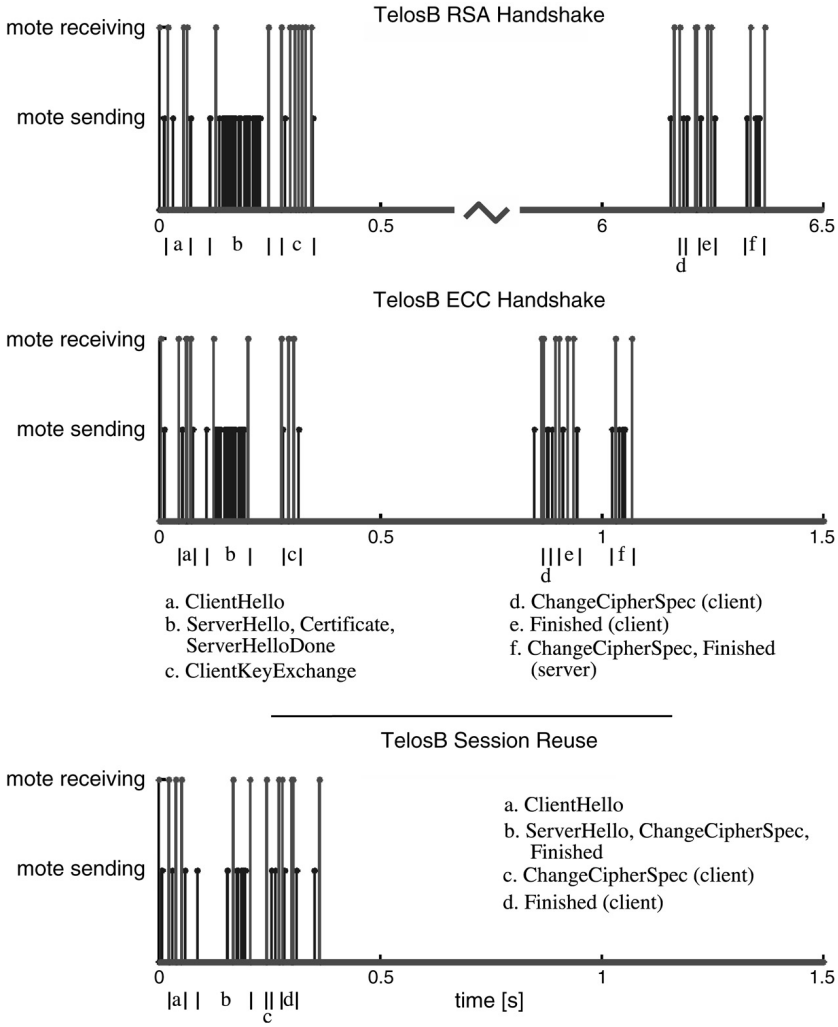

Fig. 9. Tiny OS packet exchange for different types of SSL handshakes. Public-key operations, visible as large gaps immediately following the receipt of the *ClientKeyExchange*, dominate the cost of a full handshake. An abbreviated handshake is faster because it eliminates public-key operations. The gap at the start of region "b" in the bottom plot is due to a rare retransmission event. The first two pulses in each plot depict NEWCONNECTION/ACK and the first and last packets in each message block are explicitly ACKed.

an abbreviated handshake takes 0.5–1.5 s. The plots also illustrate the impact of speeding up the CPU and the wireless network on handshake time. For example, an RSA handshake benefits the most from CPU improvements whereas an abbreviated handshake benefits the most from networking improvements. The ECC handshake falls in the middle and both kinds of improvements result in a noticeable decrease in handshake time.

Fig. 9 presents a more detailed view comparing the three types of handshakes. These plots are for the Telos platform but similar plots for other platforms have the same general

characteristics. The plots show the timing of Tiny OS packets and were obtained by flashing different pins on the gateway's base station in response to packet reception and transmission and measuring voltage changes across a resistor network connected to those pins.[7] Each short pulse corresponds to a Tiny OS packet sent from Sizzle to the gateway and each tall pulse represents a packet in the other direction. These figures illustrate our reliable communication scheme described earlier, including the use of control packets and the fragmentation of handshake messages across multiple TinyOS packets. Gaps without pulses represent periods when Sizzle is busy computing or waiting for data. The largest gap in Fig. 9(a) and (b) corresponds to the computation of the premaster secret via an RSA decryption or ECDH operation and the derivation of symmetric keys following the receipt of a *ClientKeyExchange* message. Not only is the RSA operation slower than the corresponding ECC operation, the larger RSA certificate also takes longer to transmit. The abbreviated handshake is faster because it does not involve public-key cryptography and uses fewer, shorter messages.

We collected bulk data transfer measurements using persistent HTTP with and without SSL for three web page sizes: 100, 500 and 900 bytes. Our browser was configured to send reduced HTTP headers so the total amount of application data corresponding to the three page sizes was around 200, 600 and 1000 bytes, respectively.

Fig. 10 reveals that when using RC4 and SHA1, the bulk transfer rate is limited by communication rather than computation speed. The pie charts also indicate that RC4 is significantly faster than SHA1 and there is a fixed overhead, due to initial setup, for both algorithms which causes them to take up a higher fraction of the overall transfer time for smaller amounts of application data.

Fig. 11 shows the time taken to transfer different amounts of application data with and without SSL. The plots are linear with respect to data size and their slope indicates the effective transmission rate. The additional processing time required by SSL results in a 10% drop in the effective data rate on the Mica2 and Mica2dot (from 6 to 5.4 kbits/s) and about a 20% drop (from 34 to 28 kbits/s) on the MicaZ and Telos. Typical applications on these devices transfer only small amounts of data at a time and, in these situations, the additional delay due to SSL is imperceptible (between 30 and 300 ms).

Fig. 12 presents a more detailed view comparing HTTP and HTTPS bulk transfer. These plots were obtained in the same way as those in Fig. 9 and reconfirm that time spent in cryptographic processing is small relative to that spent in data transfer.

Our reliable transmission scheme has a fairly high overhead especially when small data blocks are involved. For example, 21 signaling packets are sent for 26 data packets in a full ECC handshake and 15 signaling packets are sent for 13 data packets in an abbreviated handshake. Several of these can be piggybacked with data or other signaling packets rather than being sent separately. In spite of these shortcomings, the overall performance of Sizzle is quite acceptable for the kinds of potential applications envisioned for it: those involving infrequent communication amongst a mostly static set of entities.

[7] For the Mica2 and Mica2dot, similar plots can be obtained from direct measurements of the Receive Signal Strength Indicator (RSSI) pin on the gateway's radio transceiver.

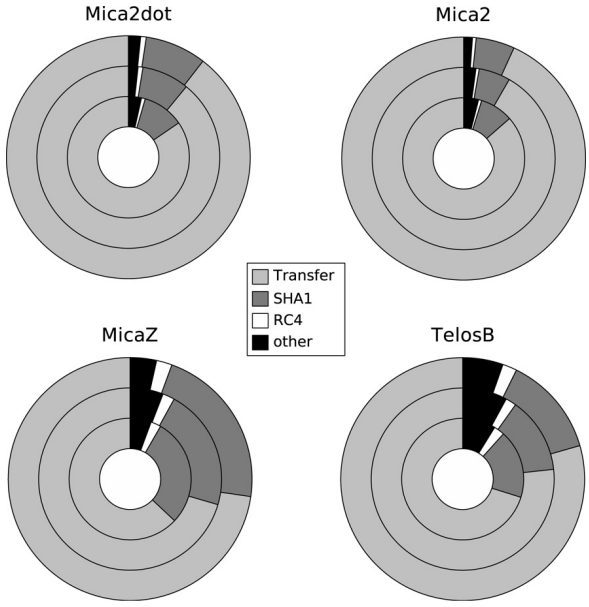

Fig. 10. Pie charts showing how time is spent in sending different amounts of application data over an SSL protected channel using RC4 for encryption and SHA1 for data integrity. From inside out, the three circles are for transferring 200, 600 and 1000 bytes, respectively. Note that wireless data transfer takes up 85%–95% of the overall time on Mica2dot and Mica2 and 60%–80% on Telos and MicaZ indicating that, in spite of faster 802.15.4 networking, secure data transfer is still limited by communication rather than computation speed.

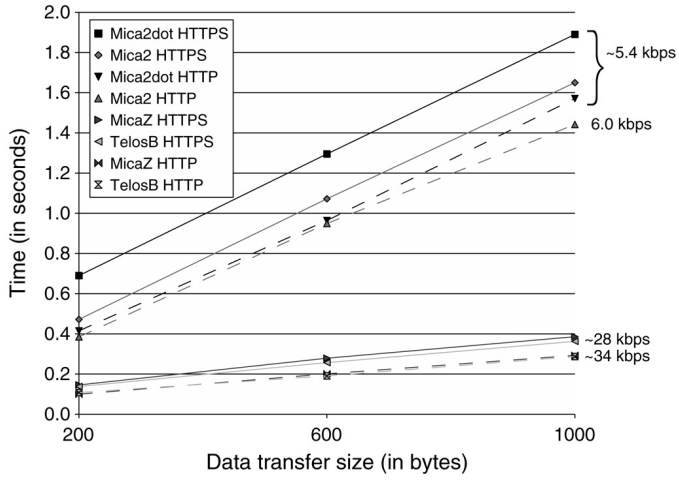

Fig. 11. Time spent in communicating and processing HTTP(S) requests and responses as a function of their size. Effective data transfer rate indicated by different slopes is shown on the right. Notice how HTTPS (solid lines) is only 100–300 ms slower than HTTP (dashed lines) on the Mica2dot and Mica2 and the difference is 30–100 ms on the MicaZ and Telos. This difference is practically imperceptible to most users.

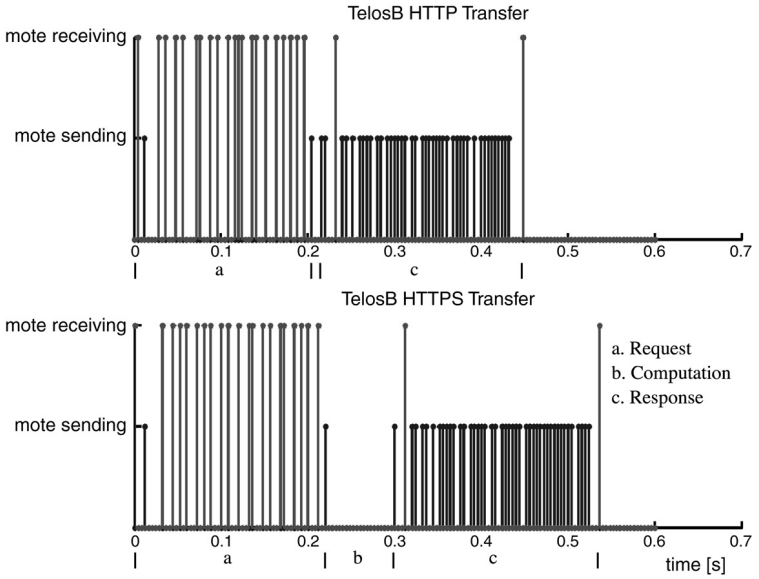

Fig. 12. Tiny OS packet exchanges comparing application data transfer over HTTP and HTTPS. Notice that interval "b" in the bottom plot is much smaller than the combined width of intervals "a" and "c". This indicates that symmetric encryption/decryption and computation/verification of the message authentication code is quite fast compared to wireless transmission.

6.3. Potential improvements

Data transmission consumes significantly more energy than computation on the motes. For example, transmitting 1 bit on the Mica2 consumes as much energy as running the CPU for over 2000 cycles. To decrease the latency and energy consumption associated with an SSL handshake, we propose protocol enhancements that reduce the amount of data transmitted across the wireless hop.

Figs. 13 and 14 show byte-level contents of the messages exchanged between the OpenSSL s_client program and Sizzle. Large portions of these messages stay unchanged between different connection requests to the same server mote and need not be transmitted explicitly. Gateway and devices can agree upon such static information a priori and permanently store or temporarily cache it. Static information can either be manually preconfigured or exchanged in an automated discovery phase whereby the gateway solicits embedded devices in its vicinity to register themselves. Only information that changes (shown shaded in the figure) needs to be sent along with an identifier selecting a message "template" (shown unshaded) for completion of the message.

The *Certificate* message is an extreme example where the entire content is fixed. Once device certificates have been conveyed to the gateway, the *Certificate* message only needs to carry a small certificate identifier (this could be the first 4 bytes of the certificate's MD5 hash) across the wireless hop. The gateway would use this identifier to reinsert the appropriate device certificate in the *Certificate* message before forwarding it on the TCP/IP connection. Similarly, the y coordinate of the elliptic curve point (public key) sent in the

```
SSL_connect:before/connect initialization
write to 0x609880 [0x186000] (55 bytes => 55 (0x37))
0000 - 80 35 01 03 01 00 0c 00-00 00 20 00 00 48 00 00
0010 - 04 01 00 80 00 00 05 d4-ef be 94 db 4f 4a a0 aa
0020 - cd d2 30 1b 0b 85 41 9f-d1 a0 ac 6f 9e 9a 41 a3
0030 - c1 aa a4 fd e0 c7 01
SSL_connect:SSLv2/v3 write client hello A
read from 0x609880 [0x18c000] (7 bytes => 7 (0x7))
0000 - 16 03 00 01 19 02
0007 - <SPACES/NULS>
read from 0x609880 [0x18c007] (279 bytes => 279 (0x117))
0000 - 00 2a 03 00 e8 72 c3 f2-b3 16 60 de 8b c9 59 02
0010 - b9 10 32 62 cd b9 41 f7-73 76 f5 d3 db b7 a3 d5
0020 - a3 87 79 7f 04 c4 81 9a-03 00 48 00 0b 00 00 e3
0030 - 00 00 e0 00 00 dd 30 81-da 30 81 9a 02 01 06 30
0040 - 09 06 07 2a 86 48 ce 3d-04 01 30 11 31 0f 30 0d
0050 - 06 03 55 04 03 13 06 53-55 4e 57 2d 45 30 1e 17
0060 - 0d 30 34 30 38 30 36 32-31 33 36 30 33 5a 17 0d
0070 - 30 38 30 39 31 34 32 31-33 36 30 33 5a 30 17 31
0080 - 15 30 13 06 03 55 04 03-13 0c 31 31 32 32 33 33
0090 - 34 34 35 35 36 36 30 3e-30 10 06 07 2a 86 48 ce
00a0 - 3d 02 01 06 05 2b 81 04-00 08 03 2a 00 04 ee 11
00b0 - 9d 01 01 4f 26 5a 62 87-f9 e3 a4 fc cc 88 84 4e
00c0 - bc 8b 46 dc be fa 7d b4-8d 0a ac 1f 01 89 8d f3
00d0 - ab 92 d4 b4 e7 f0 30 09-06 07 2a 86 48 ce 3d 04
00e0 - 01 03 30 00 30 2d 02 15-00 ad 70 97 86 11 fb da
00f0 - 60 a2 a5 af ec bc 79 8f-35 7c ad ce ed 02 14 72
0100 - 57 31 af 99 aa 69 1c 63-27 f9 90 8d 2e 23 5f bf
0110 - c8 79 92 0e
0117 - <SPACES/NULS>
SSL_connect:SSLv3 read server hello A
SSL_connect:SSLv3 read server certificate A
SSL_connect:SSLv3 read server done A
write to 0x609880 [0x2809800] (51 bytes => 51 (0x33))
0000 - 16 03 00 00 2e 10 00 00-2a 29 04 6d a0 e0 8d 9b
0010 - 60 8b c6 95 ab 1b 09 50-4a fa 82 81 d6 67 9c b2
0020 - 04 42 66 44 d2 19 bd 50-41 37 77 13 26 50 cc 66
0030 - 1e f0 98
SSL_connect:SSLv3 write client key exchange A
write to 0x609880 [0x2809800] (6 bytes => 6 (0x6))
0000 - 14 03 00 00 01 01
SSL_connect:SSLv3 write change cipher spec A
write to 0x609880 [0x2809800] (65 bytes => 65 (0x41))
0000 - 16 03 00 00 3c 7d dd 48-d3 81 9b ff 74 99 2a 82
0010 - 1f 56 30 7d 34 78 26 8e-b0 76 fd fb 4c aa f3 05
0020 - 65 50 4b bb c5 f0 54 12-8f 0c a5 11 40 7e 65 22
0030 - 37 f0 41 80 9c 2c 81 b8-1d a6 73 d8 0b ab 06 3e
0040 - 4f
SSL_connect:SSLv3 write finished A
SSL_connect:SSLv3 flush data
read from 0x609880 [0x18c000] (5 bytes => 5 (0x5))
0000 - 14 03 00 00 01
read from 0x609880 [0x18c005] (1 bytes => 1 (0x1))
0000 - 01
read from 0x609880 [0x18c000] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 3c
read from 0x609880 [0x18c005] (60 bytes => 60 (0x3C))
0000 - af ca 8a e7 ca 26 f5 44-c1 25 76 96 55 64 56 da
0010 - 2c 58 a6 e1 23 62 00 a2-b6 e6 b7 95 b3 44 a1 e5
0020 - 30 97 9e 0c 7a 39 4d 0c-5f bb 76 ec db f6 bd 02
0030 - 94 ad e6 94 97 67 2e 3d-83 ec 0f df
SSL_connect:SSLv3 read finished A
---
SSL handshake has read 357 bytes and written 177 bytes
---
New, TLSv1/SSLv3, Cipher is ECDH-ECDSA-RC4-SHA
SSL-Session:
    Protocol  : SSLv3
    Cipher    : ECDH-ECDSA-RC4-SHA
    Session-ID: C4819A03
    Master-Key: 922F0EE1622F4B61B16AA309FD1ECDDF
                1D18AB038BB81473DC115256EE366E87
          CED1240602EF76F77645633C05CF8D9E
---
```

Fig. 13. Byte-level contents of messages in a full handshake. Only small portions of the messages, shown shaded, change between different connection requests to the same server. Smarter encoding of these messages can reduce the amount of data transmitted by nearly 50%.

```
SSL_connect:before/connect initialization
write to 0x60a200 [0x191000] (58 bytes => 58 (0x3A))
0000 - 16 03 00 00 35 01 00 00-31 03 00 41 23 c0 88 37
0010 - 88 f6 33 b3 b8 70 d1 95-e0 93 cf 78 4e 6f 26 be
0020 - 7d fe 10 48 a7 b3 3e 21-c9 4c 6c 04 c4 81 9a 03
0030 - 00 06 00 48 00 04 00 05-01
003a - <SPACES/NULS>
SSL_connect:SSLv3 write client hello A
read from 0x60a200 [0x18c000] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 2e
read from 0x60a200 [0x18c005] (46 bytes => 46 (0x2E))
0000 - 02 00 00 2a 03 00 50 0f-ad b5 90 10 e6 3d 1b 66
0010 - 4f 8b da 2d bf 33 cb 6b-e2 1c 8e b3 ec a9 d9 d5
0020 - bf 14 4c 08 e9 57 04 c4-81 9a 03 00 48
002e - <SPACES/NULS>
SSL_connect:SSLv3 read server hello A
read from 0x60a200 [0x18c000] (5 bytes => 5 (0x5))
0000 - 14 03 00 00 01
read from 0x60a200 [0x18c005] (1 bytes => 1 (0x1))
0000 - 01
read from 0x60a200 [0x18c000] (5 bytes => 5 (0x5))
0000 - 16 03 00 00 3c
read from 0x60a200 [0x18c005] (60 bytes => 60 (0x3C))
0000 - 6b 8e 56 bb 6f b5 f2 cf-08 36 4a 1e ef 99 ee e1
0010 - ac 56 ec 33 83 96 50 75-6d d6 d7 b1 60 a0 59 93
0020 - 2b 9c 19 12 42 05 2e 7e-f2 92 9e 5f 5b bd 09 bc
0030 - 59 78 4f 12 51 c0 b7 e0-fc a4 3c 15
SSL_connect:SSLv3 read finished A
write to 0x60a200 [0x2809200] (6 bytes => 6 (0x6))
0000 - 14 03 00 00 01 01
SSL_connect:SSLv3 write change cipher spec A
write to 0x60a200 [0x2809200] (65 bytes => 65 (0x41))
0000 - 16 03 00 00 3c 73 d0 1b-0b da 77 0c ae 43 67 d3
0010 - 6f 2c 19 a7 6b 26 28 07-9a 77 0c 72 a4 14 7b 84
0020 - 2c 79 57 81 eb 80 00 a1-a5 2d a0 ad 8b 4e b5 7c
0030 - 36 33 75 17 4f ae e1 ee-2b da 4e 76 d3 17 58 c1
0040 - 7c
SSL_connect:SSLv3 write finished A
SSL_connect:SSLv3 flush data
---
SSL handshake has read 122 bytes and written 129 bytes
---
Reused, TLSv1/SSLv3, Cipher is ECDH-ECDSA-RC4-SHA
SSL-Session:
    Protocol  : SSLv3
    Cipher    : ECDH-ECDSA-RC4-SHA
    Session-ID: C4819A03
    Master-Key: 922F0EE1622F4B61B16AA309FD1ECDDF
                1D18AB038BB81473DC115256EE366E87
                CED1240602EF76F77645633C05CF8D9E32
---
```

Fig. 14. Byte-level contents of messages in an abbreviated handshake. Only small portions of the messages, shown shaded, change between different connection requests to the same server. Smarter encoding of these messages can reduce the amount of data transmitted by nearly 20%.

ClientKeyExchange can be deduced knowing the x coordinate and the curve equation. For a given x coordinate, there can be two possible values for y so an additional bit is needed to uniquely identify the y coordinate.[8] This data suppression/recreation functionality can be implemented in special modules inserted between the SSL record handler and the wireless

[8] An elliptic curve equation has the form $y^2 = x^3 + ax + b$.

Table 3
Contents of handshake messages after passing through the suppressor module

(a) Full handshake

Uncompressed message(s)	Compressed representation
ClientHello (55[a])	ClientHelloTemplateID (1), cipher suite len(2), cipher suite list (12[a]), client random (32)
ServerHello (51), *Certificate* (231), *ServerHelloDone* (4)	SvrHelloCertDoneTemplateID (1), server random (32), session ID (4), certificate ID (4)
ClientKeyExchange (51)	ClntKeyExchTemplateID (1), x coordinate (20), disambiguator for y coordinate (1)
ChangeCipherSpec (6) *Finished* (65)	CCSFinishedTemplateID (1), encrypted payload of Finished message (60)

(b) Abbreviated handshake

Uncompressed message(s)	Compressed representation
ClientHello (58[a])	ClientHelloTemplateID (1), cipher suite len(2), cipher suite list (6[a]), session ID (4), client random (32)
ServerHello (51)	SvrHelloTemplateID (1), server random (32), session ID (4)
ChangeCipherSpec (6) *Finished* (65)	CCSFinishedTemplateID (1), encrypted payload of Finished message (60)

Size in bytes shown within parentheses.
[a] Actual value depends on client's configuration.

transceiver at either end of the wireless hop. As far as the layers above these modules are concerned, the protocol is still SSL. This approach can reduce the amount of data transmitted by over 50% (from 534 bytes to 232 bytes) for a full SSL handshake and almost 20% (from 251 bytes to 204 bytes) for an abbreviated handshake (see Table 3).

Note that this scheme differs from SSL compression which aims to reduce the amount of application rather than handshake data. SSL compression works above the SSL record layer and uses a generic data compression algorithm (e.g., LZS or DEFLATE). The suppressor and recreator modules described above sit underneath the record layer and by using specific knowledge of the handshake messages achieve a much higher compression ratio. Our approach is similar to header compression schemes [5,16] where TCP/IP and other header portions that do not change are not sent explicitly and small changes are conveyed by only sending the differential; here, we extend this idea to a new scenario (SSL connection set up) and not just to message headers but their contents as well.

7. Conclusions

Sizzle, for the first time, brings the Internet's dominant security protocol (SSL) to devices with significant computational, memory and energy constraints. It uses highly

optimized implementations of public-key cryptography to offer scalable key management and end-to-end security without sacrificing efficiency. To the best of our knowledge, Sizzle running on the Berkeley/Crossbow Mica2dot mote represents the world's smallest secure web server in terms of both physical dimensions and resource utilization. It is now possible to embed a secure web server in a wide array of tiny devices including home appliances, light fixtures, utility meters, temperature and pressure sensors, sprinkler systems, and personal medical devices—and monitor/control them securely across the Internet.

Acknowledgments

The authors wish to thank Arun Patel and Arvinderpal Wander for their help in several aspects of this project.

References

[1] ANSI X9. 62, Elliptic Curve Key Agreement and Key Transport Protocols, American Bankers Association, 1999.
[2] ANSI X9. 63, The Elliptic Curve Digital Signature Algorithm, American Bankers Association, 1999.
[3] A. Bakre, B.R. Badrinath, I-TCP: Indirect TCP for mobile hosts, in: 15th International Conference on Distributed Computing Systems, 1995.
[4] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, R.H. Katz, A comparison of mechanisms for improving TCP performance over wireless links, IEEE/ACM Transactions on Networking 5 (6) (1997) 756–769.
[5] C. Bormann et al., RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed, IETF RFC 3095, July 2001.
[6] Crossbow Technology, Inc., Crossbow product information. http://www.xbow.com/Products/products.htm.
[7] T. Dierks, C. Allen, The TLS Protocol—Version 1.0, IETF RFC 2246, January 1999.
[8] L. Eschenauer, V. Gligor, A key management scheme for distributed sensor networks, in: 9th ACM Conference on Computer and Communication Security, ACM Press, 2002, pp. 41–47.
[9] R.T. Fielding et al., Hypertext Transfer Protocol—HTTP/1.1, IETF RFC 2068, January 1997.
[10] A.O. Freier, P. Karlton, P.C. Kocher, The SSL Protocol—Version 3.0, IETF Internet Draft, November 1996.
[11] V. Gupta, Mozilla with Elliptic Curve Cryptography, 2004. https://dev.experimentalstuff.com:8443/MozillaWithECC.html.
[12] V. Gupta et al., ECC Cipher Suites for TLS, IETF Internet Draft draft-ietf-tls-ecc-10.txt, May 2005.
[13] V. Gupta et al., Sizzle: A standards-based end-to-end security architecture for the embedded internet, in: Proc. PerCom 2005, March 2005.
[14] V. Gupta, D. Stebila, S. Chang-Shantz, Integrating elliptic curve cryptography into the web's security infrastructure, in: Proc. 13th International World Wide Web Conference—Alternate Track Papers and Posters, May 2004.
[15] N. Gura et al., Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs, in: CHES'2004, August, in: Lecture Notes in Computer Science, Springer-Verlag, 2004.
[16] V. Jacobson, Compressing TCP/IP Headers for Low-Speed Serial Links, IETF RFC 1144, February 1990.
[17] C. Karlof, N. Sastry, D. Wagner, TinySec: A link layer security architecture for wireless sensor networks, in: ACM SenSys, November 2004.
[18] N. Koblitz, Elliptic Curve Cryptosystems, Mathematics of Computation 48 (1987) 203–209.
[19] A.K. Lenstra, E.R. Verheul, Selecting cryptographic key sizes, Journal of Cryptology: The Journal of the International Association for Cryptologic Research 14 (4) (2001) 255–293.
[20] V. Miller, Uses of Elliptic Curves in Cryptography, in: Advances in Cryptology, CRYPTO'85, in: Lecture Notes in Computer Science, vol. 218, Springer-Verlag, 1985, pp. 417–462.
[21] NIST, Special Publication 800–57: Recommendation for Key Management. Part 1: General Guideline, January 2003.
[22] PeerSec Networks, MatrixSSL—Open Source Embedded SSL. http://www.matrixssl.org/.

[23] A. Perrig, J. Stankovic, D. Wagner, Security in wireless sensor networks, Communications of the ACM (June) (2004).
[24] A. Perrig et al., SPINS: Security protocols for sensor networks, Wireless Networks 8 (December) (2002) 521–534.
[25] R. Szewczyk et al., Habitat monitoring with sensor networks, Communications of the ACM 47 (6) (2004) 34–40.
[26] The WAP Forum, http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html.
[27] TinyOS Community Forum, An open-source OS for the networked sensor regime. http://www.tinyos.net/.
[28] US Dept. of Commerce and National Institute of Standards and Technology, Digital Signature Standard (DSS), Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.
[29] S.A. Vanstone, Next generation security for wireless: Elliptic curve cryptography, Computers and Security 22 (5) (2003).
[30] R. Watro et al., TinyPK: Securing sensor networks with public key technology, in: Proc. 2nd ACM Workshop on Security of adhoc and Sensor Networks, ACM Press, 2004, pp. 59–64.
[31] M. Welsh, D. Myung, M. Gaynor, S. Moulton, Resuscitation monitoring with a wireless sensor network, Supplement to Circulation: Journal of the American Heart Association (October) (2003).
[32] A. Zingg, B. Lenzlinger, Mini Web Server supporting SSL. http://www.strongsec.com/zhw/DA/Sna3_2000.pdf, October, 2000.

Vipul Gupta is a Senior Staff Engineer at Sun Microsystems Laboratories (Sun Labs) where his research interests include Internet security and small wireless devices. He received his MS and PhD in Computer Science from Rutgers University and a BTech in Computer Science and Engineering from the Indian Institute of Technology, New Delhi.

Michael Wurm is graduate student of Computer Science and Telecommunications at the Graz University of Technology in Graz/Austria and did an internship at Sun Labs from January through August, 2005.

Yu (David) Zhu is an undergraduate student at the University of Waterloo and worked on Sizzle during an internship at Sun Labs from September through Decemeber, 2004.

Matthew Millard received his BASc in Systems Design Engineering from the University of Waterloo where he is currently a masters student. He was an intern at Sun Labs from April through August, 2004.

Stephen Fung received a BMath in Computer Science and Combinatorics & Optimization from the University of Waterloo. He worked on Sizzle during a student internship from January to April of 2004. He is currently pursuing a Masters in Computer Science at the University of Toronto.

Nils Gura joined Sun Labs in January 2001 and his areas of interest include efficient implementations of cryptographic algorithms, large-scale network switches and scheduling algorithms. Nils holds a MSc in Computer Science from the University of Ulm, Germany.

Hans Eberle is a Distinguished Engineer at Sun Labs where he works in the areas of networks and security. He received a diploma in Electrical Engineering and PhD in Technical Sciences from ETH Zurich in 1984 and 1987, respectively.

Sheueling Chang Shantz is a Distinguished Engineer at Sun Labs. Her research interests include online commerce and next-generation Internet security technologies. Sheueling has a PhD in Computer Science from California Institute of Technology.