IST-2002-507932

ECRYPT

European Network of Excellence in Cryptology

Network of Excellence
Information Society Technologies

D.VAM.13

eBATS—Benchmarking of Asymmetric Systems
(benchmarking tool and call for contribution)

Due date of deliverable: 30. June 2006
Actual submission date: 24. February 2007

Start date of project: 1. February 2004  
Duration: 4.5 years

Lead contractor: G+

Revision 1.0

<table>
<thead>
<tr>
<th>Dissemination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including the Commission services)</td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including the Commission services)</td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including the Commission services)</td>
</tr>
</tbody>
</table>

Project co-funded by the European Commission within the 6th Framework Programme
The work described in this report has in part been supported by the Commission of the European Communities through the IST program under contract IST-2002-507932. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.
Contents

1 The BATMAN software package ............................................. 3
2 The call for contributions of encrypting BATs ......................... 5
3 The call for contributions of signing BATs ......................... 13
4 The call for contributions of secret-sharing BATs ................... 21
Executive summary

Users of public-key cryptography have a choice of public-key cryptosystems, including RSA, DSA, ECDSA, and many more. Exactly how fast are these systems? How do the speeds vary among Pentium, PowerPC, etc.? How much network bandwidth do the systems consume? The eBATS (ECRYPT Benchmarking of Asymmetric Systems) project aims to answer these questions.

This deliverable consists of two parts of eBATS: (1) BATMAN (Benchmarking of Asymmetric Tools on Multiple Architectures, Non-Interactively), a software package that allows a large number of asymmetric tools to be systematically measured on a large number of computers, producing an extensive database of measurements in a form suitable for easy computer processing; and (2) a call for contributions of BATs (Benchmarkable Asymmetric Tools) for benchmarking with the BATMAN framework, including detailed descriptions of the interfaces for encrypting BATs, signing BATs, and secret-sharing BATs.
Chapter 1

The BATMAN software package

eBATS (ECRYPT Benchmarking of Asymmetric Systems), run by the VAMPIRE lab, is a benchmarking project for public-key cryptography.

One component of eBATS is BATMAN (Benchmarking of Asymmetric Tools on Multiple Architectures, Non-interactively), a software package to collect measurements of public-key systems. BATMAN allows a large number of BATs (Benchmarkable Asymmetric Tools) to be systematically measured on a large number of computers. The output of BATMAN is an extensive database of measurements in a form suitable for easy computer processing. As examples, here are 8 lines selected from the 3227696 lines in the current database:

20070214 katana 20070215 ronald 3 2048 cpuid - GenuineIntel-000006f6-bfebfbff_
20070214 katana 20070215 ronald 3 2048 keypair - cycles 684835224 655209584 175447872 433535904 463250784 793169824 1041142424 798791040 921257672 684835224 827075496 328437320 472327336 510686104 1548770536

20070214 katana 20070215 ronald 3 2048 ciphertext 96397 cycles 4174008 4189592 4181048 4174008 4175112 4171456 4177008 4170968 4175576 4170648 4176088 4169248 4174032 4168936 4173472 4170048

20070214 katana 20070215 ronald 3 2048 ciphertext 96397 bytes 96472 96472 96472 96472 96472 96472 96472 96472

20070214 katana 20070215 ronald 3 2048 plaintext 96397 cycles 18519328 18392760 18445656 18539520 18546000 18560048 18487552 18510096 18430040 18547880 18519328 18534704 18445016 18529240

20070214 katana 20070215 ronald 3 2048 signedmessage 10348 cycles 14700272 14619296 14644312 14670040 14681616 14697656 14797160 14700272 14748208

20070214 katana 20070215 ronald 3 2048 signedmessage 10348 bytes 10391 10391 10391 10391 10391 10391 10391 10391

BATMAN was developed within VAMPIRE and was released to the public on 15 June 2006, along with several sample BATs (CLAUS, CLAUS++, RONALD version 1, RONALD version 2, and RONALD version 3). Improved versions of BATMAN were released on several occasions, most recently on 14 February 2007. Figure 1.1 is a copy of the 14 February 2007 version of the BATMAN web page, one of the eBATS web pages available from http://www.ecrypt.eu.org/ebats.
**EBATS: ECRYPT Benchmarking of Asymmetric Systems**

---

**BATMAN**

eBATS (ECRYPT Benchmarking of Asymmetric Systems) is a competition for public-key systems. eBATS measures BATS (Benchmarkable Asymmetric Tools) according to several criteria: time to generate a key pair, time to sign a message, length of the signed message, etc.

**Using BATMAN to test your new BATS**

Download and unpack the BATMAN program:

```bash
wget http://hyperelliptic.org/ebats/batman-20070214.tar.gz
gunzip -v batman-20070214.tar.gz
cd batman-20070214
```

Download the available BATS as examples, and move them to the sleepingbats directory:

```bash
wget http://hyperelliptic.org/ebats/bats-20070214.tar.gz
gunzip -v bats-20070214.tar.gz
mv bats/* sleepingbats
```

Download m4 1.4.8, GMP 4.2.1, NTL 5.4, and OpenSSL 0.9.8d:

```bash
(cd libraries; wget -h ftp://ftp.gnu.org/gnu/m4/m4-1.4.8.tar.gz)
(cd libraries; wget -h ftp://ftp.gnu.org/gnu/gmp/gmp-4.2.1.tar.gz)
(cd libraries; wget https://sourceforge.net/projects/ntl/source)
(cd libraries; wget http://www.openssl.org/source/openssl-0.9.8d.tar.gz)
```

Your BAT is free to call functions from GMP, NTL, and OpenSSL after an appropriate #include <gmp.h> etc. Other libraries may be integrated into BATMAN upon request.

Choose a name for your BAT; let's say it's *furry*. version 1. Create a new directory for your BAT:

```bash
mkdir bats/furry-1
cd bats/furry-1
```

Inside your new directory, create the files required by the BAT API, such as sizes.h.

Once you're done writing your BAT, measure it using BATMAN:

```bash
cd ...
```

./do leaves the resulting measurements in a new file 20070214.-hostname. You can monitor the progress of ./do by watching the file 20070214.-hostname- notes.

Beware that the ./do script compiles GMP, NTL, and OpenSSL before it begins measurements. This takes 15 minutes on a 2000MHz Athlon 64, and might take much longer on your machine.

Once you're ready to submit your BAT to eBATS, put a tarball of the subdirectory on the web, and send the URL to batsubmission@ebats-cr.yp.to.

**Contributing computer time to eBATS**

Do you have a computer that has enough time to benchmark all the available BATS (with no other tasks consuming CPU power), and that will have time in the future for updated benchmarks? Would you like to contribute CPU cycles to benchmarking? Perhaps your favorite type of computer isn't included in the current list of eBATS platforms. Even if all of your computers are similar to computers in the list, you can help by providing independent verification of the speed measurements.

To measure all the available BATS, simply download, unpack, and run the BATMAN program, along with GMP, NTL, OpenSSL, and the BATS:

```bash
wget http://hyperelliptic.org/ebats/batman-20070214.tar.gz
gunzip -v batman-20070214.tar.gz
```

Put the resulting 20070214* files on the web, and send the URLs to batmanresults@ebats-cr.yp.to.
Chapter 2

The call for contributions of encrypting BATs

On 16 June 2006, VAMPIRE issued a call for submission of public-key-encryption systems for benchmarking. See Figures 2.1 through 2.7.
Call for public-key-encryption software for benchmarking

eBATS (ECRYPT Benchmarking of Asymmetric Systems) is a competition for the most efficient public-key systems. eBATS measures public-key-encryption systems according to the following criteria:

- Time to generate a key pair.
- Length of the secret key.
- Length of the public key.
- Time to encrypt a message using the public key.
- Length of the encrypted message.
- Time to decrypt the encrypted message using the secret key.

"Time" refers to time on real computers: cycles on a Pentium III 686, cycles on a PowerPC G5, cycles on an Athlon 64 X2, etc. eBATS times each system on a wide variety of computers, ensuring direct comparability of all systems on whichever computers are of interest to the users. Tools to graph the results will be made available.

This page explains how cryptographers can submit implementations of public-key-encryption systems (RSA, McEliece, NTRU, etc.) to eBATS.

Introduction to the eBATS encrypting API

Formal submission requirements have been kept to a minimum. Your software has to be an encrypting BAT (Benchmarkable Asymmetric Tool), meaning that it supports the following three functions:

- keypair, returning a secret key and a public key;
- ciphertext, reading a message and a public key, returning an encrypted message; and
- plaintext, reading an encrypted message and a secret key, returning the message that was encrypted.

Don’t want to handle arbitrary-length messages? No problem. Instead of implementing ciphertext and plaintext, you can implement shortciphertext and shortplaintext, and define the maximum length of a short plaintext. BATMAN, the eBATS benchmarking software, will automatically handle longer plaintexts using a stream cipher.

You can also provide additional functions that document additional features of your system:

- distinguishingchance, documenting resistance to the best single-key attack known.
- multiplekeydistinguishingchance, documenting resistance to the best multiple-key attack known.
- cca, documenting resistance to chosen-plaintext attacks.
- timingattacks, documenting resistance to timing attacks.
- copyrightclaims, documenting copyright claims against distribution of the software.
- patentclaims, documenting patent claims against use of the software.

Claims regarding these additional features do not have the same level of verifiability as the eBATS measurements of key size, encryption time, etc.; eBATS will nevertheless report these claims for public discussion.

The eBATS encrypting API is described below in more detail. There’s a separate page on BATMAN; you will be able to download and use BATMAN before submission to check that your implementation works properly. There’s also a separate page discussing security evaluations in more detail.

Files in an encrypting BAT

An encrypting BAT is a tar.gz file containing one directory. The directory contains a file sizes.h, any number of additional *.s, *.c, and *.cpp files implementing the eBATS encrypting API, and a file documentation.pdf with references and other comments for cryptographers.

The directory name is the BAT name followed by a dash and a version number: e.g., ronald-1 for a BAT named ronald, version 1. eBATS will rename BATs if there is a conflict in names.

The file sizes.h defines various macros discussed below: SECRETKEY_BYTES, PUBLICKEY_BYTES, ENCRYPTION_BYTES, and optionally SHORTPLAINTEXT_BYTES.

BATMAN will automatically decide whether the BAT is a C BAT, providing the eBATS API functions in C, or a C++ BAT, providing the eBATS API functions in C++. Either way, the BAT can call C functions in its *.c files and assembly-language functions in its *.s files. BATs written in other languages have to be compiled to C++, C, or assembly language.

Parametrized BATs

Some BATs allow parameters. For example, a typical RSA implementation allows a wide range of key sizes. On
the other hand, some RSA implementations gain speed by focusing on particular key sizes.

The eBATS API can support BATs of either type. A parametrized BAT includes, in the same directory as sizes.h, a parameters file with several lines; each line specifies compilation options that select a particular parameter choice. A parameter choice is specified by BAT-specific macros, which are used by sizes.h etc., and by a PARAMETERS macro (without white space), which is used to identify parameters in the eBATS results.

For example, version 1 of the RONALD BAT has a 29-line parameters file starting

```
-DMODULUS_BITS=768 -DPARAMETERS="768"
-DMODULUS_BITS=832 -DPARAMETERS="832"
-DMODULUS_BITS=896 -DPARAMETERS="896"
-DMODULUS_BITS=960 -DPARAMETERS="960"
-DMODULUS_BITS=1024 -DPARAMETERS="1024"
```

and continuing (in roughly geometric progression) until

```
-DMODULUS_BITS=4096 -DPARAMETERS="4096"
```

The MODULUS_BITS macro controls PUBLICKEY_BYTES etc. through the lines

```
#define MODULUS_BYTES (MODULUS_BITS / 8)
#define PUBLICKEY_BYTES (MODULUS_BYTES)
```

in the sizes.h file. The PARAMETERS macro is printed in the eBATS measurements.

The parameters file can omit -DPARAMETERS=... if sizes.h defines PARAMETERS. For example, version 2 of the RONALD BAT has a 29-line parameters file starting

```
-DMODULUS_BITS=768
-DMODULUS_BITS=832
-DMODULUS_BITS=896
-DMODULUS_BITS=960
-DMODULUS_BITS=1024
```

and the following lines in sizes.h:

```
#define XSTRINGIFY(N) #N
#define STRINGIFY(N) XSTRINGIFY(N)
#define PARAMETERS (STRINGIFY(MODULUS_BITS))
```

**Tuned BATs**

A BAT can contain several implementations of the same functions: e.g., a P4-tuned implementation, a G5-tuned implementation, etc. A tuned BAT includes, in the same directory as sizes.h, a tunings file with several lines; each line specifies compilation options that select a particular tuning. A tuning is specified by BAT-specific macros, which are used by sizes.h etc., and by a TUNING macro (without white space), which is used to identify tuning in the eBATS results.

BATMAN will automatically try each tuning and select the tuning where ciphertext runs most quickly. A BAT can define a TUNETARGET macro in sizes.h; in that case BATMAN will select the tuning where TUNETARGET() runs most quickly.

Any particular tuning is allowed to be unportable, failing to compile on most platforms. BATMAN will skip tunings that don't compile or that flunk some simple tests.

**Generating random numbers**

BATMAN sets up file descriptor 0 reading from a neverending source of hard-to-predict secret random bytes. BATs are free to assume this: the keypair function, for example, can obtain secret bytes using getchar().

Functions are permitted, but not encouraged, to generate randomness in other ways, such as by opening /dev/urandom. These functions won't be benchmarkable on systems that don't have /dev/urandom, and they won't be suitable for black-box regression testing.

**Using hash functions**

BATMAN provides a cryptographic hash function hash256 callable from a BAT as follows:

```c
const unsigned char m[...]; unsigned long long mlen;
unsigned char h[32];
hash256(h, m, mlen);
```

hash256 hashes bytes m[0], m[1], ..., m[mlen-1] and puts the output into h[0], h[1], ..., h[31]. Currently

Figure 2.2: The call for encrypting BATs, version 2006.06.16, page 2.
hash256 is implemented as SHA-256.

To simplify comparisons of public-key systems, eBATS recommends that BATs use hash256 for all necessary hashing. This is not a recommendation of SHA-256 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different hash functions.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks that work with any hash function. Any security problems in SHA-256 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

**Using stream ciphers**

BATMAN provides an additive stream cipher stream256 callable from a BAT as follows:

```c
const unsigned char m[...]; unsigned long long mlen;
unsigned char c[...];
const unsigned char k[32];
const unsigned char n[8];
stream256(c,m,mlen,k,n);
stream256 encrypts (or decrypts) bytes m[0], m[1], ..., m[mlen-1] and puts the output into c[0], c[1], ..., c[mlen-1]. It uses a 32-byte key k[0], k[1], ..., k[31] and an 8-byte nonce n[0], n[1], ..., n[7]. Currently stream256 is implemented as Salsa20.
```

To simplify comparisons of public-key systems, eBATS recommends that BATs use stream256 for all necessary stream generation. This is not a recommendation of Salsa20 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different ciphers.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks that work with any stream cipher. Any security problems in Salsa20 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

**keypair: generate a new secret key and public key**

An encrypting BAT must provide a keypair function callable as follows:

```c
#include "sizes.h"

unsigned char sk[SECRETKEY_BYTES]; unsigned long long sklen;
unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
keypair(sk,&sklen,pk,&pklen);
```

The keypair function generates a new secret key and a new public key. It puts the number of bytes of the secret key into sklen; puts the number of bytes of the public key into pklen; puts the secret key into sk[0], sk[1], ..., sk[sklen-1]; and puts the public key into pk[0], pk[1], ..., pk[pklen-1]; it then returns 0.

keypair guarantees that sklen is at most SECRETKEY_BYTES, and that pklen is at most PUBLICKEY_BYTES, so that the caller can allocate enough space.

If key generation is impossible for some reason (e.g., not enough memory), keypair returns a negative number, possibly after modifying sk[0], sk[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

**ciphertext: encrypt a message using a public key**

An encrypting BAT can provide a ciphertext function callable as follows:

```c
#include "sizes.h"

const unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
const unsigned char m[...]; unsigned long long mlen;
unsigned char c[...]; unsigned long long clen;
ciphertext(c,&clen,m,mlen,pk,pklen);
```

The ciphertext function uses a public key pk[0], pk[1], ..., pk[pklen-1] to encrypt a message m[0], m[1], ..., m[mlen-1]. It puts the length of the encrypted message into clen and puts the encrypted message into c[0], c[1], ..., c[clen-1]; it then returns 0.

The ciphertext function guarantees that clen is at most mlen+ENCRYPTION_BYTES. The ENCRYPTION_BYTES macro is defined in sizes.h.

Figure 2.3: The call for encrypting BATs, version 2006.06.16, page 3.
The ciphertext function is free to assume that the public key pk[0], pk[1], ..., pk[pklen-1] was generated by a successful call to the keypair function.

If encryption is impossible for some reason, ciphertext returns a negative number, possibly after modifying c[0], c[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

Implementors of the ciphertext function are warned that they should not go to extra effort to compress the message m. Higher-level applications should be presumed to compress messages before calling the ciphertext function; in particular, BATMAN uses random messages to make compression ineffective. On the other hand, the encrypted message c is longer than the original message m and might be compressible; any reduction of the encryption overhead will be visible in the eBATS measurements.

plaintext: decrypt a message using a secret key

An encrypting BAT can provide a plaintext function callable as follows:

```c
#include "sizes.h"
const unsigned char sk[SECRETKEY_BYTES];
const unsigned char c[...];
unsigned char m[...];
plaintext(m,&mlen,c,clen,sk,sklen);
```

The plaintext function uses a secret key sk[0], sk[1], ..., sk[sklen-1] to decrypt a ciphertext c[0], c[1], ..., c[clen-1]. The plaintext function puts the length of the decrypted message into mlen, puts the decrypted message into m[0], m[1], ..., m[mlen-1], and returns 0.

The plaintext function guarantees that mlen is at most clen.

The plaintext function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the secretkey function.

If decryption is impossible for some reason, plaintext returns a negative number, possibly after modifying m[0], m[1], etc. Current implementations should return -100 for invalid ciphertexts, and -1 for all other problems; other return values with special meanings may be defined in the future.

shortciphertext: encrypt a message using a public key

An encrypting BAT can provide a shortciphertext function callable as follows:

```c
#include "sizes.h"
const unsigned char pk[PUBLICKEY_BYTES];
const unsigned char m[SHORTPLAINTEXT_BYTES];
shortciphertext(c,&clen,m,mlen,pk,pklen);
```

The shortciphertext function uses a public key pk[0], pk[1], ..., pk[pklen-1] to encrypt a message m[0], m[1], ..., m[mlen-1]. It puts the length of the encrypted message into clen and puts the encrypted message into c[0], c[1], ..., c[clen-1]. It then returns 0.

The shortciphertext function is free to assume that mlen is at most SHORTPLAINTEXT_BYTES. The shortciphertext function guarantees that clen is exactly ENCRYPTIONBYTES. The SHORTPLAINTEXT_BYTES and ENCRYPTIONBYTES macros are defined in sizes.h.

The shortciphertext function is free to assume that the public key pk[0], pk[1], ..., pk[pklen-1] was generated by a successful call to the keypair function.

If encryption is impossible for some reason, shortciphertext returns a negative number, possibly after modifying c[0], c[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

Implementors of the shortciphertext function are warned that they should not go to extra effort to compress the message m. Higher-level applications should be presumed to compress messages before calling the shortciphertext function; in particular, BATMAN uses random messages to make compression ineffective. On the other hand, the encrypted message c is longer than the original message m and might be compressible; any reduction of the encryption overhead will be visible in the eBATS measurements.

BATMAN automatically builds ciphertext on top of shortciphertext as follows. Messages with at most SHORTPLAINTEXT_BYTES - 1 bytes are simply encrypted with shortciphertext. A message with SHORTPLAINTEXT_BYTES or more bytes is handled as follows:

Figure 2.4: The call for encrypting BATs, version 2006.06.16, page 4.
The message is encrypted with Salsa20 using a random 32-byte key, producing an initial encryption e. The 32-byte Salsa20 key, the 32-byte SHA-256 hash of e, and the first SHORTPLAINTEXT_BYTES-64 bytes of e are encrypted with shortciphertext. The rest of e is appended.

SHORTPLAINTEXT_BYTES must be at least 64. Criticisms of the speed and security of Salsa20 and SHA-256 are outside the scope of ebATS; ebATS focuses on public-key cryptography, not on stream ciphers and hash functions.

**shortplaintext: decrypt a message using a secret key**

An encrypting BAT can provide a shortplaintext function callable as follows:

```c
#include "sizes.h"

const unsigned char sk[SECRETKEY_BYTES];
const unsigned long long sklen;
const unsigned char c[ENCRYPTION_BYTES];
const unsigned long long clen;
unsigned char m[SHORTPLAINTEXT_BYTES];
unsigned long long mlen;

shortplaintext(m, &mlen, c, clen, sk, sklen);
```

The shortplaintext function uses a secret key sk[0], sk[1], ..., sk[sklen-1] to decrypt a ciphertext c[0], c[1], ..., c[clen-1]. The shortplaintext function puts the length of the decrypted message into mlen, puts the decrypted message into m[0], m[1], ..., m[mlen-1], and returns 0.

The shortplaintext function is free to assume that clen is exactly ENCRYPTION_BYTES. The shortplaintext function guarantees that mlen is at most SHORTPLAINTEXT_BYTES.

The shortplaintext function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the secretkey function.

If decryption is impossible for some reason, plaintext returns a negative number, possibly after modifying m[0], m[1], etc. Current implementations should return -100 for invalid ciphertexts, and -1 for all other problems; other return values with special meanings may be defined in the future.

BATMAN automatically builds plaintext on top of shortplaintext by reversing the construction of ciphertext from shortciphertext.

**distinguishingchance: report effectiveness of best attack known**

An encrypting BAT can provide a distinguishingchance function callable as follows:

```c
#include "sizes.h"

double e;
double s;
double p = distinguishingchance(e, s);
```

The distinguishingchance function returns a number between 0 and 1, namely the ciphertext-distinguishing (IND-CPA) probability for an attacker spending e euros and s seconds against one public key. Here e and s are powers of 2 between 2^0 and 2^40.

The attacker is given a ciphertext obtained either by encrypting m0 or by encrypting m1, where m0 and m1 are messages of the same length. The attacker's goal is to guess, with probability at least 50%+p, whether the decryption is m0 or m1. The attacker is not required to carry out a passive attack; the attacker is presumed to be able to specify m0 and m1. The attacker is not required to use meaningful messages m0 and m1; any distinguished messages, no matter how random they look, are presumed to be a disaster. These presumptions are standard: without them, every application would need a separate analysis of the message space.

There is a separate page with more information on security evaluations.

**multiplekeydistinguishingchance: report effectiveness of best attack known**

An encrypting BAT can provide a multiplekeydistinguishingchance function callable as follows:

```c
#include "sizes.h"

double e;
double s;
double k;
double p = multiplekeydistinguishingchance(e, s, k);
```

Figure 2.5: The call for encrypting BATs, version 2006.06.16, page 5.
The multiple-key distinguishing chance function returns a number between 0 and 1, namely the

ciphertext-distinguishing (IND-CPA) probability for an attacker spending $e$ euros and $s$ seconds against $k$ public
keys. Here $e$, $s$, and $k$ are powers of 2 between $2^{-0}$ and $2^{-40}$.

The result of multiple-key distinguishing chance can be larger than the result of
distinguishing chance by a factor as large as $k$.

c Caitacks: report extra effectiveness of chosen-ciphertext
attacks

An encrypting BAT can provide a c Caitacks function callable as follows:

```c
#include "sizes.h"
int x = cc Caitacks();
```

The cc Caitacks function returns 100 if adaptive chosen-ciphertext attacks (IND-CCA2) are more effective than
chosen-plaintext attacks. It returns 0 if adaptive chosen-ciphertext attacks are no more effective than
chosen-plaintext attacks.

timingattacks: report extra effectiveness of timing attacks

An encrypting BAT can provide a timingattacks function callable as follows:

```c
#include "sizes.h"
int x = timingattacks();
```

The timingattacks function returns 0 if the software does not leak any secret information through timing
(variable time for branching, variable time for memory access, etc.) i.e., if the best attack known that sees
timings is as difficult as the best attack known that does not see timings. It returns 100 if the software leaks
secret information through timing.

copyrightclaims: report copyright claims

An encrypting BAT can provide a copyrightclaims function callable as follows:

```c
#include "sizes.h"
int x = copyrightclaims();
```

The copyrightclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a copyright holder that the distribution of this software
  infringes the copyright. In particular, the author of the software is not making such claims and does not
  intend to make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but
  the author has no financial connections to the copyright.
- 30: The author has financial connections to a copyright restricting distribution of this software.

More numbers may be defined in the future.

No matter what the BAT’s copyright status is, eBATs will publicly distribute copies of the BAT for benchmarking.
The submitter must ensure before submission that publication is legal.

c patentclaims: report patent claims

An encrypting BAT can provide a patentclaims function callable as follows:

```c
#include "sizes.h"
int x = patentclaims();
```

The patentclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a patent holder that the use of this software infringes
  the patent. In particular, the author of the software is not making such claims and does not intend to
  make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but
  the author has no financial connections to the patent.
- 30: The author has financial connections to a patent restricting use of this software.
Call for public-key-encryption software for benchmarking  
http://www.ecrypt.eu.org/ebats/call-encryption.html

More numbers may be defined in the future.
No matter what the BAT's patent status is, eBATS will publicly distribute copies of the BAT for benchmarking.

**Version**

This is version 2006.06.16 of the call-encryption.html web page. This web page is in the public domain.
Chapter 3

The call for contributions of signing BATs

On 16 June 2006, VAMPIRE issued a call for submission of public-key-signature systems for benchmarking. See Figures 3.1 through 3.7.
Call for public-key-signature software for benchmarking

eBATS (ECRYPT Benchmarking of Asymmetric Systems) is a competition for the most efficient public-key systems. eBATS measures public-key-signature systems according to the following criteria:

- Time to generate a key pair.
- Length of the secret key.
- Length of the public key.
- Time to sign a message using the secret key.
- Length of the signed message.
- Time to verify the signed message using the public key.

"Time" refers to time on real computers: cycles on a Pentium III 68a, cycles on a PowerPC G5, cycles on an Athlon 64 X2, etc. eBATS times each system on a wide variety of computers, ensuring direct comparability of all systems on whichever computers are of interest to the users. Tools to graph the results will be made available.

This page explains how cryptographers can submit implementations of public-key-signature systems (RSA, DSA, ECDSA, Merkle hash trees, HFE signatures, etc.) to eBATS.

Introduction to the eBATS signing API

Formal submission requirements have been kept to a minimum. Your software has to be a signing BAT (Benchmarkable Asymmetric Tool), meaning that it supports the following three functions:

- keypair, returning a secret key and a public key;
- signedmessage, reading a message and a secret key, returning a signed message; and
- messagesigned, verifying a signed message and a public key, returning the message that was signed.

Don’t want to handle arbitrary-length messages? No problem. Instead of implementing signedmessage and messagesigned, you can implement shortmessage and shortmessagesigned, and define the maximum length of a short message. BATMAN, the eBATS benchmarking software, will automatically handle longer messages using a hash function.

Don’t want to provide recovery of the original message from a signature? No problem. Instead of implementing signedshortmessage and shortmessagesigned, you can implement signatureofshorthash and verification. BATMAN will automatically handle message recovery by signing a hash and appending the original message. (But systems built in this way aren’t likely to successfully compete for the shortest signed messages!)

You can also provide additional functions that document additional features of your system:

- forgerychallenge, documenting resistance to the best single-key attack known.
- multipledisjointmessage, documenting resistance to the best multiple-key attack known.
- timingattacks, documenting resistance to timing attacks.
- copyrightclaims, documenting copyright claims against distribution of the software.
- patentclaims, documenting patent claims against use of the software.

Claims regarding these additional features do not have the same level of verifiability as the eBATS measurements of key size, signing time, etc.; eBATS will nevertheless report these claims for public discussion.

The eBATS signing API is described below in more detail. There’s a separate page on BATMAN; you will be able to download and use BATMAN before submission to check that your implementation works properly. There’s also a separate page discussing security evaluations in more detail.

Files in a signing BAT

A signing BAT is a tar.gz file containing one directory. The directory contains a file sizes.h, any number of additional *.h, *.c, and *.cpp files implementing the eBATS signing API, and a file documentation.pdf with references and other comments for cryptographers.

The directory name is the BAT name followed by a dash and a version number: e.g., ronald-1 for a BAT named ronald, version 1. eBATS will rename BATs if there is a conflict in names.

The file sizes.h defines various macros discussed below: SECRETKEY_BYTES, PUBLICKEY_BYTES, SIGNATURE_BYTES, optionally SHORTHEASH_BYTES, optionally SHORTMESSAGE_BYTES.

BATMAN will automatically decide whether the BAT is a C BAT, providing the eBATS API functions in C, or a C++ BAT, providing the eBATS API functions in C++. Either way, the BAT can call C functions in its *.c files and assembly-language functions in its *.S files. BATs written in other languages have to be compiled to C++, C, or assembly language.
Parametrized BATs

Some BATs allow parameters. For example, a typical RSA implementation allows a wide range of key sizes. On the other hand, some RSA implementations gain speed by focusing on particular key sizes.

The eBATS API can support BATs of either type. A parametrized BAT includes, in the same directory as sizes.h, a parameters file with several lines; each line specifies compilation options that select a particular parameter choice. A parameter choice is specified by BAT-specific macros, which are used by sizes.h etc., and by a PARAMETERS macro (without white space), which is used to identify parameters in the eBATS results.

For example, version 1 of the RONALD BAT has a 29-line parameters file starting

-DMODULES BITS=768  -DPARAMETERS="768"
-DMODULES BITS=832  -DPARAMETERS="832"
-DMODULES BITS=896  -DPARAMETERS="896"
-DMODULES BITS=960  -DPARAMETERS="960"
-DMODULES BITS=1024 -DPARAMETERS="1024"

and continuing (in roughly geometric progression) until

-DMODULES BITS=4096 -DPARAMETERS="4096"

The MODULES BITS macro controls PUBLICKEY_BYTES etc. through the lines

#define MODULES BYTES (MODULES BITS / 8)
#define PUBLICKEY_BYTES (MODULES BYTES)

in the sizes.h file. The PARAMETERS macro is printed in the eBATS measurements.

The parameters file can omit -DPARAMETERS=... if sizes.h defines PARAMETERS. For example, version 2 of the RONALD BAT has a 29-line parameters file starting

-DMODULES BITS=768
-DMODULES BITS=832
-DMODULES BITS=896
-DMODULES BITS=960
-DMODULES BITS=1024

and the following lines in sizes.h:

#define XSTRINGIFY(N) #N
#define STRINGIFY(N) XSTRINGIFY(N)
#define PARAMETERS (STRINGIFY(MODULES_BITS))

Tuned BATs

A BAT can contain several implementations of the same functions: e.g., a P4-tuned implementation, a G5-tuned implementation, etc. A tuned BAT includes, in the same directory as sizes.h, a tunings file with several lines, each line specifies compilation options that select a particular tuning. A tuning is specified by BAT-specific macros, which are used by sizes.h etc., and by a TUNING macro (without white space), which is used to identify tuning in the eBATS results.

BATMAN will automatically try each tuning and select the tuning where signedmessage runs most quickly. A BAT can define a TUNETARGET macro in sizes.h; in that case BATMAN will select the tuning where TUNETARGET() runs most quickly.

Any particular tuning is allowed to be unportable, failing to compile on most platforms. BATMAN will skip tunings that don't compile or that flunk some simple tests.

Generating random numbers

BATMAN sets up file descriptor 0 reading from a neverending source of hard-to-predict secret random bytes. BATs are free to assume this; the keypair function, for example, can obtain secret bytes using getchar().

Functions are permitted, but not encouraged, to generate randomness in other ways, such as by opening /dev/urandom. These functions won't be benchmarkable on systems that don't have /dev/urandom, and they won't be suitable for black-box regression testing.

Using hash functions

BATMAN provides a cryptographic hash function hash256 callable from a BAT as follows:

const unsigned char m[...] = unsigned long mlen;

Figure 3.2: The call for signing BATs, version 2006.06.16, page 2.
unsigned char h[32];
hash256(h,m,mlen);
hash256 hashes bytes m[0],m[1],... m[mlen-1] and puts the output into h[0],h[1],... h[31]. Currently hash256 is implemented as SHA-256.

To simplify comparisons of public-key systems, eBATS recommends that BATs use hash256 for all necessary hashing. This is not a recommendation of SHA-256 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different hash functions.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks that work with any hash function. Any security problems in SHA-256 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

Using stream ciphers

BATMAN provides an additive stream cipher stream256 callable from a BAT as follows:

    const unsigned char m[...]; unsigned long long mlen;
    const unsigned char c[...];
    const unsigned char k[32];
    const unsigned char n[8];
    stream256(c,m,mlen,k,n);

stream256 encrypts (decrypts) bytes m[0],m[1],... m[mlen-1] and puts the output into c[0],c[1],... c[mlen-1]. It uses a 32-byte key k[0],k[1],... k[31] and an 8-byte nonce n[0],n[1],... n[7]. Currently stream256 is implemented as Salsa20.

To simplify comparisons of public-key systems, eBATS recommends that BATs use stream256 for all necessary stream generation. This is not a recommendation of Salsa20 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different ciphers.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks that work with any stream cipher. Any security problems in Salsa20 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

keypair: generate a new secret key and public key

A signing BAT must provide a keypair function callable as follows:

    #include "sizes.h"
    unsigned char sk[SECRETKEY_BYTES]; unsigned long long sklen;
    unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
    keypair(sk,&sklen,pk,&pklen);

The keypair function generates a new secret key and a new public key. It puts the number of bytes of the secret key into sklen; puts the number of bytes of the public key into pklen; puts the secret key into sk[0], sk[1],... sk[sklen-1]; and puts the public key into pk[0], pk[1],... pk[pklen-1]. It then returns 0.

keypair guarantees that sklen is at most SECRETKEY_BYTES, and that pklen is at most PUBLICKEY_BYTES, so that the caller can allocate enough space.

If key generation is impossible for some reason (e.g., not enough memory), keypair returns a negative number, possibly after modifying sk[0], sk[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

signedmessage: sign a message using a secret key

A signing BAT can provide a signedmessage function callable as follows:

    #include "sizes.h"
    const unsigned char sk[SECRETKEY_BYTES]; unsigned long long sklen;
    const unsigned char m[...]; unsigned long long mlen;
    unsigned char sm[...]; unsigned long long smlen;
    signedmessage(sm,&smlen,m,mlen,sk,sklen);

The signedmessage function uses a secret key sk[0], sk[1],... sk[sklen-1] to sign a message m[0], m[1],... m[mlen-1]. It puts the length of the signed message into smlen and puts the signed message into sm[0], sm[1],... sm[smlen-1]. It then returns 0.
The signedmessage function guarantees that smlen is at most mlen+SIGNATURE_BYTES. The SIGNATURE_BYTES macro is defined in sizes.h.

The signedmessage function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the keypair function.

If signing is impossible for some reason, signedmessage returns a negative number, possibly after modifying sm[0], sm[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

Implementors of the signedmessage function are warned that they should not go to extra effort to compress the message m. Higher-level applications should be presumed to compress messages before calling the signedmessage function; in particular, BATMAN uses random messages to make compression ineffective. On the other hand, the signed message sm is longer than the original message m and might be compressible; any reduction of the signature overhead will be visible in the eBATS measurements.

**messagesigned: verify a message using a public key**

A signing BAT can provide a messagesigned callable interface as follows:

```c
#include "sizes.h"
const unsigned char pk[PUBLICKEY_BYTES];
const unsigned long pklen;
const unsigned char sm[...];
const unsigned long smlen;
unsigned long mlen;
messagesigned(m, &smlen, sm, smlen, pk, pklen);
```

The messagesigned function uses a public key pk[0], pk[1], ..., pk[pklen-1] to verify an allegedly signed message sm[0], sm[1], ..., sm[smlen-1]. If the message has a valid signature, the messagesigned function puts the length of the original message (without the signature) into mlen, puts the original message into m[0], m[1], ..., m[mlen-1], and returns 0.

The messagesigned function guarantees that mlen is at most smlen.

The messagesigned function is free to assume that the public key pk[0], pk[1], ..., pk[pklen-1] was generated by a successful call to the publickey function. The messagesigned function is not permitted to assume that sm[0], sm[1], ..., sm[smlen-1] was generated by a call to the signedmessage function; the messagesigned function is responsible for detecting and eliminating forgeries.

If signature verification is impossible for some reason, messagesigned returns a negative number, possibly after modifying m[0], m[1], etc. Current implementations should return -100 for invalid signatures, and -1 for all other problems; other return values with special meanings may be defined in the future.

**signedshortmessage: sign a message using a secret key**

A signing BAT can provide a signedshortmessage callable interface as follows:

```c
#include "sizes.h"
const unsigned char sk[SECRETKEY_BYTES];
const unsigned long sklen;
const unsigned char m[SHORTMESSAGE_BYTES];
const unsigned long mlen;
unsigned char sm[SIGNATURE_BYTES];
signedshortmessage(sm, &smlen, m, mlen, sk, sklen);
```

The signedshortmessage function uses a secret key sk[0], sk[1], ..., sk[sklen-1] to sign a message m[0], m[1], ..., m[mlen-1]. It puts the length of the signed message into smlen and puts the signed message into sm[0], sm[1], ..., sm[smlen-1]. It then returns 0.

The signedshortmessage function guarantees that smlen is at most SHORTMESSAGE_BYTES. The signedshortmessage function guarantees that smlen is exactly SIGNATURE_BYTES. The SHORTMESSAGE_BYTES and SIGNATURE_BYTES macros are defined in sizes.h.

The signedshortmessage function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the keypair function.

If signing is impossible for some reason, signedshortmessage returns a negative number, possibly after modifying sm[0], sm[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

Implementors of the signedshortmessage function are warned that they should not go to extra effort to compress the message m. Higher-level applications should be presumed to compress messages before calling the signedshortmessage function; in particular, BATMAN uses random messages to make compression ineffective. On the other hand, the signed message sm is longer than the original message m and might be

Figure 3.4: The call for signing BATs, version 2006.06.16, page 4.
compressible; any reduction of the signature overhead will be visible in the eBATS measurements.

**BATMAN automatically builds signedmessage on top of signedshortmessage as follows.** Messages with at most SHORTMESSAGE_BYTES-1 bytes are simply signed with signedshortmessage. Messages with SHORTMESSAGE_BYTES or more bytes are hashed with SHA-256; the 32-byte hash and the first SHORTMESSAGE_BYTES-32 bytes of the message are signed with signedshortmessage; the rest of the message is appended. SHORTMESSAGE_BYTES must be at least 32.

**shortmessagesigned: verify a message using a public key**

A signing BAT can provide a shortmessagesigned function callable as follows:

```c
#include "sizes.h"
const unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
const unsigned char sm[SIGNATURE_BYTES]; unsigned long long smlen;
unsigned char m[SHORTMESSAGE_BYTES]; unsigned long long mlen;
shortmessagesigned(m,smlen,sm,pk,pklen);
```

The shortmessagesigned function uses a public key pk[0], pk[1], ..., pk[pklen-1] to verify an allegedly signed message sm[0], sm[1], ..., sm[smlen-1]. If the message has a valid signature, the shortmessagesigned function puts the length of the original message (without the signature) into mlen, puts the original message into m[0], m[1], ..., m[mlen-1], and returns 0.

The shortmessagesigned function is free to assume that smlen is exactly SHORTMESSAGE_BYTES. The shortmessagesigned function guarantees that mlen is at most SHORTMESSAGE_BYTES.

The shortmessagesigned function is free to assume that the public key pk[0], pk[1], ..., pk[pklen-1] was generated by a successful call to the publickey function. The shortmessagesigned function is not permitted to assume that sm[0], sm[1], ..., sm[smlen-1] was generated by a call to the signedshortmessage function; the shortmessagesigned function is responsible for detecting and eliminating forgeries.

If signature verification is impossible for some reason, shortmessagesigned returns a negative number, possibly after modifying m[0], m[1], etc. Current implementations should return -100 for invalid signatures, and -1 for all other problems; other return values with special meanings may be defined in the future.

**signatureofshorthash: sign a message using a secret key**

A signing BAT can provide a signatureofshorthash function callable as follows:

```c
#include "sizes.h"
const unsigned char sk[SECRETKEY_BYTES]; unsigned long long sklen;
const unsigned char m[SHORTMESSAGE_BYTES]; unsigned long long mlen;
unsigned char sm[SIGNATURE_BYTES]; unsigned long long smlen;
signatureofshorthash(sm,smlen,m,mlen,sk,sklen);
```

The signatureofshorthash function uses a secret key sk[0], sk[1], ..., sk[sklen-1] to sign a message m[0], m[1], ..., m[mlen-1]. It puts the length of the signature into smlen and puts the signature into sm[0], sm[1], ..., sm[smlen-1]. It then returns 0.

The signatureofshorthash function is free to assume that mlen is at most SHORTHASH_BYTES. The signatureofshorthash function guarantees that smlen is exactly SHORTHASH_BYTES. The SHORTHASH_BYTES and SIGNATURE_BYTES macros are defined in sizes.h.

The signatureofshorthash function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the keypair function.

If signing is impossible for some reason, signatureofshorthash returns a negative number, possibly after modifying sm[0], sm[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

**verification: verify a message using a public key**

**Figure 3.5: The call for signing BATs, version 2006.06.16, page 5.**
A signing BAT can provide a verification function callable as follows:

```c
#include "sizes.h"
const unsigned char pk[PUBLICKEY_BYTES];
const unsigned long long pklen;
const unsigned char sm[Signature_BYTES];
const unsigned long long smlen;
verification(m, smlen, sm, mlen, pk, pklen);
```

The verification function uses a public key `pk[0], pk[1], ..., pk[pklen-1]` to verify an alleged signature `sm[0], sm[1], ..., sm[smlen-1]` on a message `m[0], m[1], ..., m[mlen-1]`. If the message has a valid signature, the verification function returns 0.

The verification function is free to assume that `smlen` is exactly `Signature_BYTES` and that `mlen` is at most `SHORTHASH_BYTES`.

The verification function is free to assume that the public key `pk[0], pk[1], ..., pk[pklen-1]` was generated by a successful call to the publickey function. The verification function is not permitted to assume that `sm[0], sm[1], ..., sm[smlen-1]` was generated by a call to the signatureofshorthash function; the verification function is responsible for detecting and eliminating forgeries.

If signature verification is impossible for some reason, verification returns a negative number, possibly after modifying `m[0], m[1], etc. Current implementations should return -100 for invalid signatures, and -1 for all other problems; other return values with special meanings may be defined in the future.

**forgerychance: report effectiveness of best attack known**

A signing BAT can provide a forgerychance function callable as follows:

```c
#include "sizes.h"
double e;
double s;
double p = forgerychance(e, s);
```

The forgerychance function returns a number between 0 and 1, namely the probability that an attacker spending `e` euros will succeed at forging at least one signed message within `s` seconds, given a public key. Here `e` and `s` are powers of 2 between $2^{-0}$ and $2^{-40}$.

The attacker is not required to carry out a selective forgery, i.e., a forgery on a message chosen in advance by the attacker. Any forged message, no matter how random it looks, is presumed to be a disaster if it was not signed by the legitimate key owner. This is a standard presumption: without it, every application would need a separate analysis of potential forgeries within the application's message space.

The attacker is not required to carry out a blind attack. The attacker is presumed to be able to see many legitimate signatures. This is a standard presumption: most signature applications do not keep signatures secret.

The attacker is not required to carry out a passive attack. The attacker is presumed to be able to influence the legitimately signed messages. This is a standard presumption: without it, every application would need a separate analysis of the attacker's influence.

There is a separate page with more information on security evaluations.

**multiplekeyforgerychance: report effectiveness of best attack known**

A signing BAT can provide a multiplekeyforgerychance function callable as follows:

```c
#include "sizes.h"
double e;
double s;
double k;
double p = multiplekeyforgerychance(e, s, k);
```

The multiplekeyforgerychance function returns a number between 0 and 1, namely the probability that an attacker spending `e` euros will succeed at forging at least one signed message within `s` seconds, given `k` public...
keys. Here e, s, and k are powers of 2 between $2^0$ and $2^{40}$.
The result of multiple-key forgery chance can be larger than the result of forgery chance by a factor as large as k.

**timingattacks: report extra effectiveness of timing attacks**

A signing BAT can provide a timingattacks function callable as follows:

```c
#include "sizes.h"
int x = timingattacks();
```

The timingattacks function returns 0 if the software does not leak any secret information through timing (variable time for branching, variable time for memory access, etc.). i.e., if the best attack known that sees timings is as difficult as the best attack known that does not see timings. It returns 100 if the software leaks secret information through timing.

**copyrightclaims: report copyright claims**

A signing BAT can provide a copyrightclaims function callable as follows:

```c
#include "sizes.h"
int x = copyrightclaims();
```

The copyrightclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a copyright holder that the distribution of this software infringes the copyright. In particular, the author of the software is not making such claims and does not intend to make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but the author has no financial connections to the copyright.
- 30: The author has financial connections to a copyright restricting distribution of this software.

More numbers may be defined in the future.

No matter what the BAT’s copyright status is, eBATS will publicly distribute copies of the BAT for benchmarking. The submitter must ensure before submission that publication is legal.

**patentclaims: report patent claims**

A signing BAT can provide a patentclaims function callable as follows:

```c
#include "sizes.h"
int x = patentclaims();
```

The patentclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a patent holder that the use of this software infringes the patent. In particular, the author of the software is not making such claims and does not intend to make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but the author has no financial connections to the patent.
- 30: The author has financial connections to a patent restricting use of this software.

More numbers may be defined in the future.

No matter what the BAT’s patent status is, eBATS will publicly distribute copies of the BAT for benchmarking.

**Version**

This is version 2006.06.16 of the call-signatures.html web page. This web page is in the public domain.
Chapter 4

The call for contributions of secret-sharing BATs

On 16 June 2006, VAMPIRE issued a call for submission of public-key-secret-sharing systems for benchmarking. See Figures 4.1 through 4.5.
Call for public-key-secret-sharing software for benchmarking

eBATS (ECRYPT Benchmarking of Asymmetric Systems) is a competition for the most efficient public-key systems. eBATS measures public-key-secret-sharing systems according to the following criteria:

- Time to generate a key pair.
- Length of the secret key.
- Length of the public key.
- Time to generate a shared secret from the secret key and another user’s public key.
- Length of the shared secret.

"Time" refers to time on real computers: cycles on a Pentium III 68a, cycles on a PowerPC G5, cycles on an Athlon 64 X2, etc. eBATS times each system on a wide variety of computers, ensuring direct comparability of all systems on whichever computers are of interest to the users. Tools to graph the results will be made available.

This page explains how cryptographers can submit implementations of public-key-secret-sharing systems (DH, LUC, XTR, ECDH, etc.) to eBATS.

Introduction to the eBATS secret-sharing API

Formal submission requirements have been kept to a minimum. Your software has to be a secret-sharing BAT (Benchmarkable Asymmetric Tool), meaning that it supports the following two functions:

- keypair, returning a secret key and a public key; and
- sharedsecret, reading a secret key and another public key, returning a shared secret.

You can also provide additional functions that document additional features of your system:

- cdhchange, documenting resistance to the best two-key attack known.
- multiplekeycdhchange, documenting resistance to the best multiple-key attack known.
- fakekeyattacks, documenting resistance to fake-key attacks.
- timingattacks, documenting resistance to timing attacks.
- copyrightclaims, documenting copyright claims against distribution of the software.
- patentclaims, documenting patent claims against use of the software.

Claims regarding these additional features do not have the same level of verifiability as the eBATS measurements of key size, secret-sharing time, etc.; eBATS will nevertheless report these claims for public discussion.

The eBATS secret-sharing API is described below in more detail. There’s a separate page on BATMAN, the eBATS benchmarking software; you will be able to download and use BATMAN before submission to check that your implementation works properly. There’s also a separate page discussing security evaluations in more detail.

Files in a secret-sharing BAT

A secret-sharing BAT is a tar.gz file containing one directory. The directory contains a file sizes.h, any number of additional *.S, *.c, and *.cpp files implementing the eBATS secret-sharing API, and a file documentation.pdf with references and other comments for cryptographers.

The directory name is the BAT name followed by a dash and a version number: e.g., ronald-1 for a BAT named ronald, version 1. eBATS will rename BATs if there is a conflict in names.

The file sizes.h defines various macros discussed below: SECRETKEY _BYTES, PUBLICKEY _BYTES, and SHAREDSECRET _BYTES.

BATMAN will automatically decide whether the BAT is a C BAT, providing the eBATS API functions in C, or a C++ BAT, providing the eBATS API functions in C++. Either way, the BAT can call C functions in its *.c files and assembly-language functions in its *.S files. BATs written in other languages have to be compiled to C++, C, or assembly language.

Parametrized BATs

Some BATs allow parameters. For example, a typical DH implementation allows a wide range of key sizes. On the other hand, some DH implementations gain speed by focusing on particular key sizes.

The eBATS API can support BATs of either type. A parametrized BAT includes, in the same directory as sizes.h, a parameters file with several lines; each line specifies compilation options that select a particular parameter choice. A parameter choice is specified by BAT-specific macros, which are used by sizes.h etc., and by a PARAMETERS macro (without white space), which is used to identify parameters in the eBATS results.
Call for public-key-secret-sharing software for benchmarking: \url{http://www.ecrypt.eu.org/ebats/call-secretsharing.html}

For example, version 1 of the RONALD BAT has a 29-line parameters file starting

```
-DMODULUS_BITS=768  -DPARAMETERS="768"
-DMODULUS_BITS=832  -DPARAMETERS="832"
-DMODULUS_BITS=896  -DPARAMETERS="896"
-DMODULUS_BITS=960  -DPARAMETERS="960"
-DMODULUS_BITS=1024 -DPARAMETERS="1024"
```

and continuing (in roughly geometric progression) until

```
-DMODULUS_BITS=4096  -DPARAMETERS="4096"
```

The MODULUS_BITS macro controls PUBLICKEY_BYTES etc. through the lines

```c
#define MODULUSBYTES (MODULUS_BITS / 8)
#define PUBLICKEYBYTES (MODULUSBYTES)
```

in the sizes.h file. The PARAMETERS macro is printed in the eBATS measurements.

The parameters file can omit -DPARAMETERS=... if sizes.h defines PARAMETERS. For example, version 2 of the RONALD BAT has a 29-line parameters file starting

```
-DMODULUS_BITS=768
-DMODULUS_BITS=832
-DMODULUS_BITS=896
-DMODULUS_BITS=960
-DMODULUS_BITS=1024
```

and the following lines in sizes.h:

```c
#define XSTRINGIFY(N) #N
#define STRINGIFY(N) XSTRINGIFY(N)
#define PARAMETERS (STRINGIFY(MODULUS_BITS))
```

### Tuned BATs

A BAT can contain several implementations of the same functions: e.g., a P4-tuned implementation, a G5-tuned implementation, etc. A tuned BAT includes, in the same directory as sizes.h, a tunings file with several lines; each line specifies compilation options that select a particular tuning. A tuning is specified by BAT-specific macros, which are used by sizes.h etc., and by a TUNING macro (without white space), which is used to identify tuning in the eBATS results.

BATMAN will automatically try each tuning and select the tuning where sharedsecret runs most quickly. A BAT can define a TUNETARGET macro in sizes.h; in that case BATMAN will select the tuning where TUNETARGET() runs most quickly.

Any particular tuning is allowed to be unportable, failing to compile on most platforms. BATMAN will skip tunings that don't compile or that flunk some simple tests.

### Generating random numbers

BATMAN sets up file descriptor 0 reading from a neverending source of hard-to-predict secret random bytes. BATs are free to assume this; the keypair function, for example, can obtain secret bytes using getc().

Functions are permitted, but not encouraged, to generate randomness in other ways, such as by opening /dev/urandom. These functions won't be benchmarkable on systems that don't have /dev/urandom, and they won't be suitable for black-box regression testing.

### Using hash functions

BATMAN provides a cryptographic hash function hash256 callable from a BAT as follows:

```c
const unsigned char m[32]; unsigned long long mlen;
hash256(h,m,mlen);
```

hash256 hashes bytes m[0], m[1], ..., m[mlen-1] and puts the output into h[0], h[1], ..., h[31]. Currently hash256 is implemented as SHA-256.

To simplify comparisons of public-key systems, eBATS recommends that BATs use hash256 for all necessary hashing. This is not a recommendation of SHA-256 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different hash functions.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks

Figure 4.2: The call for secret-sharing BATs, version 2006.06.16, page 2.
Call for public-key-secret-sharing software for benchmarking
http://www.ecrypt.eu.org/ebats/call-secretsharing.html

that work with any hash function. Any security problems in SHA-256 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

Using stream ciphers

BATMAN provides an additive stream cipher stream256 callable from a BAT as follows:

    const unsigned char m[...]; unsigned long long mlen;
    const unsigned char c[...];
    const unsigned char k[32];
    stream256(c, m, mlen, k, n);

stream256 encrypts (or decrypts) bytes m[0], m[1], ..., m[mlen-1] and puts the output into c[0], c[1], ..., c[mlen-1]. It uses a 32-byte key k[0], k[1], ..., k[31] and an 8-byte nonce n[0], n[1], ..., n[7]. Currently stream256 is implemented as Salsa20.

To simplify comparisons of public-key systems, eBATS recommends that BATs use stream256 for all necessary stream generation. This is not a recommendation of Salsa20 for any purpose other than public-key benchmarking. Public-key systems may be able to gain speed and security by choosing different ciphers.

To the extent that eBATS considers security of public-key systems, it focuses on generic attacks, i.e., attacks that work with any stream cipher. Any security problems in Salsa20 are outside the scope of eBATS, although obviously they should be discussed elsewhere.

keypair: generate a new secret key and public key

A secret-sharing BAT must provide a keypair function callable as follows:

    #include "sizes.h"
    unsigned char sk[SECRETKEY_BYTES]; unsigned long long sklen;
    unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
    keypair(sk, &sklen, pk, &pklen);

The keypair function generates a new secret key and a new public key. It puts the number of bytes of the secret key into sklen, puts the number of bytes of the public key into pklen, puts the secret key into sk[0], sk[1], ..., sk[sklen-1]; and puts the public key into pk[0], pk[1], ..., pk[pklen-1]. It then returns 0.

keypair guarantees that sklen is at most SECRETKEY_BYTES, and that pklen is at most PUBLICKEY_BYTES, so that the caller can allocate enough space.

If key generation is impossible for some reason (e.g., not enough memory), keypair returns a negative number, possibly after modifying sk[0], sk[1], etc. Current implementations should return -1; other return values with special meanings may be defined in the future.

sharedsecret: generate a shared secret using a secret key and another user's public key

A secret-sharing BAT must provide a sharedsecret function callable as follows:

    #include "sizes.h"
    const unsigned char sk[PUBLICKEY_BYTES]; unsigned long long sklen;
    const unsigned char pk[PUBLICKEY_BYTES]; unsigned long long pklen;
    unsigned char s[SHAREDSECRET_BYTES]; unsigned long long slen;
    sharedsecret(s, &slen, sk, sklen, pk, pklen);

The sharedsecret function uses a secret key sk[0], sk[1], ..., sk[sklen-1] and another user's public key pk[0], pk[1], ..., pk[pklen-1] to compute a shared secret. It puts the length of the shared secret into slen and puts the shared secret into s[0], s[1], ..., s[slen-1]. It then returns 0.

The sharedsecret function guarantees that slen is at most SHAREDSECRET_BYTES. The SHAREDSECRET_BYTES macro is defined in sizes.h.

The sharedsecret function is free to assume that the secret key sk[0], sk[1], ..., sk[sklen-1] was generated by a successful call to the keypair function.

If shared-secret generation is impossible for some reason, sharedsecret returns a negative number, possibly after modifying s[0], s[1], etc. Current implementations should return -1; other return values with special

Figure 4.3: The call for secret-sharing BATs, version 2006.06.16, page 3.
meanings may be defined in the future.

**cdhchance: report effectiveness of best attack known**

A secret-sharing BAT can provide a cdhchance function callable as follows:

```c
#include "sizes.h"

double e;
double s;
double p = cdhchance(e,s);
```

The cdhchance function returns a number between 0 and 1, namely the probability that an attacker spending e euros and s seconds can deduce at least one shared secret given two public keys. Here e and s are powers of 2 between $2^0$ and $2^{40}$.

The cdhchance function is free to ignore attacks that merely distinguish the shared secret from uniform (DDH) without computing the shared secret (CDH); shared secrets are presumed to be hashed before they are used.

There is a separate page with more information on security evaluations.

**multiplekeycdhchance: report effectiveness of best attack known**

A secret-sharing BAT can provide a multiplekeycdhchance function callable as follows:

```c
#include "sizes.h"

double e;
double s;
double k;
double p = multiplekeycdhchance(e,s,k);
```

The multiplekeycdhchance function returns a number between 0 and 1, namely the probability that an attacker spending e euros and s seconds can deduce at least one shared secret given k public keys. (More precisely, there are public keys $key_1, key_2, ..., key_k$; the attack is successful if it prints a vector $\{i, j, z\}$ where $1 \leq i < j \leq k$ and $z$ is the secret shared between $key_i$ and $key_j$.) Here e, s, and k are powers of 2 between $2^0$ and $2^{40}$.

The result of multiplekeycdhchance can be larger than the result of cdhchance by a factor as large as $k(k-1)/2$.

**fakekeyattacks: report extra effectiveness of fake-key attacks**

A secret-sharing BAT can provide an fakekeyattacks function callable as follows:

```c
#include "sizes.h"

int x = fakekeyattacks();
```

The fakekeyattacks function returns 100 if an active attacker can save time by providing fake keys (in applications that do not go to any extra effort to validate keys). It returns 0 if an active attacker obtains no benefit from fake keys (for example, if the sharedsecret function includes all necessary key validation).

**timingattacks: report extra effectiveness of timing attacks**

A secret-sharing BAT can provide a timingattacks function callable as follows:

```c
#include "sizes.h"

int x = timingattacks();
```

The timingattacks function returns 0 if the software does not leak any secret information through timing (variable time for branching, variable time for memory access, etc.). i.e., if the best attack known that sees timings is as difficult as the best attack known that does not see timings. It returns 100 if the software leaks secret information through timing.

**copyrightclaims: report copyright claims**

A secret-sharing BAT can provide a copyrightclaims function callable as follows:

```c
#include "sizes.h"
```

Figure 4.4: The call for secret-sharing BATs, version 2006.06.16, page 4.
int x = copyrightclaims();

The copyrightclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a copyright holder that the distribution of this software infringes the copyright. In particular, the author of the software is not making such claims and does not intend to make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but the author has no financial connections to the copyright.
- 30: The author has financial connections to a copyright restricting distribution of this software.

More numbers may be defined in the future.

No matter what the BAT's copyright status is, eBATS will publicly distribute copies of the BAT for benchmarking. The submitter must ensure before submission that publication is legal.

**patentclaims: report patent claims**

A secret-sharing BAT can provide a patentclaims function callable as follows:

```c
#include "sizes.h"

int x = patentclaims();
```

The patentclaims function returns one of the following numbers:

- 0: There are no known present or future claims by a patent holder that the use of this software infringes the patent. In particular, the author of the software is not making such claims and does not intend to make such claims.
- 10: The author is aware of third parties making such claims, but the author disputes those claims.
- 20: The author is aware of third parties making such claims, and the author agrees with the claims, but the author has no financial connections to the patent.
- 30: The author has financial connections to a patent restricting use of this software.

More numbers may be defined in the future.

No matter what the BAT's patent status is, eBATS will publicly distribute copies of the BAT for benchmarking.

**Version**

This is version 2006.06.16 of the call-secretsharing.html web page. This web page is in the public domain.