TTS: High-Speed Signatures on a Low-Cost Smart Card

Bo-Yin Yang[1], Jiun-Ming Chen[2], and Yen-Hung Chen[3]

[1] Mathematics Department, Tamkang University, Tamsui, Taiwan, by@moscito.org
[2] Chinese Data Security Inc. & National Taiwan University, jmchen@math.ntu.edu.tw
[3] Comp. Sci. & Info. Eng., Nat'l Taiwan U., Taipei, Taiwan, r92014@csie.ntu.edu.tw

Abstract. TTS is a genre of multivariate digital signature schemes first proposed in 2002. Its public map is composed of two affine maps sandwiching a *Tame Map*, which is a map invertible through serial substitution and solving linear equations. We implement the signing and key generation operations for a TTS instance with 20-byte hashes and 28-byte signatures, on popular extant microcontroller cores compatible to the Intel 8051. Our tests demonstrates that TTS can be even faster than SFLASH^{v2}, which is known for its celerity. The sample scheme $\mathrm{TTS}(20, 28)$ is fast enough for practical deployment on a low-end 8051-based embedded device. A really low-end part like a stock Intel 8051AH running at 3.57 MHz can sign in just 170ms. A better 8051-compatible chip will take a lot less time.
Security requirements today demand on-card key generation, and the big public keys of a multivariate PKC create a storage problem. TTS is unusual in that public keys can be synthesized on-card at a decent pace for block-by-block output, using some minimal information kept on-card. Since this does not take much more time than the I/O needed to transmit the public key to a reader, we can avoid holding the entire public key in the limited memory of a smart card. We show that this to be a gain for multivariate PKC's with relatively few terms per central equation. The literature is not rich in this kind of detailed description of an implementation of a signature scheme — capable of fast on-card public key generation, on a low-cost smart card without a co-processor, and at NESSIE-approved security levels. We look into other theory issues like safeguarding against side-channel attacks, and using unusual techniques for linear algebra under serious space restrictions, which may help implementations of other multivariate PKC's such as SFLASH.

Keywords: Multivariate public-key cryptosystem, finite field, smart card, 8051.

1 Introduction

For most adopters of Public-Key Infrastructure, the quarter-century-old RSA still remains the public-key cryptosystem of choice. We see that all is not perfect:

> RSA is too slow to be used on a smart card and this keeps the security achieved by smart card solutions insufficient: unable to implement a real public key signature. . . . N. Courtois *et al* ([1], 2003).

This must be taken in context as a historical perspective: Acceptable signing speed on smart cards with reasonably long RSA keys has become feasible around the turn of the millennium, especially with special-purpose hardware co-processors.

However, cost of deployment is still an obstacle, and there is clearly room for improvement: Chips must get even faster and cheaper, or the algorithms need revamping.

Traditionally, Public-Key Infrastructure (PKI) implementers stick to well-established Public-Key Cryptosystems (PKC) based on RSA, occasionally ECC. However these are comparatively slow, so cryptologists sought faster alternatives, among which are the "multivariate PKC", cryptosystems that use substitution into quadratic polynomials as a public map. The currently best known scheme of this type is SFLASH^{v2} ([15]), a derivative ([14]) of the prototype C^* ([9], broken in 1995 by [12]). Multivariate cryptosystems (more literature on the extended family: [3, 5, 7, 8]) are usually conceded to be faster than RSA and other traditional alternatives with a large algebraic structure. Unfortunately, the only such scheme to get a mention in the recently announced NESSIE final recommendations ([11]) is SFLASH^{v2}, and only grudgingly:

> . . . not recommended for general use but this signature scheme is very efficient on low cost smart cards, where the size of the public key is not a constraint.

Granted NESSIE was more concerned about SFLASH's untried security than size of its public keys, but still this rather sells SFLASH^{v2} and other multivariate PKC's short.

We aim to provide another example of how a multivariate PKC can provide superior performance for less. Our sample scheme, unlike the C^{*-}-derived SFLASH^{v2}, is a digital signature scheme from the different family of TTS ([2, 17]). Techniques shown here are applicable to other multivariate PKC's, however. Our test results on 8051-compatible microcontrollers are tabulated along with data published by the NESSIE project ([11]) and other recent sources (e.g. [1]). It showcases TTS well:

<table>
<tr><th>Scheme</th><th>Platform (T number)</th><th>Clock</th><th>Pr. Key</th><th>Code</th><th>RAM</th><th>Signature</th></tr>
<tr><td rowspan="3">TTS (20, 28)</td><td>Intel 8032AH (12)</td><td rowspan="5">3.57 MHz</td><td rowspan="3">1.4 kB</td><td>1.4 kB</td><td rowspan="3">128 B</td><td>144 ms</td></tr>
<tr><td>Intel 8051AH (12)</td><td rowspan="2">1.6 kB</td><td>170 ms</td></tr>
<tr><td>Winbond W77E58 (4)</td><td>64 ms</td></tr>
<tr><td>ESIGN</td><td rowspan="2">Intel 8051AH (12)</td><td>336 B</td><td>3.0 kB</td><td>800 B</td><td>12.0 s</td></tr>
<tr><td rowspan="2">SFLASH^{v2}</td><td rowspan="2">2.4 kB</td><td rowspan="2">3.3 kB</td><td rowspan="2">344 B</td><td>1.07 s</td></tr>
<tr><td rowspan="2">Infineon SLE66 (2)</td><td rowspan="2">10 MHz</td><td>59 ms</td></tr>
<tr><td rowspan="3">RSA-PSS (1024 bits)</td><td rowspan="3">320 B</td><td rowspan="5">N/A</td><td rowspan="5">$\geq$ 1kB</td><td>many s</td></tr>
<tr><td>NEC μPD789828*(12)</td><td>40 MHz</td><td>100 ms</td></tr>
<tr><td rowspan="3">Infineon SLE66*(2)</td><td rowspan="2">5 MHz</td><td>230 ms</td></tr>
<tr><td>RSA-2048</td><td>640 B</td><td>1.1 s</td></tr>
<tr><td>ECDSA-191</td><td>10 MHz</td><td>24 B</td><td>180 ms</td></tr>
<tr><td>NTRU-Sign</td><td>Philips 8051 (6)</td><td>16 MHz</td><td>100 B</td><td>5 kB</td><td>N/A</td><td>160 ms</td></tr>
</table>

Table 1. 8051-compatible results for various digital signature schemes (* = with co-processor)

Special Note: 8051-compatible parts running at the same nominal clock frequency have widely divergent speeds on the same program. On every 8051-compatible part, a majority of common instructions execute in the same amount of time, which is called the *instruction cycle* for this part. Every instruction executes in an integral number of instruction cycles. The ratio of one instruction cycle to one clock cycle is called the "T number". So a Siemens-Infineon SLE66 is up to six and usually 4-5 times faster than a 8051 at the same clock rate. Some platforms also have extras goodies including cryptographic co-processors, so some care is needed in interpreting the tabulated results.

Sometimes $\mathrm{TTS}(20, 28)$ need to hash just once while ESIGN or SFLASH^{v2}, repeated hashes must be taken. Without worrying about details, all told $\mathrm{TTS}(20, 28)$ is 6 times faster on the same resources than SFLASH^{v2}, which in turn was noted for being faster without a coprocessor than the traditional alternatives with a coprocessor.

This paper details an implementation of $\mathrm{TTS}(20, 28)$ on an 8051 compatible microcontroller. Sec. 2 describes the mathematics. Sec. 3 summarizes the 8051 platform (with a more complete review provided in Appendix A). Sec. 4 and in particular Secs. 4.4—4.6 give the innards of the algorithm. Sec. 5 discusses technical issues, including side-channel attack concerns. Some unusual maneuvers in linear algebra help to manage the large amounts of data efficiently during key generation on RAM-poor smart cards. We also explain why TTS can sign *and* generate keys quickly and efficiently. The real-time on-card public key generation capability ameliorates a problem affecting most multivariates *even when the keys are not stored on-card*, i.e., the large public key makes on-card key generation frequently infeasible and key management difficult (cf. [16]).

2 Principles of TTS and Our Sample Scheme $\mathrm{TTS}(20, 28)$

In a multivariate PKC, usually the public map is a composition $V = \phi_3 \circ \phi_2 \circ \phi_1$ with both $\phi_1 : \mathbf{w} \mapsto \mathbf{x} = \mathsf{M}_1\mathbf{w}+\mathbf{c}_1$ and $\phi_3 : \mathbf{y} \mapsto \mathbf{z} = \mathsf{M}_3\mathbf{y}+\mathbf{c}_3$ being affine and invertible. All arithmetic is done over a finite field (the *base field*) which in TTS is $K = \mathrm{GF}(2^8)$.

A digital signature scheme is considered to be in the TTS family ([2]) if ϕ_2 is a **tame map**, a polynomial map *with relatively few terms in the equations, easily invertible through serial substitution or solution of linear equations, but without a low degree explicit inverse*. Tame maps extend the concept of *Tame Transformations* from algebraic geometry ([10]), and may be said to combine the traits of triangular constructs (introduced to cryptography in [5], cf. also [6, 17]) and Oil-and-Vinegar ([7, 8]).

We refer the reader to [2] for some background, and [17] for a topical assessment of TTS. We will use for illustration the TTS instance exhibited in [17] with the following central map $\phi_2 : \mathbf{x} = (x_0, x_1, \ldots, x_{27}) \mapsto \mathbf{y} = (y_8, y_9, \ldots, y_{27})$:

$$\begin{aligned}
y_i &= x_i + \textstyle\sum_{j=1}^{7} p_{ij} x_j x_{8+(i+j \bmod 9)},\; i = 8 \cdots 16;\\
y_{17} &= x_{17} + p_{17,1}x_1x_6 + p_{17,2}x_2x_5 + p_{17,3}x_3x_4\\
&\quad + p_{17,4}x_9x_{16} + p_{17,5}x_{10}x_{15} + p_{17,6}x_{11}x_{14} + p_{17,7}x_{12}x_{13};\\
y_{18} &= x_{18} + p_{18,1}x_2x_7 + p_{18,2}x_3x_6 + p_{18,3}x_4x_5\\
&\quad + p_{18,4}x_{10}x_{17} + p_{18,5}x_{11}x_{16} + p_{18,6}x_{12}x_{15} + p_{18,7}x_{13}x_{14};\\
y_i &= x_i + p_{i,0}x_{i-11}x_{i-9} + \textstyle\sum_{j=19}^{i} p_{i,j-18}\; x_{2(i-j)}\; x_j\\
&\quad + \textstyle\sum_{j=i+1}^{27} p_{i,j-18}\; x_{i-j+19}\; x_j,\; i = 19 \cdots 27.
\end{aligned}$$

This central map works with 20-byte hashes and 28-byte signatures, and the corresponding TTS instance will be henceforth called $\mathrm{TTS}(20, 28)$.

To Generate Keys: Assign non-zero random values in $K = \mathrm{GF}(2^8)$ to parameters p_{ij}; generate random nonsingular matrices $\mathsf{M}_1 \in K^{28\times 28}$ and $\mathsf{M}_3 \in K^{20\times 20}$ (usually via LU decomposition) and vector $\mathbf{c}_1 \in K^{28}$. Compose $V = \phi_3 \circ \phi_2 \circ \phi_1$; assign $\mathbf{c}_3 \in K^{20}$ so that V has no constant part. Save quadratic and linear coefficients of V as public key (8680 bytes). Find M_1^{-1}, M_3^{-1}; save them with $\mathbf{c}_1$, $\mathbf{c}_3$, and the parameters p_{ij} as the private key (1399 bytes).

To Sign: From the message M, first take its digest $\mathbf{z} = H(M) \in K^{20}$, then compute $\mathbf{y} = \mathsf{M}_3^{-1}(\mathbf{z} - \mathbf{c}_3)$, then compute a possible $\mathbf{x} \in \phi_2^{-1}(\mathbf{y})$ as follows:

1. Randomly assign $x_1, \ldots, x_7$ and try to solve for x_8 to x_{16} in the first 9 equations. Since the determinant (for any $x_2 \cdots x_7$) of this system is a degree-9 polynomial in x_1, there can only be at most 9 choices of x_1 out of 256 to make the first system degenerate. Keep trying until we find a solution.
2. Solve serially for x_{17} and x_{18} using the next two equations (y_{17} and y_{18}).
3. Assign a random x_0 and try to solve for x_{19} through x_{27} from the last 9 equations. Again, at most 9 values of x_0 can make the determinant of the system zero. So keep trying new values of x_0 until a solution is found.

Our desired signature is $\mathbf{w} = \mathsf{M}_1^{-1}(\mathbf{x} - \mathbf{c}_1)$. Release $(M, \mathbf{w})$.

To Verify: On receiving $(M, \mathbf{w})$, compute $\mathbf{z} = H(M)$ and match with $V(\mathbf{w})$.

3 Summary of the 8051 Hardware Platform

The reader may find the details of the key-generation and signing processes (particularly those of Sec. 4.5) tedious, but all the contortions are necessitated by the fact that EEPROM memory cannot be reliably read from soon after it is written to. This is not uncommon in the embedded realm. This section provides an executive summary for those unfamiliar with the 8051 chip. Those who are already familiar with the 8051 can skip the rest of this section. Those interested in such things can please refer to the appendix.

Memory: The 8051 has 128 bytes of fast directly-addressable RAM on-board, called **data**. Some **data** locations are used as architectural stack and registers including the important registers `R0` and `R1`. The rest hold important system and user data.

Most extant 8051-compatibles have 128 bytes more fast RAM onboard, only addressable indirectly through the registers `R0` and `R1`. As the **data** can also be accessed this way, all 256 bytes are together called **idata**. In theory the 8051 can address 64kB of immutable memory (**code**) and 64kB of off-board RAM or EEPROM (**xdata**), both indirectly using the `DPTR` register. The **code** cannot be written to and can be accessed with an offset for table lookups. In practice usually all external memory are accessed identically. Accessing **code** and **xdata** takes twice as much time as **data** and **idata** besides being harder to set up for.

EEPROM and flash memory are accessed just like RAM except that *one must wait, usually for a few milliseconds, before accessing a recently written EEPROM location.* A write followed too soon by another access results in an error.

ALU: The Arithmetic-Logic Unit has many specialized registers (the most important being the accumulator `A` and the Data Pointer `DPTR`) and instructions. There is an integer multiply and divide, and instruction to increment – but not decrement – `DPTR`, so large arrays are best accessed in increasing order. Each instructions execute serially in a fixed number of **instruction cycles** (cf. paragraph after Tab. 1).

Resources: I/O is through specialized latches, and *communication to a computer or other device must be done through a reader unit, often attached to a PC on a USB port. The access is serial and slow say 1 kB/s.* Hardware random-number generation can be added cheaply to most cards. Cheap smart cards today have 4 kB or more in ROM and EEPROM, and a little RAM – sometimes 256B, frequently 0.5kB, sometimes 1 kB. 1.5kB or more RAM is mostly available in heavyweight cards.

4 Actual Performance and Implementation Details

We recompiled portable C code for $\mathrm{TTS}(20, 28)$ (cf. [17]), with C51, a popular embedded compiler. A few routines were rewritten in assembly. Test results from standard development kits are given in Sec. 4.2 and implementation details in Sec. 4.4–4.6.

4.1 Hardware Resource Requirements

As mentioned in Appendix A.3, a low-end card can either just sign or do both key generation and signing. We list our requirements in RAM (**data**, **idata** and external RAM or **xdata**, see Appendix A.1), EEPROM, total ROM, etc.:

To Sign: 128 bytes of RAM (19 bytes of **data** for state, 109 more bytes in **data/idata** or **xdata** for temporary storage), 1.3kB of EEPROM for private keys, 1.6kB of further **code** space (0.8kB of program, 0.8kB of tables). For controllers with 256 bytes of on-board RAM (**idata**), it is an option to keep all the data for the Gaussian elimination in the **idata**, which means shorter code (no need to move things between **idata** and **xdata**) at 1.4kB (0.2kB less) and at least a 12% speed up. Since we must have some RAM to put the matrix for the elimination stage, a plain-vanilla Intel 8051 will be assumed to have least 128 bytes of **xdata** RAM storage.

Both To Sign and To Generate Keys: There are two choices:

- On EEPROM-poor cards, we do not store the entire public key. During setup only the private key and some critical intermediate data are generated and stored in EEPROM, enough that chunks of the public key can be computed and output on-the-fly as requested. This requires 2.7 kB of EEPROM (1399B in the private key plus 1212B in intermediate data, plus some left-over wasted space) plus 4.2 kB more **code** space (in ROM or EEPROM) is required. There is 3.8 kB in the program for both key generation and signing, 0.4 kB in subsidiary I/O, including 0.8kB for tables as above.
- We can compute and store the entire public key for later retrieval. This takes 11.3kB of EEPROM space, plus 4.2 kB more ROM or EEPROM **code** space.

In both cases we need 128 bytes in **data**, **idata** or **xdata** storage. If we need block-writes to EEPROM or do block-outputs from the smart card, we will also need 128 more bytes of RAM for buffer. When we can afford to usually we do the entire Gaussian elimination from **idata**. PC access provided through USB-port device.

4.2 Performance Data and Brief Description of Programs

The signing portion of the program is straightforward and can be implemented straight out of Sec. 2. *On average, signing takes about 170ms on a 3.57MHz stock Intel 8051AH (a really low-end part).* Every other part is faster than the 8051AH. The same code running on a 3.57MHz ("4T") WinBond W77E58 only takes 64ms.

For reference, of the 170ms average time taken by the signing operation on the Intel 8031/32 at 3.57MHz is divided as follows: The ϕ_3 portion takes 34ms, ϕ_2 71ms, and ϕ_1 65ms. On a Winbond W77E58, the times are 13ms, 27ms, 24ms respectively. *Using*

*10MHz parts, the speedup is almost linear. The W77E58 takes about 23ms to sign and the Intel 8032AH takes 61ms — 51ms if we run the entire elimination from **idata**.*

Once we get to 16MHz or faster clocks, some instructions require extra *instruction cycles* to execute, and I/O times start to dominate, so the scaling is a lot less than linear.

The process for key generation is a lot more complicated and slower than signing. When the smart card is initialized, we must first generate M_1, M_1^{-1}, M_3, M_3^{-1} via LU decomposition, store to EEPROM, then generate $(p_{ij})_{8 \le i \le 27}$ and $\mathbf{c}_1$, compute $\mathbf{c}_3$ and store everything in EEPROM along the way. Note that $\mathbf{c}_3 = \mathsf{M}_3\left(\phi_2(\mathbf{c}_1)\right)$ and hence:

$$(\mathbf{c}_3)_k = \sum_{\ell=n-m}^{n-1} \left[(\mathsf{M}_3)_{k,(\ell-n+m)} \left((\mathbf{c}_1)_\ell + \sum_{p\, x_\alpha x_\beta \text{ in } y_\ell} p\, (\mathbf{c}_1)_\alpha (\mathbf{c}_1)_\beta \right) \right]. \quad (1)$$

The sum is over each term $p\, x_\alpha x_\beta$ in the equation for y_ℓ. We may end the setup process here, and the generated information is enough to compute the coefficients of the public key polynomials, 20 at a time. In this mode, the card awaits an appropriate outside input before signing or computing and emitting the public key in small chunks on-the-fly.

Setting up on an Intel 8032AH at 3.57MHz (computing the private key and intermediate data) takes 7.7 seconds, including a little error checking. The process takes 3.6 seconds on a 3.57MHz Winbond W77E58.

The rest of public-key generation is to compute for each (i, j) the coefficients of $w_i w_j$ or w_i^2 or w_i in z_k for every k at once. To show that this is possible, we will follow Imai and Matsumoto ([9]) and divide the coefficients involved in each public key polynomial into linear, square, and crossterm portions as follows:

$$z_k = \sum_i P_{ik} w_i + \sum_i Q_{ik} w_i^2 + \sum_{i>j} R_{ijk} w_i w_j. \quad (2)$$

The coefficients $(P_{ik}, Q_{ik}, R_{ijk})$ are related to M_1, M_3, $\mathbf{c}_1$, and the parameters $(p_{ij})_{8 \le i \le 27}$ as follows, where each sum is over the terms $p\, x_\alpha x_\beta$ in the equation for y_ℓ:

$$P_{ik} = \sum_{\ell=n-m}^{n-1} \left[(\mathsf{M}_3)_{k,(\ell-n+m)} \left((\mathsf{M}_1)_{\ell i} + \sum_{p\, x_\alpha x_\beta \text{ in } y_\ell} p\, \left((\mathsf{M}_1)_{\alpha i} (\mathbf{c}_1)_\beta + (\mathbf{c}_1)_\alpha (\mathsf{M}_1)_{\beta i} \right) \right) \right] (3)$$

$$Q_{ik} = \sum_{\ell=n-m}^{n-1} \left[(\mathsf{M}_3)_{k,(\ell-n+m)} \left(\sum_{p\, x_\alpha x_\beta \text{ in } y_\ell} p\, (\mathsf{M}_1)_{\alpha i} (\mathsf{M}_1)_{\beta i} \right) \right] \quad (4)$$

$$R_{ijk} = \sum_{\ell=n-m}^{n-1} \left[(\mathsf{M}_3)_{k,(\ell-n+m)} \left(\sum_{p\, x_\alpha x_\beta \text{ in } y_\ell} p\, \left((\mathsf{M}_1)_{\alpha i} (\mathsf{M}_1)_{\beta j} + (\mathsf{M}_1)_{\alpha j} (\mathsf{M}_1)_{\beta i} \right) \right) \right] (5)$$

Here $m = 20$, $n = 28$. For a smart card equipped with a lot of EEPROM or flash, we need not compute and emit the public key piecemeal. It is possible to compute everything right there and write everything to EEPROM, to be read at a later time.

A 3.57MHz Intel 8051 or 8032AH averages about 150ms to generate a 20-byte block of the public key from the intermediate data and signal that it is ready to send. *On a 3.57MHz Winbond W77E58 with sufficient storage, generating the entire public key takes 33 seconds.* It takes some 15 seconds to transmit everything from card to PC.

4.3 Finite Field Arithmetic and Other Details

As in any other multivariate PKC, we need to represent each element of $\mathrm{GF}(2^8)$ as an integer between 0 and 255 (an `unsigned char`). We choose the "Standard" representation used by AES ([4]), but we could choose any encoding as long as addition can be represented as a bitwise `xor`.

A standard way of implementing finite field multiplication is to choose a fixed primitive element[4] $g \in \mathrm{GF}(2^8)$ and store logarithm and exponential look-up tables in base g, intending to multiply non-zero x and y as $xy = g^{(\log_g x + \log_g y) \bmod 255}$. We will do a lot of manipulations of data in *log-form*, which means we represent the zero of the field $\mathrm{GF}(2^8)$ by the byte 0, and any other field element a by the unique positive integer $x \leq 255$ that satisfies $a = g^x$. *Note: 1 is represented as g^{255}, not g^0.*

In implementing the signing portion of the algorithm, we need the following data in ROM: 256-byte log-table in base g; 512-byte table of exponentiation ($x \mapsto g^x$), this can be shortened to 256 bytes at a roughly 15% speed penalty.

The private key comprises the matrices $(\mathsf{M}_1^T)^{-1}$ and $(\mathsf{M}_3^T)^{-1}$, parameters (p_{ij}) of the central map, and the vectors $\mathbf{c}_1$, $\mathbf{c}_3$. We store everything except the vectors $\mathbf{c}_1$ and $\mathbf{c}_3$ in log_g-form, and the matrices column-first (as indicated by the transposed notation).

The intermediate key-generation data are M_1^T, M_3^T (in column-first, $\log_g$-form), and a componentwise log of $\mathbf{c}_1$. The public key consists of coefficients (P_{ik}), (Q_{ik}), $(R_{ijk})_{i>j}$, with each block arranged in order of increasing i, j, then k.

4.4 The Signing Process

The actual signing program operates on a 20-byte array $\mathbf{z}$ in **idata** in three stages, corresponding to ϕ_1^{-1}, ϕ_2^{-1}, and ϕ_3^{-1}. Due to the amount of array access in the Gaussian elimination, ϕ_2^{-1} takes most of the time. If we put the entire system matrix in **idata** we can save at least 10 percent upwards of running time, but most often we forego that and do it mostly from **xdata** due to memory resource problems.

1. Do $(\phi_3)^{-1}$, which is essentially a matrix multiplication, as follows:
   (a) Zero out a 20-byte area $\mathbf{y}$ and replace $\mathbf{z}$ by $\mathbf{z}'$, the componentwise log of $\mathbf{c}_3 + \mathbf{z}$;
   (b) looping over $i = 19, 18 \cdots 0$ and do the following loop if $z_i' \neq 0$:
   (c) Looping over $j = 19 \cdots 0$, when $(\mathsf{M}_3)_{ji} \neq 0$, add (`xor`) $g^{(z_i' + \log_g (\mathsf{M}_3)_{ji}) \bmod 255}$ into y_{j+8}. **Note:** M_3 is stored in log-form and transposed so that it can be accessed sequentially, and we can compute $(R + \mathtt{A}) \bmod 255$ in only two instructions: `add A,` R (add register R to the accumulator `A`) then `adc A,#0` (add the carry from the last add into accumulator).

   The inner loop of this routine reads coefficients off a table, multiplies to a variable, then adds them to different components of a vector (cf. also Sec. 5.3).
2. Do $(\phi_2)^{-1}$, which is performed as follows:
   (a) For $i = 1 \cdots 7$, generate randomly and save (in an **idata** array) $\log_g x_i$, again with the proviso that 0 represents 0, not 1 which is represented by 255.
   (b) Establish in a 90-byte block `BA` (in **xdata** or **idata**) of RAM the first linear system to be solved for $x_8, \ldots, x_{16}$ by doing the following for $i = 8 \cdots 16$:

[4] We chose as g the canonical generator of the AES field representation.

i. The constant of each equation location ($\mathtt{BA}[10(i-8)+9]$) is filled with y_i.
ii. Looping over $j = 1..7$, insert into the location corresponding to the coefficient of $x_{(i+j \pmod 9))+8}$ (location $\mathtt{BA}[10(i-8) + (i + j \pmod 9))]$).
iii. `xor` $\mathtt{BA}[10(i-8) + (i + 1 \pmod 9))]$ with 1.
iv. Let $\mathtt{BA}[10(i-8)+(i+8 \pmod 9))] = \mathtt{BA}[10(i-8)+(i+9 \pmod 9))] = 0$.

(c) Run elimination on 9 variables to get $x_8, \ldots, x_{16}$, then find x_{17} and x_{18} by solving for them in the next two equations (all x_i will be stored in log-form).

(d) Establish another system of equations in BA by looping over $i = 19 \cdots 27$:
i. Insert $y_i + p_{i0}x_{i-11}x_{i-9}$ as the constant term (location $\mathtt{BA}[10(i-19)+9]$).
ii. Looping over $j = 0 \cdots i-19$, let $\mathtt{BA}[10(i-19)+j] = p_{i,j+1}x_{2(i-j-19)}$.
iii. Looping over $j = i-19 \cdots 8$, let $\mathtt{BA}[10(i-19)+j] = p_{i,j+1}x_{i-j}$.
iv. `xor` $\mathtt{BA}[10(i-8) + (i + 1 \pmod 9))]$ with 1.

(e) Run elimination on 9 variables to obtain $x_{19}, \ldots, x_{27}$ (again in log-form).

3. Do $(\phi_1)^{-1}$, another matrix multiplication like $(\phi_3)^{-1}$, with different parameters.

4.5 Key Generation, First Half: Generating M_1, M_3, and Their Inverses

The following routine computes and stores M_1^T, $(\mathsf{M}_1^T)^{-1}$, M_3^T, $(\mathsf{M}_3^T)^{-1}$, (p_{ij}) for $8 \le i \le 18$, $1 \le j \le 6$ and $19 \le i \le 27$, $0 \le j \le 9$, $\mathbf{c}_1$, $\log_g \mathbf{c}_1$, and $\mathbf{c}_3$. Total EEPROM space required is 2768 bytes, with 1399 bytes in private keys ($(\mathsf{M}_1^T)^{-1}$, $(\mathsf{M}_3^T)^{-1}$, the (p_{ij}), $\mathbf{c}_1$, $\mathbf{c}_3$) and 1212 bytes of intermediate data to be used to produce the public keys. There are 157 bytes used and erased. No more RAM than the 256 bytes of **idata** is needed; in fact, only 128 bytes are necessary if a write buffer is not needed. Of course, extra RAM helps. Recall that matrices M_1, M_1^{-1}, M_3, M_3^{-1} are stored transposed and in log-form (cf. Sec. 4.3) for convenience.

1. Generate matrices M_1 and $(\mathsf{M}_1)^{-1}$ via LU decomposition.

(a) Generate and write to EEPROM entries (in log-form) of the diagonal matrix D_1 (28 non-zero bytes), the lower triangular matrix L_1 ($28 \times 27/2 = 378$ bytes, also in log-form), and the upper triangular matrix U_1 (same as above), and do so in the area from the 1569th to 2352th bytes from the beginning (hence, leaving the first 1568 bytes empty). The entries of L and U are generated in an unusual format. We line L up column-first, but U will be in *column-first but reverse order*, i.e.: $L_{10}, L_{20}, \ldots, L_{n-1,0}, L_{21}, \ldots, L_{n-1,1}, \ldots, L_{n-1,n-2}$ and $U_{n-2,n-1}, \ldots, U_{0,n-1}, U_{n-3,n-2}, \ldots, U_{0,n-2}, \ldots, U_{23}, U_{13}, U_{12}$.

(b) Invert D_1, L_1, U_1 and write to EEPROM (in the first 784 bytes). Inverting D_1 is easy. We invert L_1 into L_1^{-1} (stored in the same format) as follows:
i. Repeat (ii.–v.) over $i = 1, 2, \ldots, n-1$:
ii. Read $[(L_1)_{i,i-1}, \ldots, (L_1)_{n-1,i-1}]$ from **xdata** into $[z_i, \ldots, z_n]$ in **idata**.
iii. For each $j = i, \ldots, n-1$ where $z_j \neq 0$, replace z_j by $(\log_g z_j)$ and do:
iv. For each $k = j+1, \ldots, n-1$ such that $L_{kj} \neq 0$, add $g^{m+L_{kj}}$ to z_k.
v. Now $[z_i, \ldots, z_n]$ is the column $[\log_g(L_1^{-1})_{i,i-1}, \ldots, \log_g(L_1^{-1})_{n-1,i-1}]$. Write to EEPROM, or (for parts with special EEPROM/flash writing rules) copy to a 128-byte buffer and block-write only if the buffer is full.

The same subroutine can invert U_1 into U_1^{-1} in the same inverted column order.

(c) Compute $\mathsf{M}_1^{-1} = U_1^{-1}D_1^{-1}L_1^{-1}$ and write to EEPROM in the next 784 bytes:

i. Read $[\log_g((D_1^{-1})_{jj})]$ into array $[d_j]$ in **idata**; repeat (ii.–v.) for $i < n$.
ii. Zero out z_0 to z_{i-1} (array $z_0, \ldots, z_{n-1}$ is in **idata**); let $z_i = d_j$; for $i+1 \leq j \leq n-1$ let $z_j = \left(d_j + \log_g(L_1^{-1})_{ji}\right) \bmod 255$. Note (cf. 1b) that $\log(L_1^{-1})_{ji}$ was stored serially.
iii. Looping over $j = i, i+1, \ldots, n-1$, do the following:
iv. For $k = 0, \ldots, j-1$, add $g^{(z_j + \log_g(U_1^{-1})_{kj}) \bmod 255}$ into z_k. Note that U_1^{-1} is in reverse order. After the k-loop, replace z_j by g^{z_j}.
v. Now (end of j-loop) the $[z_j]$ array holds the i-th column of M_1. Take the componentwise log, then write appropriately into EEPROM (cf. 1b).

Note that we used n instead of 28 because the same routines are used for M_3.

(d) Erase the first 784 bytes, the memory block used for $D_1^{-1}, L_1^{-1}, U_1^{-1}$.
(e) Compute (and write out to the freshly erased block of 784 bytes) $\mathsf{M}_1 = L_1 D_1 U_1$.
i. Read $[\log_g((D_1)_{jj})]$ into array $[d_j]$ in **idata**; repeat (ii.–v.) for $i < n$.
ii. Read $\left(\log_g(U_1)_{ji} + d_j\right) \bmod 255$ to z_j (in **idata**) for $j = 0 \cdots i-1$.
iii. Let $z_i = d_i$. For $j = 0 \cdots i$, let $y_i = g^{z_i}$, and zero out $y_{i+1}, \ldots, y_{n-1}$.
iv. Looping over $0 \leq j \leq i$, $j \leq k < n$, add $g^{\left(z_k + \log_g(L_1)_{kj}\right) \bmod 255}$ into y_k.
v. The $(\log y_k)$ is the i-th column of $\log_g \mathsf{M}_1$, write to EEPROM.
(f) Erase the 784-byte EEPROM block used for D_1, L_1, U_1.

We should conclude with 1568 bytes of data and 784 freshly erased bytes.

2. Generate M_3 and $(\mathsf{M}_3)^{-1}$ as above, reusing the memory from Step 1. We should have written 800 bytes of data and have 400 recently erased bytes.
3. Generate and store (p_{ij}) for $8 \leq i \leq 18$, $1 \leq j \leq 7$ and $19 \leq i \leq 27$, $0 \leq j \leq 9$ (167 bytes), reusing the memory from Step 2.
4. Generate $\mathbf{c}_1$ (28 bytes) and store in space left from Step 2. Compute $\mathbf{c}_3$ (20 bytes):
(a) Storing (in **idata**) in the arrays $\mathbf{c}$ and $\mathbf{y}$ the componentwise log of $\mathbf{c}_1$.
(b) Reading from the parameter table, compute $\phi_2(\mathbf{c}_1)$ by looping over all indices $i = 8 \cdots 27$, adding each cross-term into the appropriate y_i.
(c) Write the $\log(\mathbf{c}_1)_i$ to EEPROM. Take the logs of y_i; jump to the multiplication routine in Sec. 4.2. The result $(\mathbf{c}_3)$ is written to EEPROM. We are done.

4.6 Key Generation, Second Half: Computing and Outputting the Public Key

After the process of Sec. 4.5, we can generate the public key in units of 20 bytes (see Sec. 4.2). Generating the public key takes a long time, but the routine itself is simpler:

Generating R_{ijk}: It is more convenient to compute first the coefficients of $w_i w_j$ in y_k:
1. Read in the i-th and j-th columns of $\log_g \mathsf{M}_1$ and place in the arrays $\mathbf{c}$ and $\mathbf{c}'$.
2. Zero out the array $\mathbf{z}$ and loop over each cross-term $p\ x_\alpha x_\beta$ in the equation for y_i (reading off parameter table) thusly:
3. Compute $p\ [(\mathsf{M}_1)_{\alpha i}(\mathsf{M}_1)_{\beta j} + (\mathsf{M}_1)_{\alpha j}(\mathsf{M}_1)_{\beta i}]$ via $\log_g\left(g^{c_\alpha + c'_\beta} + g^{c'_\alpha + c_\beta}\right)$, adding $\log_g p$ (read off the parameter table) and exponentiating. Add to z_i.
4. We now have the coefficient of $x_i x_j$ in y_k. Take the logs and multiply by M_3 and jump to the matrix multiplication routine (Sec. 4.2) to get R_{ijk} for each k.

Generating P_{ik}: As above, except for initializing the array $\mathbf{z}$ to $[(\mathsf{M}_1)_{ki}]_{k=8\cdots27}$ instead of zeroing out $\mathbf{z}$, and reading in $\mathbf{c}_1$ and the i-th column of M_1 instead of the i-th and j-th columns of M_1.

Generating Q_{ik}: Like the above, but we only need to read a single column of the M_1, and it is faster because there is one fewer add and one fewer multiply in Eq. 4.

After each block of 20 is produced, write it to EEPROM or accumulate for buffered output as needed. Our test code outputs every 6 blocks to get max throughput.

5 Discussions and Conclusion

We discuss some issues germane to our study including side channel attack considerations, optimization, and possible changes to the scheme to suit smart cards better.

5.1 Why does TTS have Faster Key Generation?

The state-of-the-art in key generation for multivariate PKC's is probably the kind of procedures as given by C. Wolf ([16]). At least, we can find no better, and he managed to save 30% of running time from previous algorithms. However Wolf states, and it seems commonly agreed to, that computations of the public polynomial using interpolation for large field multivariates, i.e. HFE ([13]) or C^*-derivatives where the private map really operates on some googol-sized field, take time proportional to n^6. A cursory look at Eqs. 3–5 will reveal that the number of multiplications in key generation is about n^4 (really m^2n^2) times the average number of terms in an equation in ϕ_2, hence $O(n^4)$. So we expect key generation in TTS to run at about a few hundred times the speed that SFLASH might need if they use the same dimensions.

Timings given in [2, 17] support this hypothesis. We do not claim to be anywhere close to as good 8051 programmers as the authors of [1], but the factor of n^4 vs. n^6 gives an edge that makes on-card key generation passably quick as opposed to snail-like. In general, a multivariate PKC can be called *tame-like* ([17]) if its central map has relatively few terms per equation and has a fast inverse. Sec. 4.6 and Eqs. 3–5 demonstrate tame-like-ness are useful for a smart card.

5.2 Side Channel Attack Considerations

In [1], the authors discuss defending against a possible differential-power attacks. The structure of SFLASH^{v2} is somewhat simpler, but we can take similar precautions against DPA probes to those in [1]. The steps for a DPA-safe signing are:

1. Start out with hash value $\mathbf{z}$. Take a random vector $\mathbf{z}' \in K^m$. Compute $\mathbf{z}'' = \mathbf{z} + \mathbf{z}'$.
2. Compute $\mathbf{y}' = (\mathsf{M}_3)^{-1}(\mathbf{z}' - \mathbf{c}_3)$ and $\mathbf{y}'' = (\mathsf{M}_3)^{-1}\mathbf{z}''$. We see that $\mathbf{y} = \mathbf{y}' + \mathbf{y}''$.
3. Take random bytes x_i', x_i'' for $i = 0 \cdots 7$. Loop until the systems are solvable.
4. Construct linear systems as in Sec. 4.4, except that we use twice as large a RAM buffer, put in two systems: One filled in using the x_i' and y_i', one with the x_i'' and y_i''. Note: Step 2(b)iii of Sec. 4.4 only needs to be performed on one of the matrices.
5. Run a "conjoined Gaussian" with the two matrices. A key operation is division by the sum of the two coefficients at the pivot position. If they are $c_{ii}' \neq c_{ii}''$, then we can achieve division by the pivot coefficient $(c_{ii}'+c_{ii}'')$ through dividing successively with c_{ii}' and then by $(1 + c_{ii}''/c_{ii}')$. For 9×9 matrices, this means slightly less than triple the number of multiplications and is time-consuming, but we eventually come down to $(x_8' \cdots x_{16}',\, x_8'' \cdots x_{16}'')$, where $x_i' + x_i'' = x_i$.

6. For $i = 17$ and 18, do the following so that $x_i = x'_i + x''_i$.

$$x'_i = y'_i + \sum_{px_\alpha x_\beta \text{ in } y_i} p \cdot (x'_\alpha x'_\beta + x''_\alpha x''_\beta);\ x''_i = y''_i + \sum_{px_\alpha x_\beta \text{ in } y_i} p \cdot (x''_\alpha x'_\beta + x'_\alpha x''_\beta). \quad (6)$$

7. Similarly for $i = 19 \cdots 27$ we do the "conjoined Gaussian". We have found $\mathbf{x}'$ and $\mathbf{x}''$ that sum to the $\mathbf{x}$ of Sec. 2 at about one-quarter speed.
8. Compute $\mathbf{w}' = (\mathsf{M}_1)^{-1}\mathbf{x}'$ and $\mathbf{w}'' = (\mathsf{M}_1)^{-1}(\mathbf{x}'' - \mathbf{c}_1)$. Output $\mathbf{w} = \mathbf{w}' + \mathbf{w}''$.

Since TTS$(20, 28)$, like SFLASH^{v2} uses each byte of the entire key continuously, it should be as safe as SFLASH^{v2} under the same attacks. The signing code expands to about 3.2 kB, and the speed is between a third and a quarter of what it was (still ok). This is obviously not optimal programming and we should be able to optimize it better.

5.3 Optimization Concerns

Since we are only doing proof-of-concept work we used C51-compiled code with only a few key routines hand-coded 8051 assembly. This is not the optimal performance available from our hardware, but is a lot better than using just C51, and saves many man-hours compared to doing assembly from the ground up. We feel that we are not giving up much in the way of performance because this is common practice in software design. A local expert on the 8051 offered his opinion that we might be able to improve it by a factor of two in pure assembly. However, the quaint, quirky 8051 architecture severely restricts our options. There is only so much possible with our limited resources, especially the single pointer into **xdata**. Some notes on optimization possibilities:

– Of particularly note is the idiosyncratic ways the arrays were arranged in memory. *We do not have hard proofs but believe that this arrangement is already correctly optimized under the circumstances.*
– If the μC has access to 1.5kB of **xdata** RAM, all the temporary data can stay in RAM and we need not erase from EEPROM at all during key generation. Even 1 kB RAM would make life easier.
– The `movx A,@Ri` command that use the output latch `P0` for the high byte of the address line and the register `R0` or `R1` for the low byte is seldom used, because I/O lines from the outside often get in the way, changing the value of the latch when it shouldn't. With custom solutions, it is possible to have programs up to 20% faster by using these specialized commands.
– An $n \mapsto \log_g(1 + g^n)$ table can help. Let this table be `E`. When computing R_{ijk} (Sec. 4.6), we need to do $\left[\log_g\left(g^{c_\alpha + c'_\beta} + g^{c'_\alpha + c_\beta}\right) + \log_g p\right] \bmod 255$. Instead, we can save lookups via $\left[(c_\alpha + c'_\beta) + \mathtt{E}[c'_\alpha + c_\beta - (c_\alpha + c'_\beta)] + \log_g p\right] \bmod 255$.
– We never really rewrote our programs for parts with dual `DPTR`'s. With proper utilization, this is said to increase the speed by some 25% for array-heavy code.
– With the base field $K = \mathrm{GF}(2^7)$, the signing can run with the exponential table in **idata** (did the designer of SFLASH think of this?). More extremely, with $K = \mathrm{GF}(2^6)$, both the log and the exponential table can fit into the **idata**. We can change to a $(\mathrm{GF}(2^7))^{28} \mapsto (\mathrm{GF}(2^7))^{36}$ version of TTS for speed if key size and extra RAM storage is not a problem. Preliminary tests show that it is about 13% faster.

5.4 Conclusion

We believe that our test result shows the TTS family of multivariate signature schemes in general and $\text{TTS}(20, 28)$ in particular to be very speedy. Not only is it fast, but it also consumes little enough resources that it definitely merits further attention as an on-card solution. The family of schemes has great potential for use in low-cost smart cards, especially those without a cryptographic coprocessor.

We note that $\text{TTS}(20, 28)$ by current estimates should be at least as secure as RSA-1024, and $\text{TTS}(24, 32)$ at least as secure to RSA-1536 (cf. [17]). Even if this estimate is slightly off, the speed of the implementations should still make TTS quite useful for smart cards. Furthermore, TTS is clearly an extensible system and we did an implementation for the following central map ([17])

$$
\begin{aligned}
y_i &= x_i + \textstyle\sum_{j=1}^{7} p_{ij} x_j x_{8+(i+j+1 \bmod 10)},\ i = 8 \cdots 17; \\
y_i &= x_i + p_{i1} x_{i-17} x_{i-14} + p_{i2} x_{i-16} x_{i-15} + p_{i3} x_{i-10} x_{i-1} + p_{i4} x_{i-9} x_{i-2} \\
&\qquad + p_{i5} x_{i-8} x_{i-3} + p_{i6} x_{i-7} x_{i-4} + p_{i7} x_{i-6} x_{i-5},\ i = 18 \cdots 21; \\
y_i &= x_i + p_{i,0} x_{i-10} x_{i-14} + \textstyle\sum_{j=22}^{i} p_{i,j-21}\ x_{2(i-j)}\ x_j \\
&\qquad + \textstyle\sum_{j=i+1}^{31} p_{i,j-21}\ x_{i-j+21}\ x_j,\ i = 22 \cdots 31.
\end{aligned}
$$

This scheme has $m = 24$ (192-bit hashes) and $n = 32$. We tabulate our new test results with the old. Again, only the i8032AH code ran the elimination entirely in **idata**.

As we observed in Sec. 5.3, we expect to improve on our work still. All in all, we think that we have shown TTS and similar variants are worth more attention.

Scheme	Controller	PrivKey Length	Signing Time	Signing Code	Keygen Time	Keygen Code	Extra EEPROM	Setup Time
$\text{TTS}(20, 28)$	i8032AH	1399 B	144ms	1.5 kB	78.5 s	4.2 kB	1.2 kB	7.7s
	i8051AH		170ms	1.6 kB				
	W77E58		60ms		38.3 s			3.6s
$\text{TTS}(24, 32)$	i8032AH	1534 B	191ms	1.5 kB	134 s		1.6 kB	11.7s
	i8051AH		227ms	1.6 kB				
	W77E58		85ms		65.2 s			5.4s

Table 2. Summary for TTS on a 8051

Acknowledgments

We thank Messrs. Po-Yi Huang and Sam Tsai of Solutioninside Inc. for technical assistance, and to Messrs. Bo-Yuan Peng and Hsu-Cheng Tsai of National Taiwan University for commentary and discussion. We thank the anonymous referees for their suggestions and constructive criticism. The first author would also like to thank his beloved Ping.

References

1. M. Akkar, N. Courtois, R. Duteuil, and L. Goubin, *A Fast and Secure Implementation of SFLASH*, PKC 2003, LNCS v. 2567, pp. 267–278.
2. J.-M. Chen and B.-Y. Yang, *A More Secure and Efficacious TTS Scheme*, ICISC '03, LNCS v. 2971, pp. 320-338; full version at `http://eprint.iacr.org/2003/160`.

3. D. Coppersmith, J. Stern, and S. Vaudenay, *The Security of the Birational Permutation Signature Schemes*, Journal of Cryptology, 10(3), 1997, pp. 207–221.
4. J. Daemen and V. Rijmen, *The Design of Rijndael, AES - The Advanced Encryption Standard.* Springer-Verlag, 2002.
5. H. Fell and W. Diffie, *Analysis of a Public Key Approach Based on Polynomial Substitution*, CRYPTO'85, LNCS v. 218, pp. 340–349.
6. L. Goubin and N. Courtois, *Cryptanalysis of the TTM Cryptosystem*, ASIACRYPT 2000, LNCS v. 1976, pp. 44–57.
7. A. Kipnis, J. Patarin, and L. Goubin, *Unbalanced Oil and Vinegar Signature Schemes*, CRYPTO'99, LNCS v. 1592, pp. 206–222.
8. A. Kipnis and A. Shamir, *Cryptanalysis of the Oil and Vinegar Signature Scheme*, CRYPTO'98, LNCS v. 1462, pp. 257–266.
9. T. Matsumoto and H. Imai, *Public Quadratic Polynomial-Tuples for Efficient Signature-Verification and Message-Encryption*, EUROCRYPT'88, LNCS v. 330, pp. 419–453.
10. T. Moh, *A Public Key System with Signature and Master Key Functions*, Communications in Algebra, 27 (1999), pp. 2207–2222.
11. The NESSIE project webpage: `http://www.cryptonessie.org`.
12. J. Patarin, *Cryptanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt'88*, CRYPTO'95, LNCS v. 963, pp. 248–261.
13. J. Patarin, *Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms*, EUROCRYPT'96, LNCS v. 1070, pp. 33–48.
14. J. Patarin, L. Goubin, and N. Courtois, C^*_{-+} *and HM: Variations Around Two Schemes of T. Matsumoto and H. Imai*, ASIACRYPT'98, LNCS v. 1514, pp. 35–49.
15. J. Patarin, N. Courtois, and L. Goubin, *FLASH, a Fast Multivariate Signature Algorithm*, CT-RSA 2001, LNCS v. 2020, pp. 298–307. Updated version available at [11].
16. C. Wolf, *Efficient Public Key Generation for Multivariate Cryptosystems*, preprint, available at `http://eprint.iacr.org/2003/089/`.
17. B.-Y. Yang and J.-M. Chen, *Rank Attacks and Defence in Tame-Like Multivariate PKC's*, see `http://eprint.iacr.org/2004/061`.

A Architecture of 8051-Based Smart Cards

We summarize the specifics of our hardware platforms for implementation and testing. Some discussion about clock vs. actual execution speed can be found in Sec. 1 (following Tab. 388).

A.1 Storage Areas on an 8051-Based Smart Card

The general structure of an 8051-based 8-bit microcontroller (μC) core is the same across implementations. The chip usually has a CPU portion plus some extra memory. On a 8051-like device, we have the following different locations for data storage:

data The 128 bytes of on-chip high-speed RAM directly addressable by the 8051 core in one instruction cycle. Sometimes referred to as "Register RAM" because in effect they are registers, like the zeroth page of the 6502 and other 8-bit CPUs. A peculiarity of the 8051 instruction set is that some **data** can be accessed bitwise as flags. This saves valuable **data** space as well as instructions.

idata On-chip high-speed RAM that may be accessed in indirect address mode through a byte-long pointer (either of the special registers `R0` or `R1`) in one instruction cycle. The **data** can also be accessed this way and is considered part of the **idata**. Almost every 8051-compatibles has the 128 extra bytes of **idata** for 256 bytes total.

code ROM, not writable. May be only read indirectly through the Data Pointer (`DPTR`) register with a fixed latency (normally 2 instruction cycles on a cheap part) via a special `movc` command. An 8051-like μC can address up to 64 kB of **code** ROM.

xdata Off-chip read-write memory, accessible only via the `movx` command, with indirection through either `DPTR` or a combination of the I/O port register `P0` and either of the special register `R0` and `R1`. Normally `movx` takes 2 instruction cycles, but on a high-end part there may be banks of memory with different latency, and a special register to control the access time.

One expect **xdata** to be RAM, and because the `movx/movc` commands both set signals lines, effectively adding an extra address line, an 8051-like part can theoretically address 64kB of ROM (**code**) and 64kB of RAM (**xdata**) for 128kB memory. However, when the μC does not use more than 64kB total, as a rule the control lines are wired together for convenience[5], and code and data are read identically. Another important point is that read-write memory can be EEPROM or flash memory as well as RAM:

- RAM for a μC is usually costly SRAM. In theory there may be as much as 64kB, but there is often only 256B, seldom more than 1kB and almost never more than 2kB. *A smart card intended for RSA or ECC work will always have at least 1 kB (usually 1.5kB), because there is a lot of data and co-processors also need memory.*
- We will use EEPROM and flash memory as synonyms like most people, although EEPROM can often be written to much faster and far more times than flash.

  A μC may have no EEPROM at all or as much as 64kB. Reading EEPROM is just like reading off-chip RAM, but completing a write into one EEPROM location takes about 50μs and erasing (resetting it into a writable state) takes much longer, about 5ms per access. Often erasure and sometimes writes must be by *lines*, which are units of access that may be 8 to 128 bytes each. Control signals from the EEPROM can be polled via I/O port latches to tell the programmer whether the EEPROM is ready for writing or successfully written. *After an EEPROM address is written to but before the requisite amount of time (some 100 instruction cycles or more) has elapsed, reading that location generates unpredictable results.*

There are several modes of writing into EEPROM/flash, and a given part normally does only one. There are 8051 parts with safeguards against loss of power and a block-write operation with a same latency of 5ms per 128-byte block, but these tend to be very expensive parts. In parts without the power-failure guard and block-writing features, EEPROM is essentially accessed like RAM, but the program needs to check manually the signal lines. If you are safety-minded, you check first, write, then keep checking the signals to make sure that it is done properly. A more cavalier designer would go about his business after the write is issued, but would presumably do some error-checking.

[5] `movx` also can often be faster than `movc` on high-end parts for a speed advantage.

A.2 The 8051-Based Parts We Tested and Their Characteristics

Intel 8051AH One of the original MCS-51 NMOS single-chip 8-bit microcontroller parts, with 32 I/O lines, 2 Timers/Counters, 5 Interrupts/2 Priority Levels, and 128 bytes on-chip RAM (i.e. no extra **idata**). It has 2 kB of on-chip ROM.

Intel 8032AH A stripped-down 8052; just like the i8051 except for having 3 Timers (Counters), 6 Interrupts/4 priority levels, no ROM and 128B more RAM on-chip.

Winbond W77E58 A more upscale CMOS part, with 36 I/O lines, 3 Timers/Counters, 12 Interrupts/2 priority levels, and 256 bytes on-chip RAM. 32kB ROM built-in. Its big pluses are: Dual Data Pointers (so that using two off-chip memory blocks is a lot faster), 1kB extra on-chip SRAM (appropriately latched for optimal `movx` access), and the fact that it is “4T”, i.e., many frequently used instructions takes 4 clock cycles to execute, so it is up to 3 times faster at the same nominal clock.

Dual Data Pointers is a common goodie in high-end 8051-like parts. By toggling a special flag bit, the programmer can switch between two possible `DPTR`’s, which means that the μC can quickly access two different look-up tables, among other things.

A.3 Random Number Generation and Other I/O Concerns on an 8051

For modern-day PKI applications, security concerns dictate that keys be generated on-card. The private key should stay there, never read by any human. The public key can be released off-card to a PC or other device when the appropriate commands are issued. A hardware random-number generator must be accessible on-card by the CPU for this purpose. If only signing is needed, then the RNG does not need to be all that fast, because with each signing action only around 8 bytes of entropy is needed. During key generation, it is better to have a faster RNG; it is feasible that a cryptography-grade software PRNG be used, seeded with a hardware randomness source. It would be slow, but not overly so. According to local vendors, in practice the *marginal cost of adding a hardware RNG is very low, essentially nil.*

Access to random number generation hardware is either implemented via an I/O port read or as special memory whence a random byte can always be read. Usually it is as easy as using a single `movx` instruction (takes about 4 instruction cycles including setup). In our tests, the sampling of a random byte is implemented as a separate “black-box” routine in assembly that returns a value after about 10 instruction cycles, because we wish to account for slower hardware RNG implementations.

For signing, it is assumed that only the hash value of a message needs to be passed into a smart card. For key generation, the smart card needs to be able to save the key to EEPROM (flash memory) or spitted out of the card in blocks of an appropriate size. In general, the smart card is wired up to transmit data only in blocks of 128 or 256 bytes (at most) at a time, and the transfer rate is about 9600 baud, which makes for an effective bandwidth of at most 1 kB/s.