Sizzle: A Standards-based end-to-end Security Architecture for the Embedded Internet

Vipul Gupta, Matthew Millard*, Stephen Fung*, Yu Zhu*,
Nils Gura, Hans Eberle, Sheueling Chang Shantz
Sun Microsystems Laboratories
16 Network Circle, UMPK16 160
Menlo Park, CA 94025
vipul.gupta@sun.com, mmillard@kos.net, fungstep@hotmail.com
davidyuzhu@hotmail.com, {nils.gura, hans.eberle, sheueling.chang}@sun.com

Abstract

This paper introduces Sizzle, the first fully-implemented end-to-end security architecture for highly constrained embedded devices. According to popular perception, public-key cryptography is beyond the capabilities of such devices. We show that elliptic curve cryptography (ECC) not only makes public-key cryptography feasible on these devices, it allows one to create a complete secure web server stack including SSL, HTTP and user application that runs efficiently within very tight resource constraints. Our small footprint HTTPS stack needs less than 4KB of RAM and interoperates with an ECC-enabled version of the Mozilla web browser. We have implemented Sizzle on the 8-bit Berkeley/Crossbow Mica2 "mote" platform where it can complete a full SSL handshake in less than 4 seconds (session reuse takes under 2 seconds) and transfer 450 bytes of application data over SSL in about 1 second. We present additional optimizations that can further improve performance. To the best of our knowledge, this is the world's smallest secure web server (in terms of both physical dimensions and resources consumed) and significantly lowers the barrier for connecting a variety of interesting new devices (e.g. home appliances, personal medical devices) to the Internet without sacrificing end-to-end security.

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 phase in the Internet's evolution when it 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 [26], 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 popular 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/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

*This work was performed while the authors were on a student internship from the Univ. of Waterloo, Canada.

[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) [8].

experiments involving RSA, today's dominant public-key cryptosystem [5].

First proposed by Victor Miller [19] and independently by Neal Koblitz [17] in the mid-1980s, Elliptic Curve Cryptography (ECC) is emerging as an attractive alternative to RSA for resource-constrained environments. Recent work in our research group has shown that it is possible to develop an efficient software implementation of ECC for 8-bit CPUs and bring the advantages of public-key cryptography to constrained devices where traditional alternatives like RSA are impractical [14].

On top of this ECC implementation, we have built a small-footprint secure web server stack (including HTTP and SSL), called Sizzle[2], that runs efficiently under tight resource constraints and interoperates with an ECC-enabled version of the Mozilla web browser [11]. The main contributions of this paper are:

- We describe the first fully-implemented, end-to-end security architecture for embedded devices.
- We describe the challenges posed by tight resource constraints on these devices and design choices we made to overcome them.
- We measure 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.

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. We present performance results and resource consumption statistics for Sizzle in Section 6. 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]. However, none of these efforts has produced an implementation suitable for highly constrained embedded platforms such as the 8-bit Berkeley/Crossbow Mica2 motes shown in Figure 1. The "mote" is particularly interesting because it is emerging as the preferred platform for much of sensor related research in academia and industry [28].

The Mini Web Server with SSL [32] targets the IPC@CHIP platform which has a 20MHz, 16-bit Intel 80186 processor, 512KB of Flash, 512KB of RAM and a built-in Ethernet connection. The SSL code size is around 100KB. MatrixSSL [22] has a smaller footprint (around 50-70KB), 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. Even on the more capable 16-bit CPU inside IPC@CHIP, RSA decryption takes nearly 45 seconds [32].

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

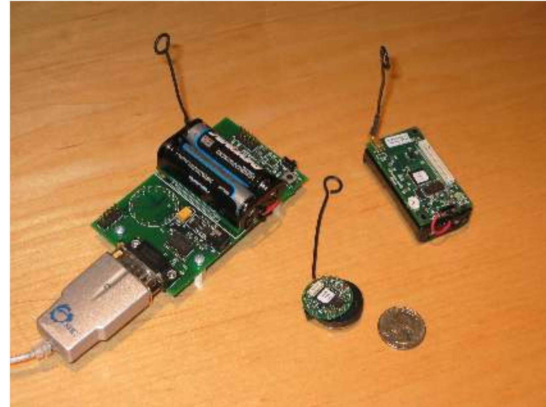

Figure 1. The Berkeley/Crossbow family of "mote" devices [7] (left to right): (a) development board/base station, (b) Mica2dot mote, and (c) Mica2 mote.

In contrast, Sizzle runs efficiently on both the Mica2 and Mica2dot motes: wireless, battery-powered devices with an 8-bit Atmel ATmega128L processor, 128KB of instruction memory, 4KB of EEPROM, 4KB of SRAM and a Chipcon CC1000 radio transceiver with a rated bandwidth of 19.2 kbits/s.

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 [9, 16, 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 public-key cryptography in the form of ECC.

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 attack ECC have fully exponential run times but the best known algorithms to attack RSA have sub-exponential run times [30], ECC can offer equivalent security with substantially smaller key

sizes [18, 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, 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.* [14], 160-bit ECDH operations are 13 times faster than 1024-bit RSA decryption operations on the mote, but 224-bit ECDH 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. In ECDH key agreement, two communicating parties A and B agree to use the same curve parameters. They generate their private keys k_A and k_B, and corresponding public keys $Q_A = k_A G$ and $Q_B = k_B G$. The parties exchange their public keys and each party multiplies its private key and the other's public key to arrive at a common shared secret $k_A Q_B = k_B Q_A = k_A k_B G$. While a description of ECDSA is not provided here, it similarly parallels DSA.

Recently, NIST approved ECC for use by the U.S. government [29]. Several standards organizations, such as IEEE, ANSI, OMA (Open Mobile Alliance) and the IETF, have ongoing efforts to include ECC 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*, *e.g.* the cipher suite *TLS_RSA_WITH_RC4_128_SHA* uses RSA for key exchange, 128-bit RC4 for bulk encryption, and SHA for hashing.

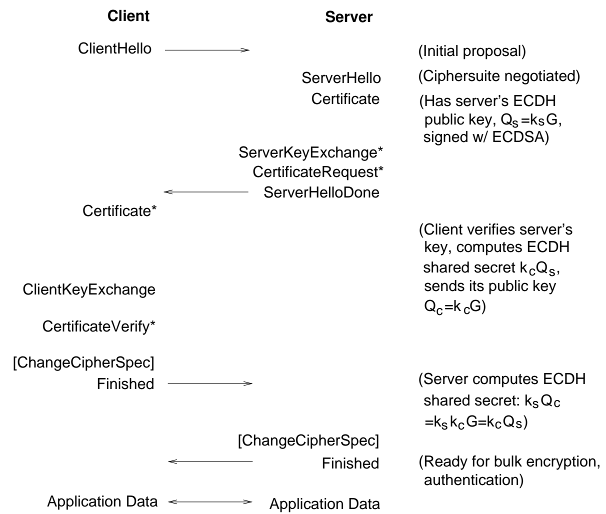

Figure 2. An ECC-based SSL handshake.

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 than a full handshake.

4.1. ECC-based Full Handshake

The general operation of an ECDH-ECDSA handshake, as specified in [12], is shown in Figure 2.[3] The client and server first exchange random values (used for replay protection) and negotiate a cipher suite using the *ClientHello* 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

[3]Messages marked with an asterisk are optional and only sent in certain protocol variants.

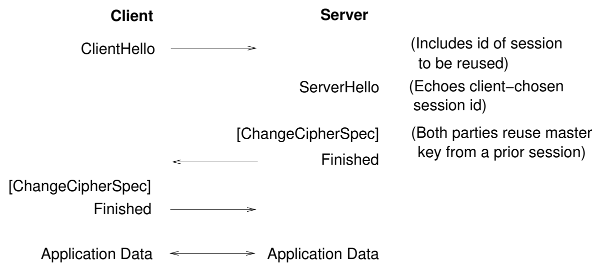

Figure 3. Abbreviated SSL handshake.

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 [13].

4.2. Abbreviated Handshake

The abbreviated handshake protocol is shown in Figure 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.[4] Otherwise, it returns a new session ID thereby signaling the client to engage in a full handshake. The derivation of symmetric keys from the master secret is identical to the full handshake scenario.

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 mote due to its reliance on public-key cryptography [16]. ECC makes public-key cryptography feasible on these devices, and Sizzle is our small footprint implementation of an HTTPS stack, built around ECC, 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. Figure 4 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.
- 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)
  - 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 [27], 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 so even if the latter is compromised, an attacker is unable to view or alter any data.

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

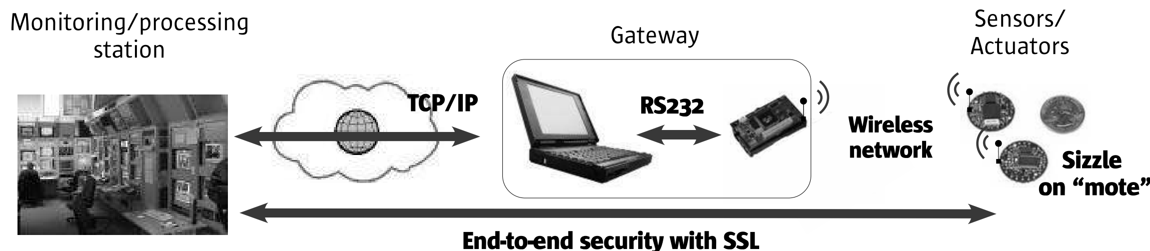

Figure 4. Gateway-based architecture for making embedded devices accessible across the Internet

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 uses ECC, which is more resource-efficient than RSA, for key exchange. In an effort to conserve program and data memory, only a single version (3.0) of the SSL protocol, a single cipher suite (ECDH-ECDSA-RC4-SHA) and a single elliptic curve (secp160r1) is enabled.

2. The cipher suite enabled in Sizzle does not entail sending the *ServerKeyExchange* message and the server's ECC public key is sent in a certificate. ECC keys are already smaller than RSA keys, but additional effort has been made to keep the certificate small. In particular, the subject and issuer names have been deliberately chosen to be brief and optional certificate extensions have been omitted resulting in a 222-byte certificate (the corresponding RSA certificate is over 400 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. In particular, tight memory constraints preclude the possibility of storing information for more than a few connections in the session cache and a four-byte identifier is more than sufficient to identify each cached session uniquely.

4. Sizzle does not send out the *CertificateRequest* message which automatically eliminates the client's *Certificate* and *CertificateVerify* messages. This obviates the need to have any certificate parsing code in Sizzle but does not preclude authentication of the client using passwords (one-time or otherwise) over SSL as is commonly done for on-line banking, stock trading and e-commerce.[5]

5. Static information such as the private key and corresponding certificate for the SSL server are stored in program memory, rather than data memory, to reduce SRAM usage (the mote has only 4KB of SRAM). The ATmega128 used on the mote allows the programmer to specify protected memory areas that cannot be accessed from the outside unless they are erased prior to the operation.

We have implemented the architecture shown in Figure 4 using mote devices for sensors/actuators and an ECC-enabled version of the Mozilla 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. Figure 5 shows some of the wireless devices built in our labs. We embedded a Mica2 mote with the Tiny OS operating system [28] running Sizzle inside a wireless thermostat whose settings can be read and modified using the Mozilla browser. Figure 6 shows a screenshot of Mozilla 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 show that the certificate contains an ECC public key.

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 implementation effort went into designing such a mechanism. We conducted experiments to study packet loss characteristics of the wireless link between the gateway and the mote. In one experiment, we transmitted 25000 packets and discovered that with Tiny OS 1.1.0, on average, 1 packet in 25 was lost/damaged. Radio transmission was considerably better in Tiny OS 1.1.6 where, on average, just 1 packet in 3125 had a transmission error. Even the (unreliable) data transmission rate was better with Tiny OS 1.1.6 (28 pkts/s) relative to Tiny OS 1.1.0 (10 pkts/s).[6] The improved data

[5]We have implemented this feature in a watch that can be used to monitor the wearer's body temperature and pulse rate. This information is revealed only to authorized entities.

[6]In both instances, however, the observed rate was significantly lower than the theoretical peak of 53 pkts/s for Tiny OS 1.1.6 and 42 pkts/s for

Figure 5. ECC-enabled wireless devices built at Sun Labs: (a) thermostat, (b) health-monitoring watch (c) magnetic stripe reader.

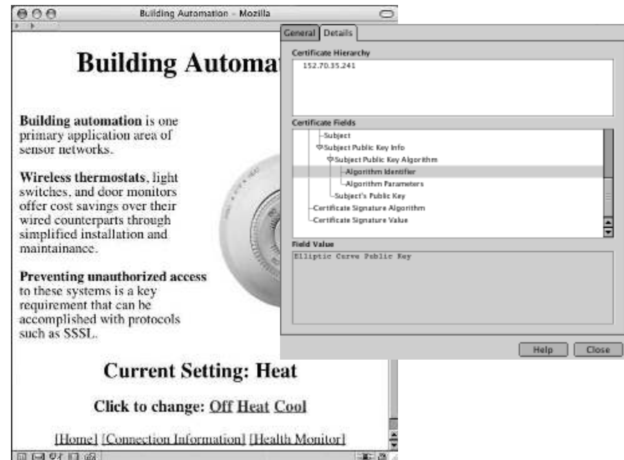

Figure 6. A screenshot of an ECC-enabled Mozilla browser communicating with Sizzle embedded inside a wireless thermostat. Images linked to the HTML page are served from another location, *i.e.* to conserve memory, these images are not stored inside Sizzle.

rate and reliability of Tiny OS 1.1.6 can be attributed to a number of improvements in the radio stack including a reduced preamble and tweaks to the backoff timer [25]. Our first prototype of a reliable data transfer scheme used a simple stop-and-go ACK-based protocol. However, since Tiny OS uses fixed-size packets, sending one ACK packet per data packet halved the effective throughput. Our current prototype transmits the first and last packets in a data block with explicit ACKs. All 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 after an HTTPS request has been fulfilled, and (iii) READY indicates a mote's readiness to receive the next SSL record from the gateway. All three are sent with explicit ACKs.

SSL records 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 29 bytes, but our reliable transmission scheme uses 6 of those bytes for housekeeping chores like sequence numbers and return addresses.[7] Therefore, an exemplary 286-byte record containing the *ServerHello*, *Certificate* and *ServerHelloDone* messages is fragmented into $\lceil 286/23 \rceil$=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. Although the gateway terminates the TCP/IP connection (see Figure 4), the security offered by SSL is end-to-end.

Tiny OS 1.1.0.

[7] There is considerable room for optimization here.

6. Experimental Results

Running the objdump utility on the Sizzle binary compiled for the ATmega128 shows that we use about 60KB of program memory (out of the available 128KB) and consume nearly 3150 out of the 4096 bytes of SRAM for global or static objects.[8] These numbers include resources used by Tiny OS, our reliable transmission mechanism, SSL with ECDH, RC4, SHA1 and MD5, and the HTTP/application layer. On a Linux x86 platform, the size of the stripped Sizzle binary is under 27KB. This includes code/data used by the MD5, SHA1 and RC4 algorithms, the SSL and HTTP layers and a simple web page. It does not include ECDH for which we currently do not have an x86 assembly implementation.[9] The ECDH code in optimized AVR assembly takes up 3682 bytes and we estimate the x86 assembly size to be well under 4KB. Thus the estimated x86 binary size of our complete secure web server is below 31KB. In comparison, the x86 binary size of MatrixSSL (only SSL and cryptographic algorithms) is around 50KB [22].

We used tcpdump to monitor network traffic between a Mozilla browser and Sizzle and used packet timestamps to calculate the total time in fulfilling an HTTPS request (including TCP connection setup, SSL handshake, HTTP request/response and TCP teardown). We performed these experiments on two platforms (Mica2 and Mica2dot[10]), with and without session reuse while transferring differ-

[8] Dynamic memory usage is slightly higher due to stack structures.

[9] Our MD5, SHA1 and RC4 implementations are written in C.

[10] The quarter-sized Mica2dot runs at 4MHz while the slightly larger Mica2 runs at 7.37MHz.

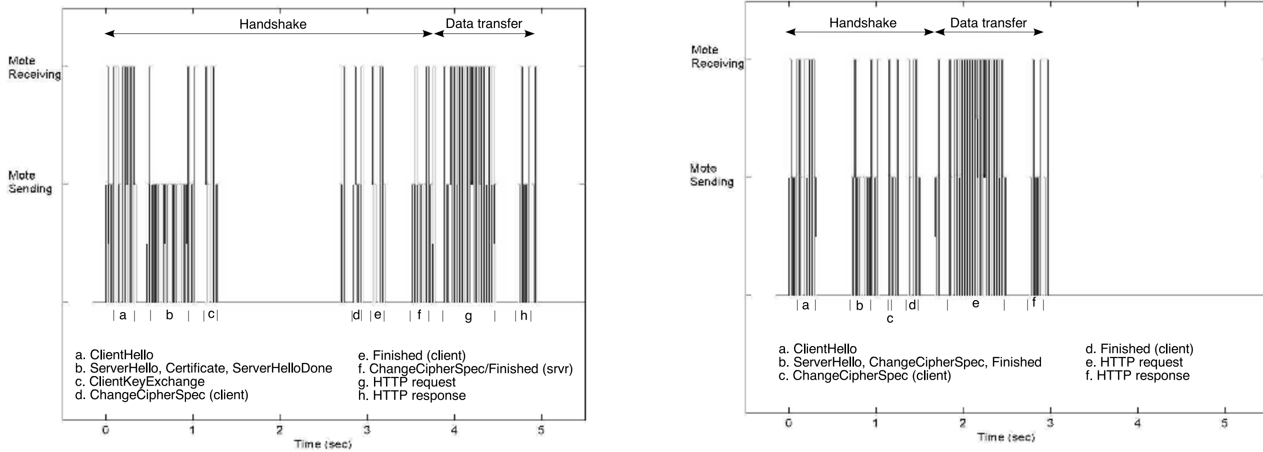

Figure 7. Tiny OS packet exchange for transferring 450 bytes of data over HTTPS using (a) Full, and (b) Abbreviated handshakes. The first two pulses depict NEWCONNECTION/ACK and the last two show DISCONNECT/ACK.

ent amounts of application data (˜450 bytes and ˜1350 bytes).[11] Our results are shown in Table 1. The numbers in parentheses indicate SSL handshake latency as measured between the *ClientHello* and the last *Finished* message. On the Mica2, a full handshake takes less than 4 seconds and an abbreviated handshake takes less than 1.2 seconds. The handshake overhead can be reduced further by using persistent HTTP connections that enable multiple request/response transactions within a single TCP connection (and handshake). On a Mica2dot, whose CPU runs 1.8 times slower, a full handshake takes around 5.6 seconds and an abbreviated handshake around 1.6 seconds. This indicates that data transmission is a fairly significant component of the overall latency and reducing CPU time will have a limited impact.

Table 1. Time (in seconds) for a complete HTTPS transaction. SSL handshake time is in parentheses.

	Application data size (request + response)			
	450 bytes		1350 bytes	
	Full	Abbrev.	Full	Abbrev.
Mica2	4.9 (3.8)	3.3 (1.2)	6.6 (3.9)	4.8 (1.1)
Mica2dot	7.0 (5.6)	3.7 (1.5)	8.8 (5.6)	5.6 (1.4)

Figure 7 shows the timing of Tiny OS packets in HTTPS transactions based on filtered measurements from the receive signal strength indicator (RSSI) pin of the CC1000 transceiver at the gateway. 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 the reliable transmission scheme described earlier (including the use of NEWCONNECTION, DISCONNECT, READY and ACK packets[12]). Gaps without pulses represent periods when Sizzle is busy computing or waiting for data, *e.g.* the largest gap in Figure 7(a) corresponds to Sizzle processing the *ClientKeyExchange* message. This involves computing the premaster secret via an ECDH operation and the derivation of symmetric keys for use by the record layer. Figures 7(a) and (b) also show that the time to symmetrically encrypt/decrypt application data and compute/verify its message authentication code is a small portion of its transmission time.[13] Our reliable transmission scheme has a fairly high overhead especially when small data blocks are involved. For example, in a full handshake, 21 signaling packets are sent for 26 data packets 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.

The next generation of mote-like devices (MicaZ [7], Te-

[11] More than 400 of these bytes are from the default Mozilla HTTP request header and this highlights the benefits of using a customized (or customizable) SSL client for communicating with embedded devices.

[12] These samples do not show any NACKs.

[13] The gap between intervals g and h in Figure 7(a) is small compared to the sum of those intervals.

```
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
---
```

Figure 8. Byte-level contents of messages in a Full Handshake

los [20]) will have an IEEE 802.15.4 radio with much higher data rates (250 kbps) such that we expect significantly lower connection times. We are evaluating these new platforms with respect to their wireless characteristics and a possible redesign of our reliable transmission scheme.

We observed that data transmission consumes significantly more energy than computation. For example, transmitting 1 bit consumes as much energy as running the CPU for over 2000 cycles. With the optimizations described in Section 5, the total number of bytes exchanged in a full SSL handshake is under 600 bytes. In contrast, for an RSA-based handshake, the server's certificate alone is typically more than 600 bytes. Section 6.1 describes ways in which the amount of data transmitted to establish an SSL connection can be further reduced.

6.1. Potential improvements

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. Figures 8 and 9 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 pre-configured 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 *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.[14] This data suppression/recreation functionality can be implemented in special modules inserted between the SSL record handler and the wireless 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 2).

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

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

```
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
---
```

Figure 9. Byte-level contents of messages in an Abbreviated Handshake

Uncompressed Message(s)	Compressed Representation
ClientHello (55*)	ClientHelloTemplateID (1), cipher suite len(2), cipher suite list (12*), 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 co-ordinate (20), disambiguator for y co-ordinate (1)
ChangeCipherSpec (6) *Finished (65)*	CCSFinishedTemplateID (1), encrypted payload of Finished message (60)

(a) Full Handshake

Uncompressed Message(s)	Compressed Representation
ClientHello (58*)	ClientHelloTemplateID (1), cipher suite len(2), cipher suite list (6*), 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)

(b) Abbreviated Handshake

Table 2. Contents of handshake messages after passing through the suppressor module. Size in bytes shown within parentheses (*actual value depends on client's configuration).

(*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 [6, 15] 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 public-key cryptography, in the form of ECC, 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, personal medical devices and monitor/control them securely across the Internet.

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 (ECDSA)", American Bankers Association, 1999.

[3] A. Bakre and B. R. Badrinath, "I-TCP: Indirect TCP for mobile hosts", *15th International Conference on Distributed Computing Systems*, 1995.

[4] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A comparison of mechanisms for improving TCP performance over wireless links", *IEEE/ACM Transactions on Networking*, 5(6):756-769, 1997.

[5] BBN Technologies, "TinyPK Project", http://www.is.bbn.com/projects/lws-nest/.

[6] C. Bormann *et al.*, "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed", IETF *RFC 3095*, Jul. 2001.

[7] Crossbow Technology, Inc., see http://www.xbow.com/Products/products.htm.

[8] T. Dierks and C. Allen, "The TLS Protocol — Version 1.0", IETF *RFC 2246*, Jan. 1999.

[9] L. Eschenauer and 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.

[10] A. O. Freier, P. Karlton, and P. C. Kocher, "The SSL Protocol — Version 3.0", IETF internet-draft, Nov. 1996.

[11] V. Gupta, "Mozilla with Elliptic Curve Cryptography", http://dev.experimentalstuff.com:8081/MozillaWithECC.html.

[12] V. Gupta, S. Blake-Wilson, B. Möller, C. Hawk, and N. Bolyard, "ECC Cipher Suites for TLS", IETF internet-draft, Dec. 2004.

[13] V. Gupta, D. Stebila, and S. Chang-Shantz, "Integrating Elliptic Curve Cryptography into the Web's Security Infrastructure", In *The 13th International World Wide Web Conference – Alternate Track Papers and Posters*, New York City, May 2004, pp. 402-403.

[14] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Chang Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs", *CHES 2004*, Aug. 2004.

[15] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links", IETF *RFC 1144*, Feb. 1990.

[16] C. Karlof, N. Sastry, and D. Wagner, "TinySec: A Link Layer Security Architecture for Wireless Sensor Networks", *ACM SenSys*, Nov. 2004.

[17] N. Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, 48:203–209, 1987.

[18] A. K. Lenstra and E. R. Verheul, "Selecting Cryptographic Key Sizes", *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001.

[19] V. Miller, "Uses of Elliptic Curves in Cryptography", In *Advances in Cryptology, CRYPTO'85*, LNCS 218, Springer-Verlag, 1985, pp. 417–462.

[20] Moteiv Corporation, "Telos product information", http://www.moteiv.com/products/.

[21] NIST, "Special Publication 800-57: Recommendation for Key Management. Part 1: General Guideline", Jan. 2003.

[22] PeerSec Networks, "MatrixSSL – Open Source Embedded SSL", see http://www.matrixssl.org/.

[23] A. Perrig, J. Stankovic, and D. Wagner, "Security in Wireless Sensor Networks", *Communications of the ACM*, Jun. 2004.

[24] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "SPINS: Security Protocols for Sensor Networks", *Wireless Networks*, 8:521–534, Dec. 2002.

[25] J. Polastre, "Radio Stack Iteration: How to improve the CC1000", Jan. 2004, see http://webs.cs.berkeley.edu/retreat-1-04/ slides/joep-nest-cc1000.pdf.

[26] R. Szewczyk *et al.*, "Habitat Monitoring with Sensor Networks", *Communications of the ACM*, 47(6):34–40, Jun. 2004.

[27] The WAP Forum, see http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html.

[28] TinyOS Community Forum, "An open-source OS for the networked sensor regime", http://www.tinyos.net/.

[29] U.S. Dept. of Commerce and NIST, "Digital Signature Standard (DSS)", *FIPS PUB 186-2*, Jan. 2000.

[30] S. A. Vanstone, "Next Generation Security for Wireless: Elliptic Curve Cryptography", *Computers and Security*, 22(5), Aug. 2003.

[31] M. Welsh, D. Myung, M. Gaynor, and S. Moulton, "Resuscitation monitoring with a wireless sensor network", *Supplement to Circulation: Journal of the American Heart Association*, Oct. 2003.

[32] A. Zingg and B. Lenzlinger, "Mini Web Server supporting SSL", Oct. 2000, http://www.strongsec.com/zhw/DA/Sna3_2000.pdf.