KEY AGREEMENT, THE ALGEBRAIC ERASER^{TM}, AND LIGHTWEIGHT CRYPTOGRAPHY

Iris Anshel, Michael Anshel, Dorian Goldfeld, Stephane Lemieux

§1. Introduction:

Our purpose is to present a new key agreement protocol for public-key cryptography suitable for implementation on low-cost platforms which constrain the use of computational resources. In the process we introduce the concept of an Algebraic Eraser^{TM}, AE, and make a case that AE is a suitable primitive for use within lightweight cryptography. Our underlying motivation is the need to secure networks which deploy Radio Frequency Identification (RFID) tags used for identification, authentication, tracing and point-of-sale applications. The reader should consult [GJP] and [OSK] for further discussion.

The idea behind AE is to deny the cryptanalyst certain algebraic information inherent in many contemporary algebraic key agreement protocols employing group-theoretic transformations such as discrete exponention certain finite abelian groups or conjugation on certain infinite groups including braid groups (see [KM]). AE employs certain groups, homomorphisms, and actions of groups on monoids which to date force the cryptanalyst to primarily employ exhaustive search to determine the key. After careful formulation of the basic structure of AE we specify a general key agreement protocol based on the AE and go on to give some explicit examples including possible attacks and choice of secure parameters.

§2. The Algebraic Eraser^{TM} and its Associated Protocol:

The concept of the Algebraic Eraser emerges naturally when considering the following structures in tandem. Let M, N denote monoids and let S denote a group which acts on M on the left, and does not act on N. Given elements $s \in S$ and $m \in M$, we denote the result of s acting on m by ${}^{s}m$. The semidirect product of M and S, $M \rtimes S$ is defined to be the monoid whose underlying set is $M \times S$ and whose internal binary operation is given by

$$(m_1, s_1) \circ (m_2, s_2) \;=\; ({m_1}^{s_1} m_2,\; s_1 s_2).$$

The authors would like to thank SecureRF for its support of this research. The authors would also like to thank Alan Silvester for doing a lot of the C++ coding.

The direct product of N and S is denoted by $N \times S$.

The *algebraic eraser* $\mathbf{E}$ is the binary operation specified within the 6-tuple,

$$(M \rtimes S, N, \Pi, \mathbf{E}, A, B),$$

termed the $\mathbf{E}$–*structure*, where $M \rtimes S$ and N are as above, Π is (an easily computable) monoid homomorphism

$$\Pi : M \to N,$$

$\mathbf{E}$ is the function

$$\mathbf{E} : (N \times S) \times (M \rtimes S) \to N \times S$$

given by

$$\mathbf{E}\big((n, s), (m_1, s_1)\big) = \big(n\ \Pi({}^{s}m_1), ss_1\big),$$

and A, B are submonoids of $M \rtimes S$ such that for all $(a, s_a) \in A, (b, s_b) \in B$

$$\mathbf{E}\big((\Pi(a), s_a), (b, s_b)\big) = \mathbf{E}\big((\Pi(b), s_b), (a, s_a)\big). \tag{1}$$

The submonoids A and B, which satisfy (1) above, are termed $\mathbf{E}$–*Commuting*. For simplicity we will use the notation $\star$ as follows:

$$\mathbf{E}\big((n, s), (m_1, s_1)\big) \ = \ (n, s) \star (m_1, s_1).$$

The operation $\star$ satisfies the property that given $(n, s) \in N \times S$ and $(m_1, s_1), (m_2, s_2) \in M \rtimes S$ then

$$\Big((n, s) \star (m_1, s_1)\Big) \star (m_2, s_2) \ = \ (n, s) \star \Big((m_1, s_1) \circ (m_2, s_2)\Big). \tag{2}$$

The identity (2) is easily verified and allows one to compute $\star$ iteratively provided an element $(m, s) \in M \rtimes S$ is expressed as a product of generators.

The term *algebraic eraser* is a fitting description of the function $\mathbf{E}$ in that given an elements of $N \times S$,

$$(n, s), \qquad \mathbf{E}\big((n, s), (m_1, s_1)\big)$$

the element (m_1, s_1) cannot generally be recovered since the action of the element s on m_1 is not visible once the function Π has been applied to ${}^{s}m_1$ i.e., the action of s on m_1 has been effectively erased.

With the algebraic eraser $\mathbf{E}$ and its associated $\mathbf{E}$–*structure* specified we are in a position to introduce an associated key agreement protocol, $\mathbf{E}$–KAP. Referring to the protocol users as Alice and Bob, each user is assigned a submonoid of N, N_A and N_B respectively so that N_A and N_B commute. Furthermore Alice and Bob are assigned the $\mathbf{E}$–commuting submonoids A and B, respectively, which are determined by the fixed $\mathbf{E}$–structure. With

these assignments in place Alice and Bob choose their respective private keys which take the form

$$A_{\text{Private}} \;=\; \text{Alice's Private Key} \;=\; \Big(n_a, \langle (a_1, s_{a_1}), (a_2, s_{a_2}), \dots, (a_k, s_{a_k})\rangle\Big)$$

and

$$B_{\text{Private}} \;=\; \text{Bob's Private Key} \;=\; \Big(n_b, \langle (b_1, s_{b_1}), (b_1, s_{b_2}), \dots, (b_\ell, s_{b_\ell})\rangle\Big)$$

where $n_a \in N_A$, $n_b \in N_B$,

$$(a_1, s_{a_1}), (a_2, s_{a_2}), \dots, (a_k, s_{a_k}) \in A,$$

and

$$(b_1, s_{b_1}), (b_2, s_{b_2}), \dots, (b_\ell, s_{b_\ell}) \in B.$$

Having made these choices, Alice and Bob can then announce their respective public keys:

$$A_{\text{Public}} \;=\; \text{Alice's Public Key} \;=\; (\cdots((n_a, \text{id})\star(a_1, s_{a_1}))\star(a_2, s_{a_2}))\star\cdots)\star(a_k, s_{a_k}) \in N\times S,$$

$$B_{\text{Public}} \;=\; \text{Bob's Public Key} \;=\; (\cdots((n_b, \text{id})\star(b_1, s_{b_1}))\star(b_2, s_{b_2}))\star\cdots)\star(b_\ell, s_{b_\ell}) \in N\times S,$$

where id denoted the identity element in S. With this done Alice and Bob are now each in a position to compute the shared secret:

$$\begin{aligned}(\cdots((n_a, \text{id})\cdot B_{\text{Public}}\star(a_1, s_{a_1}))\star(a_2, s_{a_2}))\star\cdots)\star(a_k, s_{a_k}) \;=\;& \\ (\cdots((n_b, \text{id})\cdot A_{\text{Public}}\star(b_1, s_{b_1}))\star(b_2, s_{b_2}))\star\cdots)\star(b_\ell, s_{b_\ell}),&\end{aligned} \tag{3}$$

where $\cdot$ denoted multiplication in $N \times S$. The identity (3) holds because the submonoids A, B where chosen to $\mathbf{E}$–commute, and the submonoids N_A, N_B themselves commute.

§3. Algebraic Constructions

The $\mathbf{E}$–structure $(M \rtimes S, N, \Pi, \mathbf{E}, A, B)$ and its associate key agreement protocol lend themselves naturally to various natural algebraic/categorical constructions. Furthermore when we focus on the case of M being a group and S being a (sub)group of automorphisms of the group, a generalized commutator emerges from the $\mathbf{E}$–commuting condition.

The direct product of two $\mathbf{E}$–structures, $\mathbf{E}_1$ and $\mathbf{E}_2$ yield a third $\mathbf{E}$–structure:

$$\begin{aligned}(M_1\rtimes S_1, N_1, \Pi_1, \mathbf{E}_1, A_1, B_1)\times(M_2\rtimes S_2, N_2, \Pi_2, \mathbf{E}_2, A_2, B_2) \;=\;& \\ \Big((M_1\times M_2)\rtimes(S_1\times S_2), N_1\times N_2, \Pi_1\times\Pi_2, \mathbf{E}_1\times\mathbf{E}_2, A_1\times A_2, B_1\times B_2\Big).&\end{aligned}$$

Given a submonoid $H \leq M$ which is S invariant, there is a natural sub–**E**–structure of $(M \rtimes S, N, \Pi, \mathbf{E}, A, B)$ to consider:

$$(H \rtimes S, N, \Pi \downarrow_H, \mathbf{E} \downarrow_{(N\times S)\times(H\rtimes S)}, A \cap H, B \cap H).$$

Finally the concept of a image of an **E**–structure can be approached by starting with a homomorphism $\Psi : N \to N_0$ and considering the **E**–structure

$$(M \rtimes S, N_0, \Psi \circ \Pi, \mathbf{E}_0, A, B),$$

where $\Psi \circ \Pi$, denotes the composite of Ψ and Π.

In the case M is actually a group and the homomorphism Π is surjective, then another possible image can be defined. In this case we know that $N \cong M/K$ where $K \trianglelefteq M$. If $L \trianglelefteq M$ is a subgroup which is invariant under S then S acts on the group M/L and we can form $M/L \rtimes S$. This allows us to define an image of

$$(M \rtimes S, M/K, \big(M \to M/K\big), \mathbf{E}, A, B),$$

to be

$$\Big(\big(M/L \rtimes S\big), M/LK, \big(M/L \to M/LK\big), \mathbf{E}_L, \big(AL/L\big) \rtimes S, \big(BL/L\big) \rtimes S\Big)$$

where unspecified homomorphisms are simply the natural homomorphisms.

When we again restrict ourselves to the case of a group, G and we assume the group S is actually a group of automorphisms of G, $S \leq \ \mathrm{Aut}(G)$ then the hypothesis of **E**–commuting takes the following form. Elements in the subgroups A, B can be written as

$$(a, \alpha), \quad (b, \beta)$$

where $a, b \in G$ and $\alpha, \beta \in \ \mathrm{Aut}(G)$. The function Π can be assumed to take the form $G \to G/K$, and the identity (1) becomes

$$\big(a\ \alpha(b)\big)K, \alpha \circ \beta) \ = \ \big((b\ \beta(a)\big)K, \beta \circ \alpha).$$

The identity emerging from the first component leads naturally to the following generalization of the classical commutator. Given elements $x, y \in G$, and $(a, \alpha), (b, \beta) \in \ \mathrm{Aut}(G)$ define

$$C(\alpha, \beta, x, y) \ = \ x\ y\beta(x^{-1})\alpha(y^{-1}).$$

Clearly when $\alpha, \beta \ = \mathrm{id}$ we are reduced to the classical definition. Some analogues of the various classical commutator identities generalize as follows (and are left to the reader to verify). Let

$$\Omega(\alpha, \beta, x, y) \ = \ \alpha(x)\ y\ \beta(x)^{-1},$$

then we have

Proposition 1. *With the notation as above, the following identities hold:*

- $C(\alpha,\beta,x,y)^{-1} \;=\; C(\beta^{-1},\alpha^{-1},\alpha(y),\beta(x))$
- $C(\alpha,\beta,xy,z) \;=\; \Omega\big(\mathrm{id},\beta,x,C(\alpha,\beta,y,z)\big)\; C(\mathrm{id},\mathrm{id},\beta(x),\alpha(z))$
- $C(\alpha,\beta,x,yz) \;=\; C(\mathrm{id},\mathrm{id},x,y)\; \Omega\big(\mathrm{id},\alpha,y,C(\alpha,\beta,x,z)\big)$
- (identity of Hall–Witt type, see [MKS])

$$\begin{aligned} y^{-1}C\big(\mathrm{id},\alpha,&C(\alpha,\alpha,y,\alpha(x^{-1})),\alpha(z^{-1})\big)y \\ &\cdot z^{-1}C\big(\mathrm{id},\alpha,C(\alpha,\alpha,z,\alpha(y^{-1})),\alpha^2(x^{-1})\big)z \\ &\qquad\cdot \alpha(x^{-1})C\big(\mathrm{id},\alpha,C(\alpha,\alpha^2,\alpha(x),z^{-1}),\alpha(y^{-1})\big)\alpha(x) \end{aligned}$$

Before delving into the examples of our protocol we present a brief aside regarding a group theoretic authentication method. Recall the protocol introduced in [AAG1]: users Alice and Bob each choose private elements a, b in assigned subgroups A, B of a group G and in the end agree on the commutator $[a,b]$ known only to the users. In the course of the protocol Alice actually obtains the conjugate $b^{-1}ab$ and hence is in a position to compute the element

$$b^{-1}ab\cdot a\cdot[a,b] \;=\; b^{-1}a^2b \;=\; (b^{-1}ab)^2.$$

Assuming that extraction of roots, in particular square roots, is known to be a difficult problem, Alice can forward the element $b^{-1}a^2b$ to Bob who can then conjugate by the inverse of his private key b^{-1} to obtain the element a^2. Thus Alice has effectively transmitted the square of her private key a^2 to Bob over an open channel. With this done, any choice of a hash function $\mathcal{H}$ (i.e., a one–way collision–free function) generates an authentication protocol in the spirit of [D]:

(i) Alice chooses an element $r\in G$ and sends Bob the the element $c=\mathcal{H}(ra^2r^{-1})$,

(ii) Bob chooses a random bit d and sends d to Alice,

(iii) If $d=0$ Alice sends the element r and Bob verifies that $c=\mathcal{H}(ra^2r^{-1})$,

(iv) If $d=1$ Alice sends the conjugate $s=ra^2r^{-1}$ and Bob verifies that $c=\mathcal{H}(s^2)$.

§4. Examples of Key Agreement based on the Algebraic Eraser^{TM}:

Fix an integer $n\geq 7$ and a prime $p>n$. As an example of an algebraic eraser $\mathbf{E}$ whose associated key agreement protocol merits attention we begin by considering a subgroup

$$M\leq \mathrm{GL}(n,\mathbb{F}_p(t)),$$

where $t \;=\; (t_1,\ldots,t_n)$. Also, let $S \;=\; S_n$ be the symmetric group on n symbols. The group S acts on $\mathrm{GL}(n,\mathbb{F}_p(t))$ by permuting the variables $\{t_1,\ldots,t_n\}$, and we shall assume

that the action of S maps M to itself. Furthermore we assume that the semidirect product $M \rtimes S$ is finitely generated by some set of elements,

$$\big\{(x_1(t), s_1), \dots, (x_\lambda(t), s_\lambda)\big\}. \tag{4}$$

In this example, the monoid N is chosen to be

$$N \;=\; \mathrm{GL}(n, \mathbb{F}_p).$$

In order to define the homomorphism Π, we fix n elements in $\mathbb{F}_p$,

$$\tau_1, \tau_2, \dots, \tau_n \in \mathbb{F}_p,$$

and then evaluate

$$\Pi : M \to N$$

by setting

$$t_1 = \tau_1, \;\; t_2 = \tau_2, \qquad \dots, \qquad t_n = \tau_n.$$

A crucial assumption needs to be made at this point.

Assumption τ: *Let $\tau = (\tau_1, \tau_2, \dots, \tau_n)$. We assume that $x_i(\tau)$, $x_i(\tau)^{-1}$ are well defined for all $i = 1, 2, \dots, n$.*

There are, of course, many possible choices for the commuting submonoids N_A, N_B, which need to be chosen. One elementary choice for N_A and N_B is to first fix a matrix $m_0 \in GL(n, \mathbb{F}_p)$ of order $p^n - 1$. Then let $N_A = N_B$ be the submonoid of all matrices of the form

$$\ell_1 m_0^{k_1} + \ell_2 m_0^{k_2} + \;\; \cdots \;\; + \ell_r m_0^{k_r}, \tag{5}$$

where $\ell_1, \ell_2, \dots, \ell_r \in \mathbb{F}_p$ and $r, k_1, k_2, \dots, k_r \in \mathbb{Z}^+$. Each users private n_a and n_b are then elements of the above form (5). As to the subgroups $A, B \leq M \rtimes S$, which must **E**–commute for the protocol to succeed, one possibility is to proceed as follows. Fix an element $z \in M \rtimes S$ and choose two subsets of generators of $M \rtimes S$,

$$\Big\{\big(x_{a_1}(t), s_{a_1}\big), \dots, \big(x_{a_\mu}(t), s_{a_\mu}\big)\Big\}, \qquad \Big\{\big(x_{b_1}(t), s_{b_1}\big), \dots, \big(x_{b_\nu}(t), s_{b_\nu}\big)\Big\},$$

so that

$$x_{a_i}(t) \cdot x_{b_j}(t) \;=\; x_{b_j}(t) \cdot x_{a_i}(t), \qquad i = 1, \dots, \mu, \quad j = 1, \dots, \nu, \tag{6}$$

$$s_{a_i} t_{b_j} \;=\; t_{b_j} s_{a_i}, \qquad i = 1, \dots, \mu, \quad j = 1, \dots, \nu, \tag{7}$$

and

$$(8) \qquad {}^{s_{a_i}}x_{b_j}(t) \;=\; x_{b_j}(t), \quad {}^{s_{b_j}}x_{a_i}(t) \;=\; x_{a_i}(t), \quad i = 1, \ldots, \mu, \quad j = 1, \ldots, \nu.$$

Alice and Bob are then assigned the subgroups

$$(9) \qquad \begin{array}{c} z \cdot \Big\langle \big(x_{a_1}(t), s_{a_1}\big), \ldots, \big(x_{a_\mu}(t), s_{a_\mu}\big) \Big\rangle \cdot z^{-1}, \\ z \cdot \Big\langle \big(x_{b_1}(t), s_{b_1}\big), \ldots, \big(x_{b_\nu}(t), s_{b_\nu}\big) \Big\rangle \cdot z^{-1}, \end{array}$$

respectively, which will automatically **E**–commute with each other.

Hidden Elements Assumption: *We assume that the element $z \in M \rtimes S$ and the elements $x_{a_1}(t), \ldots, x_{a_\mu}(t),\ x_{b_1}(t), \ldots, x_{b_\nu}(t) \in M$ are secretly chosen and that it is difficult to determine these elements given that the conjugates (9) are publically announced.*

We are now in a position to summarize the above example of the Algebraic Eraser^{TM} key agreement protocol.

General Public Information: A subgroup M of the matrix group

$$N = GL\big(n, \mathbb{F}_p(t_1, \ldots, t_n)\big).$$

The symmetric group $S = S_n$ acting on the n variables $t_1, \ldots, t_n$ by permuting them. The subgroup M is chosen to be invariant under the action of S allowing for the formation of the semidirect product $M \rtimes S$.

Covert Information: A finite set of generators,

$$\Big\{\big(x_{a_1}(t), s_{a_1}\big), \ldots, \big(x_{a_\mu}(t), s_{a_\mu}\big)\Big\} \bigcup \Big\{\big(x_{b_1}(t), s_{b_1}\big), \ldots, \big(x_{b_\nu}(t), s_{b_\nu}\big)\Big\} \subseteq \Big\{(x_1(t), s_1), \ldots, (x_\lambda(t), s_\lambda)\Big\},$$

of $M \rtimes S$ satisfying (6), (7), (8), and the hidden elements assumption. An element $z \in M \rtimes S$ satisfying the hidden elements assumption.

Public Information: *An integer $n \geq 7$. A prime number $p > n$. The* **E***–commuting subgroups*

$$A = z \cdot \Big\langle \big(x_{a_1}(t), s_{a_1}\big), \ldots, \big(x_{a_\mu}(t), s_{a_\mu}\big) \Big\rangle \cdot z^{-1},$$

$$B = z \cdot \Big\langle \big(x_{b_1}(t), s_{b_1}\big), \ldots, \big(x_{b_\nu}(t), s_{b_\nu}\big) \Big\rangle \cdot z^{-1},$$

where z is the hidden conjugating element and the $x_i(t)$ are the hidden subgroup generators. The homomorphism $\Pi : M \to N$ satisfying Assumption τ. The operation $\star$ satisfying (2). A fixed matrix $m_0 \in N$.

Alice's Private Key: *A matrix of the form* $n_a = \ell_1 m_0^{\alpha_1} + \ell_2 m_0^{\alpha_2} + \ \cdots \ + \ell_r m_0^{\alpha_r}$, *where* $\ell_1, \ldots, \ell_r \in \mathbb{F}_p$ *and* $r, \alpha_1, \ldots, \alpha_r \in \mathbb{Z}^+$ *are secret.*
A subset of generators $\left\{ \left(x_{a_{i_1}}(t), s_{a_{i_1}}\right), \ldots, \left(x_{a_{i_\mu}}(t), s_{a_{i_\mu}}\right) \right\}$ *of* A.

Alice's Public Key:

$$A_{\text{Public}} \ = \ \Big(\big(\cdots \big(\big(n_a, id\big) \star z \big) \star \big(x_{a_{i_1}}(t), s_{a_{i_1}}\big) \star \cdots \big) \star \big(x_{a_{i_\mu}}(t), s_{a_{i_\mu}}\big) \Big) \star z^{-1}$$

Bob's Private Key: *A matrix of the form* $n_b = \ell'_1 m_0^{\beta_1} + \ell'_2 m_0^{\beta_2} + \ \cdots \ + \ell'_{r'} m_0^{\beta_{r'}}$, *where* $\ell'_1, \ldots, \ell'_{r'} \in \mathbb{F}_p$ *and* $r', \beta_1, \ldots, \beta_{r'} \in \mathbb{Z}^+$ *are secret.*
A subset of generators $\left\{ \left(x_{b_{j_1}}(t), s_{b_{j_1}}\right), \ldots, \left(x_{b_{j_\nu}}(t), s_{b_{j_\nu}}\right) \right\}$ *of* B.

Bob's Public Key:

$$B_{\text{Public}} \ = \ \Big(\big(\cdots \big(\big(n_b, id\big) \star z \big) \star \big(x_{b_{j_1}}(t), s_{b_{j_1}}\big) \star \cdots \big) \star \big(x_{b_{j_\nu}}(t), s_{b_{j_\nu}}\big) \Big) \star z^{-1}$$

Shared Secret:

$$\begin{aligned} &\Big(\cdots \big((n_a, id) \cdot B_{\text{Public}} \star z \big) \star \big(x_{a_{i_1}}(t), s_{a_{i_1}}\big) \big) \star \cdots \big) \star \big(x_{a_{i_\mu}}(t), s_{a_{i_\mu}}\big) \Big) \star z^{-1} \\ &\qquad = \Big(\cdots \big((n_b, id) \cdot A_{\text{Public}} \star z \big) \star \big(x_{b_{j_1}}(t), s_{b_{j_1}}\big) \big) \star \cdots \big) \star \big(x_{b_{j_\nu}}(t), s_{b_{j_\nu}}\big) \Big) \star z^{-1}. \end{aligned}$$

In order to analyze the cryptographic applicability of the above algorithm, we shall make the following simplifying assumptions and definition.

- $i_\mu = j_\nu = g =$ *the number of generators in Alice and Bob's private keys.*
- $\lambda \leq n^2$ *where* λ *is equal to the number of generators of* $M \rtimes S$.

It is now possible to compute the size (in bits) of the public and private keys that occur in the Algebraic Eraser^{TM} Key Agreement Protocol. First of all, Alice and Bob's public keys will simply be a pair consisting of an $n \times n$ matrix with coefficients in the finite field $\mathbb{F}_p$ and an element of the permutation group S_n. Each entry in this matrix will have at most $\log_2(p)$ bits. It follows that the matrix component of the public key will have bit size equal to $n^2 \log_2(p)$. The permuation can be specified by a list of n numbers where each number is between 1 and n. Thus the bit size of the permutation is at most $n \log_2(n) \leq n \log_2(p)$. Consequently, the size of the public key is at most $(n^2 + n) \log_2(p)$. The private key also has two saperate components. First, the high power of the fixed matrix m_0 can be represented with at most $n^2 \log_2(p)$ bits. Secondly, each generator can be specified with at most $\log_2(\lambda) \leq 2 \log_2(n)$ bits. It follows that the size of the private key is at most $n^2 \log_2(p) + 2 \log_2(n) g$. We record these observations in the following proposition.

Proposition 2. *In the Algebraic Eraser^{TM} protocol specified above, the bit-size of the private key is at most*

$$n^2 \log_2(p) + 2\log_2(n)g, \tag{10}$$

while the bit-size of the public key is at most

$$(n^2 + n)\log_2(p). \tag{11}$$

Next, we examine the running time of the algorithm and show that it is essentially linear in the number of generators g in the private key. We shall obtain a crude estimate of the running time in terms of elementary processor operation. By an elementary processor operation we mean either a search and replace operation or a multiplication/addition/subtraction/involving two bits. It is convenient to make the simplifying assumption that each matrix $x_k(t)$ $(k = 1, 2, \ldots, \lambda)$ occurring in the generator $(x_k(t), s_k)$ differs from the identity matrix in at most ℓ entries and that each of these entries is a Laurent polynomial in $\mathbb{F}_p(t)$ where the Laurent polynomial itself has at most ρ terms of degree at most d. For example, the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & t_1 - 2t_2^{-1} - t_2 + t_2^3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 + 2t_1 & 0 & 0 & 2 \end{pmatrix}$$

differs from the identity matrix in exactly 3 entries and each of these entries involves Laurent polynomials of at most 4 terms of degree at most 3. The degree is defined to be the absolute value of the largest power, i.e., t_2^{-4} has degree 4. Given an element

$$(x(t), s) \in M \rtimes S$$

and a generator

$$(x_j(t), s_j)$$

of $M \rtimes S$, the most expensive and time consuming operation of the protocol is the computation of

$$(x(t), s) \star (x_j(t), s_j) = \Big(x(t) \cdot \Pi\left({}^s x_j(t)\right),\ ss_j\Big).$$

First of all, the multiplication of permutations ss_j can be done in n search and replace operations, so this is clearly linear in the number of generators g. Second, the computation of ${}^s x_j(t)$ requires at most $\ell\rho$ search and replace operations. The computation of $\Pi\left({}^s x_j(t)\right)$ requires an additional $\ell\rho$ search and replace operations followed by at most $\ell\rho d$ computations in $\mathbb{F}_p$. Finally, the computation of $x(t) \cdot \Pi\left({}^s x_j(t)\right)$ involves at most $n\ell$ multiplications and additions in $\mathbb{F}_p$. This gives an upper bound of $n + 2\ell\rho + 2\ell\rho d(\log_2 p)^2 + 2n\ell(\log_2 p)^2$ elementary

operations for each of the g generators. In the final step of the key agreement protocol, it is necessary to multiply two $n \times n$ matrices over $\mathbb{F}_p$. This will take $n^3(\log_2 p)^2$ operations. Assuming that the conjugating element z is made up of g_z generators, the total estimate for the running time of the algorithm is:

$$n^3(\log_2 p)^2 \ + \ (g + 2g_z) \cdot \Big(n + 2\ell\rho + 2\ell\rho d(\log_2 p)^2 + 2n\ell(\log_2 p)^2\Big). \tag{12}$$

One may also give estimates for the memory size (fixed and rewriteable) needed to run the protocol.

§5. The Colored Burau Key Agreement Protocol (CBKAP):

Fix an integer $n \geq 7$, and let $t = (t_1, \ldots, t_n)$. Define

$$x_1(t) = \begin{pmatrix} -t_1 & 1 & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix},$$

and for $i = 2, \ldots n-1$, let

$$x_i(t) := \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & t_i & -t_i & 1 & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix}.$$

which is the identity matrix except for the i^{th} row where it has successive entries $t_i, -t_i, 1$ with $-t_i$ on the diagonal. For each $i = 1, 2, \ldots, n-1$, we define

$$s_i \ = \ (i \ \ i+1)$$

which is just the transposition (element of the symmetric group S_n) which interchanges i and $i+1$. The elements $\big(x_i(t), s_i\big)$, for $i = 1, 2, \ldots, n-1$, satisfy the braid relations and hence determine a representation of the braid group (see [AAG2]). Next, fix a prime $p > n$, then the set of pairs

$$\Big\{\big(x_1(t), s_1\big), \ldots, \big(x_{n-1}(t), s_{n-1}\big)\Big\}$$

will generate the semidirect product $M \rtimes S$ with $S = S_n$ and $M \subset GL(n, \mathbb{F}_p)$. We call the group $M \rtimes S$ the colored Burau group. The general key agreement protocol given in §4, with this choice of M, is termed the colored Burau key agreement protocol (CBKAP). If we

choose $\tau = (\tau_1, \tau_2, \dots, \tau_n)$ with $1 \leq \tau_i < p$ for $1 \leq i \leq n$ then one may easily check that Assumption τ of §4 is satisfied.

In order to implement the CBKAP with the above choice of M it is necessary to effectively choose the matrix m_0, the elements $z \in M \rtimes S$, and

$$x_{a_1}(t), \dots, x_{a_\mu(t)},\ x_{b_1}(t), \dots, x_{b_\nu}(t) \in M.$$

With regard to the matrix m_0 one can begin by generating a random matrix from $GL(N, \mathbb{F}_p)$ and test to see if this matrix has an irreducible characteristic polynomial over $\mathbb{F}_p$. If it does not we simply choose another random matrix and repeat the process. Appendix A contains a Mathematica program that performs this task which heuristically runs quickly and is always successful. The resulting matrix m_0 then has an easily calculable multiplicative order because m_0 is diagonalizable over $\mathbb{F}_p$. The non-zero entries of the diagonal matrix will lie in $\mathbb{F}_{p^n}$ and be the roots of the characteristic polynomial. With pobability better than $1/2$, each of these roots will have order $p^n - 1$, and so the matrix m_0 will likewise have order $p^n - 1$. If the roots have a lower order, we again discard and choose a new m_0. Eventually a suitible m_0 will be found.

We now turn to the task of choosing the elements $z \in M \rtimes S$ and

$$x_{a_1}(t), \dots, x_{a_\mu}(t),\ x_{b_1}(t), \dots, x_{b_\nu}(t)\ \in\ M.$$

Assuming that we do not want either party to be able to obtain the other's key, a trusted third party (TTP) will be performing the algorithm. If one wishes to design a system which allows for a "master key" then the TTP would simply be one of the users who would then be in possession of the "master key."

The TTP performs the following actions to establish two commuting sets of generators in the braid group. By the representation described above, this produces two **E**–commuting sets of generators in the colored Burau group. Note that these two sets can then be made public and used by any two parties that wish to establish, secretly, a common key. Thus the TTP need only be called upon once.

Let $B_n = \{b_1, \dots, b_{n-1}\}$ be the Artin representation of the braid group on n strings. Recall that the left canonical form of a braid word may be written as a power of the fundamental braid times a sequence of short braid words, called permutation braids. For further details see [B]. To further shorten the lengths of keys, any even power of the fundamental braid can be ommitted since it is a central element. For the same reason, any odd power of the fundamental braid can simply be replaced by the fundamental braid itself. This will considerably shorten the sequences of integers representing keys.

TTP Algorithm:

(1) *Choose two secret subsets* $BL = \{b_{\ell_1}, \dots b_{\ell_\alpha}\},\ BR = \{b_{r_1}, \dots b_{r_\beta}\}$ *of the set of generators of* B_n*, where* $|\ell_i - r_j| \geq 2$ *for all* $1 \leq i \leq \ell_\alpha$ *and* $1 \leq j \leq r_\beta$.

(2) *Choose a secret element* $z \in B_n$.

(3) *Choose words* $\{w_1, \ldots, w_\gamma\}$ *of bounded length from BL.*

(4) *Choose words* $\{v_1, \ldots, v_\gamma\}$ *of bounded length from BR.*

(5) *For* $1 \leq i \leq \gamma$*:*

- **(a)** *calculate the left normal form of* zw_iz^{-1} *and reduce the result modulo the square of the fundamental braid;*
- **(b)** *set* w_i' *equal to the sequence of integers that corresponds to the element calculated in (a);*
- **(c)** *calculate the left normal form of* zv_iz^{-1} *and reduce the result modulo the square of the fundamental braid;*
- **(d)** *set* v_i' *equal to the sequence of integers that corresponds to the element calculated in (c).*

(5) *Publish the two sets* $\{w_1', \ldots, w_\gamma'\}$ *and* $\{v_1', \ldots, v_\gamma'\}$.

§6. Linear Algebraic Attack on CBKAP:

There is a successful linear algebraic attack on CBKAP if the conjugating element z is known. We assume, for simplicity, that n is even. There is a similar attack if n is odd. Suppose that the matrix m_0, the element z, and the user public keys are given by

$$(m_0^\alpha \cdot \Pi(z) \cdot \Pi({}^{s_z}A) \cdot \Pi({}^{s_z s_A} z^{-1}),\ s_z\, s_A s_{z^{-1}}),$$

and

$$(m_0^\beta \cdot \Pi(z) \cdot \Pi({}^{s_z}B) \cdot \Pi({}^{s_z s_B} z^{-1}),\ s_z\, s_B s_{z^{-1}}).$$

Apriori, the matrix $\Pi({}^{s_z}A)$ takes the form $\begin{pmatrix} X & 0 \\ 0 & I \end{pmatrix}$, where X is an element of $GL(n/2, \mathbb{F}_p)$ and I is the identity matrix in $GL(n/2, \mathbb{F}_p)$. Similarly $\Pi({}^{s_z}B)$ takes the form $\begin{pmatrix} I & 0 \\ 0 & Y \end{pmatrix}$. Note that, in general, the condition that A, B E-commute require that $\Pi({}^{s_z}A)$ and $\Pi({}^{s_z}B)$ should be commuting matrices which differ from the identity in disjoint blocks. We can always bring them to the form specified above by conjugating by a suitable permutation matrix.

The attack that emerges does not derive the users' secret keys, but only the agreed upon key. The attacker, Eve, begins by diagonalizing the matrix m_0,

$$Qm_0Q^{-1} \;=\; \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}.$$

Despite the fact that Eve does not know α, she can assert that

$$m_0^\alpha \;=\; Q^{-1} \begin{pmatrix} \lambda_1^\alpha & & \\ & \ddots & \\ & & \lambda_n^\alpha \end{pmatrix} Q,$$

and hence Alice's public key actually takes the form

$$Q^{-1}\begin{pmatrix}\lambda_1^\alpha & & \\ & \ddots & \\ & & \lambda_n^\alpha\end{pmatrix} Q\cdot\Pi(z)\begin{pmatrix}X & 0\\ 0 & I\end{pmatrix}\Pi({}^{s_z s_A}z^{-1}). \tag{13}$$

Since the element z is known Eve can compute $\Pi(z)$, s_z, $s_z s_A$, and then $\Pi({}^{s_z s_A}z^{-1})$. Multiplying (13) appropriately on the left and right Eve obtains

$$\begin{pmatrix}\lambda_1^\alpha & & \\ & \ddots & \\ & & \lambda_n^\alpha\end{pmatrix} Q\cdot\Pi(z)\begin{pmatrix}X & 0\\ 0 & I\end{pmatrix}.$$

Eve may set $Q\cdot\Pi(z)=((c_{ij}))$, where c_{ij} are known for all $1\leq i\leq n,\ 1\leq j\leq n$. What remains is simply

$$\begin{pmatrix}\lambda_1^\alpha & & \\ & \ddots & \\ & & \lambda_n^\alpha\end{pmatrix}\cdot((c_{ij}))\cdot\begin{pmatrix}X & 0\\ 0 & I\end{pmatrix}.$$

The powers λ_j^α (with $j>\frac{n}{2}$) of the eigenvalues are thus visible. Similarly, one may switch the first half of the columns of Q with the second half of the columns (by conjugating everything by an appropriate permutation matrix)which then allows Eve to obtain m_0^α.

At this point Eve can recover $\Pi({}^{s_z}A)$ and the matrix $\Pi({}^{s_z s_B s_A}z^{-1})$. The shared key can be obtained from Bob's public key by multiplying on the left and right by known elements: m_0^α on the left, and $(\Pi({}^{s_z s_B}z^{-1})^{-1}$, $\Pi({}^{s_z}A)$, and $\Pi({}^{s_z s_B s_A}z^{-1})$ on the right (recall that we assumed $\Pi({}^{s_z s_B}A)\ =\ \Pi({}^{s_z}A)$).

The attack just described was implemented in C^{++}. Assuming the conjugating element z in CBKAP is known, the shared secret can be found from the public keys in 0.04 seconds, regardless of the length of the second private keys chosen by Alice and Bob. Part of the C^{++} code is included in Appendix B. Special thanks to Alan Silvester for producing this code.

§7. Security Analysis of CBKAP:

The linear algebraic attack discussed in §6 cannot be implemented unless the conjugating element z is correctly guessed. This implies that performing an exhaustive search for z can yield Alice's and Bob's shared secret. Let g_z denote the number of Artin generators of z. To ensure m bits of security against this attack we need

$$g_z\cdot\frac{\ln(2n-2)}{\ln(2)}\ \geq\ m. \tag{14}$$

Since the advent of braid group cryptography in [AAG1], various algorithms have emerged for determining an element z in the Artin braid group B_n provided the set of elements

$$a_1, \ \ldots \ , a_\ell, \ z^{-1}a_1 z, \ \ldots \ , z^{-1}a_\ell z \ \in \ B_n$$

is publicly known. This is called the simultaneous conjugacy search problem. We shall examine several such algorithms and show that they cannot successfully generate an attack on the CBKAP.

In [G], a probabilistic approach to the conjugacy search problem in Garside groups is obtained. His algorithm yields a linear solution to the conjugacy search problem in braid groups which almost always works (except on a set of measure zero).

Recall that in the TTP algorithm, the elements

$$w_i' = z w_i z^{-1}, \ \ v_i' = z v_j z^{-1}, \qquad (1 \leq i \leq \ell_\alpha, \ 1 \leq j \leq \ell_\beta)$$

are publicly announced, but $z, w_i, v_j, (1 \leq i \leq \ell_\alpha, \ 1 \leq j \leq \ell_\beta)$ are kept secret. All that we really know is that all the w_i commute with all the v_j, for $1 \leq i \leq \ell_\alpha, \ 1 \leq j \leq \ell_\beta$. The difficulty in using this attack on CBKAP to recover z, is that although each w_i and v_i is conjugate to w_i' and v_i' respectively, modulo the square of the fundamental braid, w_i and v_i are secrets known only to the TTP. Thus an attacker would have to guess each w_i and v_i correctly and then determine the braid element that conjugates the w_i's and v_i's to the w_i''s and v_i''s. The length of each w_i and v_i could easily be set long enough to prevent such an attack but even this is unnecessary for several reasons.

First there will be many elements conjugate to w_1', even of the same length as w_1 if this length was correctly guessed by the attacker. For example all cyclic conjugates of w_1 will generally have the same length as w_1. Second the length of w_1 is only known to be less than a given bound which increases the number of elements conjugate to w_1' which are not w_1, i.e the number of false positives. Third, w_i' was reduced modulo the square of the fundamental braid, further increasing the number of false positives. Finally, each false positive will yield a corresponding false positive for each of the other w_i''s and v_i''s.

The reason for the false positives is that the TTP conjugates each by z and then removes the highest power of Δ^2 that he can. In general, after conjugating, you will get much higher powers of Δ^2 in the normal form. Even more importantly, the permutation braids resuting will be all different. For example, if the attacker: chooses words of length 10, then removes powers of Δ^2, and then tries to solve the conjugacy search problem, it will fail to give a usable z.

Length based attacks for the conjugacy search problem were introduced in [HT]. The algorithm has been further studied and developed in [GKTTV], and [D].

The length attacks work as follows. Let $x, y \in B_n$ such that x and y are known to be conjugate in B_n and y is much longer than x. For each $c \in B_n$ of length 10, if the length of $c^{-1}yc$ is significantly less than that of y then to some probability we know that c is the first 10 generators of an element that conjugates x to y. The attacker then repeats the algorithm with $c^{-1}yc$ in place of x.

This attack is ineffective against CBKAP again because the w_i's and v_i's are not known. Further, since w_i' is actually equal to $zw_iz^{-1} \times \Delta^{2r}$, where r is some integer and Δ is the fundamental braid, the actual length of the conjugate element is unknown to all but the TTP. Also, the short length of z as will be determined in the next section, implies that checking all c of length 10, amounts essentially to brute force search for z with only a probabilistic verification of success. Finally if the algorithm is attempted using shorter c's then heuristics indicate that even choosing the correct c produces $c^{-1}yc$ of length equal to or even greater than that of y.

§8. Parameter Choices and Running Times:

The intended application of CBKAP is for constrained devices where only lightweight security is possible. With this, in mind, we will choose parameters to ensure 60, 80, and 120 bits of security, i.e., that, respectively, 2^{60}, 2^{80}, or 2^{120} attempts are needed by any of the above attacks in order to compromise CBKAP. It should be noted however, that CBKAP could be scaled to achieve higher levels of security. We will show the derivation of parameters only for 80 bits of security, the other cases being analogous. The results for all three security levels with a small selection of parameter choices are reported in the tables below. Although it is not the focus of this paper, we note that CBKAP will run at least 10 times faster (but probably significantly more) when implemented in a symmetric platform.

The most immediate and straightforward attack on CBKAP is to perform an exhaustive search of all possible choices of Alice and Bob's private keys. This attack, as one might expect, is easy to overcome by choosing large enough parameters. Recall that Alice's public key is the eraser product of a linear combination, over $\mathbb{F}_p$, of r powers of the publicly known matrix, m_0 and the elements of her second private key which we assume has length g. We know m_0 has order $p^n - 1$ so on average $\frac{1}{2}(p^n - 1)^r$ powers will need to be checked before finding the correct one. Likewise the keyspace for Alice's second private key is all words of length g in generators and their inverses that the TTP gives her. Say there are T of these. Then the total number of searches for her public key will be $\frac{1}{2}(p^n - 1)^r$ for the first private key and T^g for the second private key. Thus for m bits of security, we need $\frac{r \cdot n \cdot \ln(p)}{\ln(2)} \geq m$, and

$$g \cdot \frac{\ln(T)}{\ln(2)} \geq m. \tag{15}$$

Note that the inequality (15) is critical for without it in place the system is not secure.We will show in the next section that we can choose relatively small parameters to meet our desired standards of security.

The case of $n = 14$ serves as a good illustrative example. Thus CBKAP will run, in this instance, with elements from B_{14} and over matrices of dimension 14. Recall that g_z denotes the length of the element z in the Artin generators. To provide 80 bits of security against

an exhaustive search for z, the inequality, as in (14), asserts that we must have

$$g_z \frac{\ln(2n-2)}{\ln(2)} \geq m$$

implies that $g_z \geq 80\frac{\ln(2)}{\ln(26)} \approx 17$, and, hence, z must be chosen to be of length 17. If we choose the length of the w_i's and v_i's to be bounded by 10, then the conjugate elements, reduced modulo Δ^2 appear to have length around 188 generators. We label the average length of the w_i's and v_j's by L. Once the length g of Alice's and Bob's second private keys are set, and the prime p is chosen, the number of bit operations required for either of them to generate the shared secret using their private keys and the other's public key is approximately

$$\left(7 \cdot 14 \cdot L \cdot g + 14^2\right) \log_2(p).$$

If the TTP produces 27 generators each for Alice and Bob then including their inverses we have 54 generators in all. Against exhaustive search for either Alice's or Bob's second private keys, we need to ensure, as in (15), that

$$g \cdot \frac{\ln(54)}{\ln(2)} \geq 80.$$

The smallest value possible, satisfying this inequality, is $g = 14$. Likewise, to ensure 80 bits of security against a brute force attack on the first private key, we can choose $p = 13$, and $r = 3$. The chart below summarizes these results and lists the approximate number bit operations needed to produce a shared secret, for the different security levels and the different average lengths of the w_i's and v_j's.

Security Analysis for CBKAP over B_{14}

Bits of Security	Lengths of w_i, v_j	Length of 2^{nd} Private Key	Bit Ops to make shared secret
60	188	11	816144
80	188	14	1037232
120	572	21	4714192

To decrease the number of bit ops, the system can be scaled down further. We next consider $n = 12$. Thus CBKAP will run, in this instance, with elements from B_{12} and over matrices of dimension 14. To provide 80 bits of security against an exhaustive search for z, the inequality

$$g_z \frac{\ln(2n-2)}{\ln(2)} \geq m$$

implies that $g_z \geq 80\frac{\ln(2)}{\ln(22)} \approx 18$ so z will need to be of length 18. If we choose the length of the w_i's and v_j's to be bounded by 10, then the conjugate elements, reduced modulo Δ^2 appear to have length around 130 generators. We label the average length of the w_i's and v_i's by L. Once the length g of Alice's and Bob's second private keys are set, and the prime p is chosen, the number of bit operations required for either of them to generate the shared secret using their private keys and the other's public key is approximately

$$\left(7 \cdot 12 \cdot L \cdot g + 12^2\right) \log_2 p.$$

If the TTP produces 27 generators each for Alice and Bob then including their inverses we have 54 generators. Against exhaustive search for either Alice's or Bob's second private keys, we need to ensure that

$$g \cdot \frac{\ln(54)}{\ln(2)} \geq 80.$$

The smallest value possible, satisfying this inequality, is $g = 14$. Likewise, to ensure 80 bits of security against a brute force attack on the first private key, we can choose $p = 13$, and $r = 3$. The chart below summarizes these results and lists the approximate number bit operations needed to produce a shared secret, for the different security levels and the different average lengths of the w_i's and v_j's.

Security Analysis for CBKAP over B_{12}

Bits of Security	Lengths of w_i, v_j	Length of 2^{nd} Private Key	Bit Ops to make shared secret
60	130	11	484512
80	188	14	615552
120	572	21	9121312

§9. Hardware Implementations:

The CBKAP may be employed in the RFID space. An initial application would be authentication for a local area network consisting of passive RFID tags and readers. Passive RFID tags are tags with no battery power that draw energy from a reader. We are interested in providing authentication between tag and reader. The CBKAP algorithm allows the RFID tag and a reader to establish a common secret key which can then be used for mutual authentication by standard cryptographic techniques which we do not focus on. We also note that the readers in the local area network typically have heavy weight computational resources and do not require lightweight cryptographic algorithms. Also, the TTP algorithm can be performed offline or by the readers.

We now discuss the part of the CBKAP algorithm which needs to be performed by the passive RFID tag. Consider CBKAP over B_{12} with a security level of 2^{60} and parameters as specified in section 8. For references regarding VLSI signal processing (see [DR]) and for references regarding EPC$^{\text{TM}}$ Radio-Frequency Identity Protocols Class (see [EPC]).

The practical deployment of the Algebraic Eraser (AE) within an extremely constrained device such as a passive RFID tag is primarily governed by the twin concerns of tag economics and tag performance. At the time of writing these economic constraints limit the total cost of implementation of the AE to less than 0.5 cents. Similarly the performance constraints impose a run time limit of under 20ms if the AE is not to interfere with normal tag access rates provided for by the Electronic Product Code (EPC) protocol or conflict with FCC dwell time regulations. Current CMOS semiconductor industry norms of $1000 per wafer thus limit the available die size for the AE to 163,000 um^2.

One approach is to build a custom Algebraic Eraser processing engine (AEPE) directly in hardware such that the datapaths and dedicated finite field processing elements (Multiplies/Adds) are arranged to compute just enough information on every clock cycle. The AEPE consists of the following four components: A non-volatile memory (NVM) to contain the tag private keys, A permutation engine (SE), A matrix multiplication engine (ME) and control functions (C) to schedule the AE operations. We seek an implementation that balances both sets of constraints

We are now in a position to analyze the following example where $p = 13$ and the AEPE is required to perform 2,200 rounds of a permutation followed by a 12×12 matrix multiplication running at a typical clock rate of 1 MHz.

The matrix multiplier takes advantage of the sparse form of the right hand matrix to reduce the number of computations per row to two finite field multiplications and additions. Furthermore, the choice of $p = 13$ limits all of the operands to 4 bits or less in width. Thus a complete matrix multiplication only requires a total of 24 multiplications and additions. By allowing 5 clock cycles for the matrix multiplication (to give a total run time of 11 ms) we can construct a systolic array multiplier architecture using bit-serial signal processing to reduce all of the datapaths from 4 bits down to 1 bit wide and use the same gates to perform both the multiplication and the addition (the control scheduler allocates 4 clock cycles to the multiplication and 1 to the addition).

The permutation engine SE can be implemented using a Benes switch which, for a bit-serial non-blocking $12 \rightarrow 12$ permutation requires less than 56 1-bit wide crossovers or 112 gates.

Having established that we can use 1 bit wide datapaths we complete our gate count and area analysis to determine that we need 5,374 logic gates and 1200 bit of NVM. Based on industry norms for 0.13um CMOS of 242,000 gates/mm2 these functions can be built in circa 22,000 um^2 for the logic and 120,000 um^2 for the NVM.

The gate counts associated with this bit serial architecture are:

Multiplication/Addition = $(24 \times 4 \times 18) = 1728$ gates
Addition = 0 gates (now part of the multiplier).
Inline storage for temporary results = $144 \times (4 \times 4 + 1) = 2448$ gates

AEPE Scheduler < 1200 gates.

Finally we mention that the creation of security standards for the RFID market is at an early stage of development [PKK]

References

[AAG1] Anshel, Iris, Anshel, Michael, Goldfeld, Dorian, *An algebraic method for public-key cryptography*, Math. Res. Lett. 6 (1999), no. 3-4, 287–291.

[AAG2] Anshel, Iris, Anshel, Michael, Goldfeld, Dorian, *A Linear time matrix key agreement protocol over small finite fields*, to appear.

[B] Birman, Joan S. *Braids, links, and mapping class groups*, Annals of Mathematics Studies, No. 82. Princeton University Press, Princeton, N.J.; University of Tokyo Press, Tokyo, 1974.

[D] Dehornoy, Patrick, *Braid-based cryptography.* Group theory, statistics, and cryptography, 5–33, Contemp. Math., 360, Amer. Math. Soc., Providence, RI, 2004.

[DR] *VLSI Signal Processing - A Bit Serial Approach,* Denyer & Renshaw, (1985), Addison-Wesley, 0-201-14404-2

[EPC] *EPC*$^{\text{TM}}$ *Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 - 960 MHz,* Version 1.0.9, EPCGlobal Inc, (2004).

[GKTTV] D. Garber, S. Kaplan, M. Teicher, B. Tsaban, U. Vishne, *Length-based conjugacy search in the Braid group* , math. GR0209267)

[GJP] S.L. Garfinkel, A. Juels and R. Pappu, *RFID Privacy: An Overview of Solutions and Problems,* IEEE Security and Privacy Volume 3 Number 3 (May/June 2005), 34–43.

[G] V. Gebhardt, *A new approach to the conjugacy problem in Garside groups*, (English. English summary) J. Algebra 292 (2005), no. 1, 282–302.

[HT] J. Hughes, A. Tannenbaum, *Length-based attacks for certain group based encryption rewriting systems,* 2000 (Published in SECI 02)).

[KM] N. Koblitz and A.J. Menenzes, *A Survey of Public-Key Cryptosystems,* SIAM Review, Volume 44, Number 4 (December 2005), 599–634.

[MKS] Magnus, Wilhelm, Karrass, Abraham, Solitar, Donald *Combinatorial group theory, Presentations of groups in terms of generators and relations*, Reprint of the 1976 second edition. Dover Publications, Inc., Mineola, NY, 2004.

[OSK] M. Ohkubo, K. Suzuki, and S. Kinoshita,*RFID Privacy Issues and Technical Challenges,* Communications of the ACM. Volume 48, Number 9 (2005), 66–71.

[PKK] T. Phillips, T. Karygiannis, T. Kuhn, *Security Standards for the RFID Market*, IEEE Security & Privacy Volume 3 Number 6 (November/December 2005), 85-89.

Iris Anshel, 31 Peter Lynas Ct. Tenafly, NJ 07670, USA

Michael Anshel, City College of New York, New York, NY 10031, USA

Dorian Goldfeld, Columbia University, Department of Mathematics, New York, NY 10027

Stephane Lemieux, University of Calgary, Department of Mathematics, Calgary, Canada

Appendix A

```
Generate a matrix of high order in Maple
By: Stephane Lemieux
Date: August 2005
```

```
> interface(rtablesize):=13;
```

$$interface\,(rtablesize) := 13$$

This Maple program generates a 12 by 12 matrix with irreducible characteristic polynomial over F_p. Such matrices usually have order $p^{12} - 1$.

```
> matgen:=proc(p::posint)
>    local tag, m, i, j, r, f;
>    tag := 0;
>    while tag = 0 do
>       m:=Matrix(12,12):
>       r:=rand(p):
>       for i from 1 to 12 do
>          for j from 1 to 12 do
>             m[i,j]:=r();
>          od;
>       od;
>       f := x -> Det(m - x*Matrix(12,12,shape=identity))
>                 mod p;
>       if Irreduc(f) mod p then return m; fi;
>    od:
> end proc:

> matgen(13);
```

$$\begin{bmatrix} 2 & 5 & 6 & 8 & 6 & 11 & 4 & 7 & 6 & 11 & 0 & 11 \\ 8 & 11 & 2 & 7 & 6 & 7 & 8 & 0 & 4 & 5 & 4 & 4 \\ 3 & 7 & 0 & 12 & 5 & 5 & 10 & 2 & 9 & 5 & 5 & 1 \\ 2 & 5 & 12 & 5 & 0 & 3 & 6 & 0 & 9 & 3 & 9 & 10 \\ 2 & 3 & 11 & 7 & 9 & 8 & 6 & 3 & 1 & 12 & 3 & 5 \\ 6 & 9 & 0 & 10 & 5 & 6 & 11 & 3 & 0 & 9 & 7 & 5 \\ 0 & 9 & 11 & 12 & 8 & 12 & 7 & 1 & 3 & 2 & 4 & 3 \\ 12 & 8 & 0 & 3 & 5 & 12 & 0 & 11 & 0 & 2 & 2 & 0 \\ 12 & 0 & 0 & 5 & 6 & 8 & 10 & 5 & 7 & 5 & 2 & 7 \\ 4 & 6 & 12 & 9 & 6 & 10 & 5 & 4 & 9 & 6 & 7 & 0 \\ 0 & 6 & 11 & 12 & 9 & 0 & 7 & 7 & 11 & 9 & 5 & 2 \\ 12 & 4 & 4 & 10 & 9 & 5 & 5 & 7 & 9 & 2 & 6 & 5 \end{bmatrix}$$

```
> matgen(997);
```

$$\begin{bmatrix} 795 & 20 & 774 & 716 & 581 & 13 & 57 & 611 & 231 & 275 & 448 & 637 \\ 83 & 480 & 556 & 14 & 927 & 157 & 378 & 968 & 514 & 763 & 165 & 476 \\ 631 & 669 & 457 & 748 & 620 & 23 & 854 & 635 & 906 & 28 & 595 & 869 \\ 809 & 867 & 517 & 273 & 270 & 81 & 303 & 649 & 768 & 875 & 256 & 446 \\ 309 & 840 & 893 & 824 & 50 & 91 & 515 & 436 & 713 & 656 & 658 & 182 \\ 209 & 687 & 123 & 97 & 371 & 43 & 351 & 599 & 508 & 799 & 140 & 686 \\ 333 & 74 & 298 & 956 & 728 & 47 & 303 & 24 & 294 & 443 & 477 & 195 \\ 224 & 333 & 123 & 638 & 428 & 447 & 129 & 712 & 581 & 368 & 728 & 127 \\ 605 & 50 & 549 & 164 & 725 & 515 & 974 & 511 & 530 & 70 & 519 & 500 \\ 855 & 786 & 856 & 607 & 159 & 522 & 409 & 541 & 773 & 241 & 670 & 767 \\ 766 & 437 & 838 & 695 & 542 & 729 & 149 & 213 & 338 & 497 & 673 & 200 \\ 953 & 691 & 447 & 407 & 410 & 266 & 100 & 120 & 566 & 222 & 79 & 316 \end{bmatrix}$$

```
> m:=matgen(13);
```

$$m := \begin{bmatrix} 6 & 0 & 0 & 1 & 1 & 0 & 2 & 1 & 6 & 3 & 9 & 0 \\ 6 & 0 & 4 & 7 & 12 & 2 & 7 & 2 & 2 & 11 & 8 & 6 \\ 2 & 2 & 8 & 7 & 8 & 9 & 3 & 9 & 3 & 3 & 9 & 10 \\ 1 & 0 & 12 & 8 & 0 & 0 & 8 & 7 & 7 & 1 & 10 & 4 \\ 6 & 5 & 12 & 2 & 4 & 7 & 0 & 8 & 0 & 8 & 0 & 12 \\ 6 & 6 & 5 & 2 & 3 & 5 & 1 & 10 & 0 & 2 & 3 & 1 \\ 4 & 2 & 3 & 3 & 12 & 12 & 1 & 5 & 6 & 0 & 1 & 12 \\ 12 & 3 & 5 & 7 & 12 & 3 & 6 & 3 & 5 & 2 & 7 & 10 \\ 8 & 5 & 2 & 1 & 11 & 7 & 2 & 6 & 5 & 9 & 5 & 8 \\ 5 & 4 & 0 & 6 & 12 & 1 & 9 & 3 & 12 & 7 & 11 & 0 \\ 0 & 3 & 4 & 6 & 4 & 6 & 7 & 9 & 9 & 0 & 0 & 10 \\ 12 & 11 & 9 & 7 & 10 & 8 & 4 & 9 & 2 & 0 & 2 & 5 \end{bmatrix}$$

```
> Det(m) mod 13;
```

$$0$$

```
> x:=matgen(5);
```

$$x := \begin{bmatrix} 3 & 4 & 2 & 0 & 0 & 4 & 0 & 3 & 1 & 0 & 0 & 3 \\ 4 & 0 & 1 & 2 & 2 & 2 & 1 & 3 & 0 & 3 & 1 & 3 \\ 3 & 3 & 0 & 1 & 4 & 0 & 0 & 2 & 2 & 2 & 2 & 3 \\ 4 & 0 & 0 & 3 & 1 & 2 & 0 & 1 & 1 & 2 & 3 & 3 \\ 2 & 4 & 0 & 3 & 1 & 1 & 0 & 1 & 1 & 2 & 2 & 3 \\ 1 & 4 & 3 & 0 & 4 & 4 & 4 & 3 & 4 & 2 & 3 & 0 \\ 3 & 0 & 4 & 3 & 4 & 3 & 4 & 0 & 0 & 3 & 1 & 2 \\ 4 & 1 & 4 & 2 & 4 & 1 & 0 & 2 & 2 & 3 & 0 & 1 \\ 1 & 1 & 4 & 1 & 4 & 0 & 2 & 2 & 0 & 4 & 2 & 1 \\ 0 & 4 & 0 & 0 & 1 & 3 & 2 & 4 & 3 & 3 & 0 & 4 \\ 2 & 2 & 1 & 1 & 4 & 3 & 1 & 4 & 2 & 0 & 2 & 1 \\ 3 & 2 & 0 & 2 & 3 & 2 & 3 & 3 & 4 & 1 & 0 & 0 \end{bmatrix}$$

```
> Det(x) mod 5;
```

1

Appendix B

```
/*
 *  Program: attack2
 *  Created: 2005-09-20 Alan Silvester
 *                                                                    ||
 *  Implements the attack
 */

#include <iostream>
#include <sstream>
#include <fstream>
#include <string>

#include <NTL/mat_ZZ_p.h>

#include "assert.h"
#include "timer.h"
#include <sys/time.h>

#include "functions.h"

#include "field_functions.h"

#include "AAG_System.h"

NTL_CLIENT

using namespace std;
using namespace NTL;

std::ostream & operator<<(std::ostream &os,
                          const GeneratorIndex &gi)
{
   gi.print(os);
}

Blob AAGSpace::u_compute (const ValueMatrix& seed) const

{
   Blob x (seed, PermutationVector (m_matrix_size, true));
   size_t i;


   for (i = 0; i < m_u.size (); ++i)
```

```
   {

      x = eraser (x, *(get_generator (m_u[i])));

   }


   return (x);

}


Blob AAGSpace::u_inv_compute (const ValueMatrix& seed,
                              const PermutationVector& p1) const
{
   Blob x (seed, p1);

   size_t i;


   for (i = 0; i < m_u_inverse.size (); ++i)

   {

      x = eraser (x, *(get_generator (m_u_inverse[i])));

   }


   return (x);

}

int main (int argc, char * argv[])
{
   ////////////////////////////////////////////////////////////
   //
   // Opening output log file
   //
   ////////////////////////////////////////////////////////////

   ofstream output ("outputfile3.txt");
   if (!output.is_open())
   {
      cout << "error opening outputfile3.txt\n";
      return 1;
```

```
   }

   // Seed RNG
// timeval tv;
// gettimeofday(&tv, NULL);
// srand(tv.tv_sec);

   ////////////////////////////////////////////////////////////
   //
   // Writing temp ini file
   //
   ////////////////////////////////////////////////////////////

   int kl = 100;

   ofstream inioutput ("test5.ini");
   if (!inioutput.is_open())
   {
      cout << "error opening test5.ini\n";
      return 1;
   }

// inioutput << "RandomSeed=" << tv.tv_sec << endl;
   inioutput << "DisplaySharedSecret=no\n";

   inioutput << "Iterations=10\n";

   inioutput << "Prime=13\n";

   inioutput << "VC=12\n";

   inioutput << "T=1D,1 2 8 9 10 11 12 3 4 5 6 7\n";

   inioutput << "KeyLength=" << kl << endl;

   inioutput << "Partition=1D,1 2 3 4 5\n";
   inioutput << "Partition=1D,7 8 9 10 11\n";

   inioutput << "ULength=12\n";
   inioutput << "U=1,12\n";
   inioutput << "U=7,-12\n";
   inioutput << "U=12,-12\n";
   inioutput << "U=8,12\n";
   inioutput << "U=9,-12\n";
   inioutput << "U=11,12\n";
   inioutput << "U=4,-12\n";
```

```
inioutput << "U=5,-12\n";
inioutput << "U=6,12\n";
inioutput << "U=2,-12\n";
inioutput << "U=3,12\n";
inioutput << "U=10,-12\n";
inioutput << "U=1,-12\n";
inioutput << "U=7,12\n";
inioutput << "U=12,12\n";
inioutput << "U=8,-12\n";
inioutput << "U=9,12\n";
inioutput << "U=11,-12\n";
inioutput << "U=4,12\n";
inioutput << "U=5,12\n";
inioutput << "U=6,-12\n";
inioutput << "U=2,12\n";
inioutput << "U=3,-12\n";
inioutput << "U=10,12\n";

inioutput.close();

////////////////////////////////////////////////////////////
//
// Initialisation
//
////////////////////////////////////////////////////////////

// Setup the generator matricies
vector<pair<Generator, Generator> > g;
size_t matrix_size = 6;
timer t;
cout << "setting up generator matricies\n";

cout << "making new AAGSpace\n"; ;
AAGSpace space = AAGSpace("test5.ini");

cout << "have " << space.m_partitions.size()
     << " partitions\n"; ;

cout << "m_partition[0] data: ";
for (int i = 0; i < space.m_partitions[0].size(); i++)
   cout << space.m_partitions[0][i] << " ";
cout << endl; ;

cout << "m_partition[1] data: ";
for (int i = 0; i < space.m_partitions[1].size(); i++)
   cout << space.m_partitions[1][i] << " ";
```

```
cout << endl; ;

ValueMatrix vm_seed_matrix = ValueMatrix(matrix_size*2,
                                         true);

////////////////////////////////////////////////////////////
//
// Start timer for alice key-gen
//
////////////////////////////////////////////////////////////

t.start();

////////////////////////////////////////////////////////////
//
// Generate Alice's keypair
//
////////////////////////////////////////////////////////////

cout << "generating alice's keypair\n";

// This skips the exponentiation of the seed matrix
Keypair alice_key = space.make_keypair(0, 0);

cout << "keygen time: " << t << endl;
Blob alice_key_blob = alice_key.build_public();
alice_key.print("Keypair::print - alice's keypair", cout);

////////////////////////////////////////////////////////////
//
// Start timer for bob key-gen
//
////////////////////////////////////////////////////////////

t.restart();

////////////////////////////////////////////////////////////
//
// Generate Bob's keypair
//
////////////////////////////////////////////////////////////

cout << "generating bob's keypair\n";

// This skips the exponentiation of the seed matrix
Keypair bob_key = space.make_keypair(1, 0);
```

```
cout << "keygen time: " << t << endl;
Blob bob_key_blob = bob_key.build_public();
bob_key.print("Keypair::print - bob's keypair", cout);

//////////////////////////////////////////////////////////
//
// Start timer for key exchange
//
//////////////////////////////////////////////////////////

t.restart();

//////////////////////////////////////////////////////////
//
// Shared secret
//
//////////////////////////////////////////////////////////

Blob shared_secret = space.key_exchange(bob_key_blob,
                                alice_key.PrivateKey(),
                                vm_seed_matrix);
cout << "secret gen time: " << t << endl;
shared_secret.print("shared secret (alice)", cout);

space.m_t.print("t = ");

cout << "Key exchange time: " << t << endl;

//////////////////////////////////////////////////////////
//
// Start timer for crack
//
//////////////////////////////////////////////////////////

t.restart();

//////////////////////////////////////////////////////////
//
// Make identity matrix
//
//////////////////////////////////////////////////////////

ValueMatrix vm = ValueMatrix(matrix_size*2, true);

//////////////////////////////////////////////////////////
```

```
//
// Calculate pi(u)
//
//////////////////////////////////////////////////////////////

cout << "Computing pi(u)\n";
Blob pi_u = space.u_compute(vm);
pi_u.print("pi(u), u.p", cout);

//////////////////////////////////////////////////////////////
//
// Calculate P1 (alice)
//
//////////////////////////////////////////////////////////////

// Dummy data
PermutationVector alice_p1 = alice_key_blob.m_permutation;

alice_key_blob.m_permutation.Permute(pi_u.m_permutation,
                                     alice_p1);

cout << "\nP1 (alice)\n";
alice_p1.print(cout);
cout << endl;

//////////////////////////////////////////////////////////////
//
// Calculate P1 (bob)
//
//////////////////////////////////////////////////////////////

// Dummy data
PermutationVector bob_p1 = bob_key_blob.m_permutation;
bob_key_blob.m_permutation.Permute(pi_u.m_permutation,
                                   bob_p1);

cout << "\nP1 (bob)\n";
bob_p1.print(cout);
cout << endl;

//////////////////////////////////////////////////////////////
//
// Calculate pi(u inv) (alice)
//
//////////////////////////////////////////////////////////////
```

```
cout << "Computing pi(u inv to sig u sig a)\n";
Blob alice_pi_u_inv = space.u_inv_compute(vm, alice_p1);
alice_pi_u_inv.print("pi(u_inv_to_sig_u_sig_a), alice_p1",
                     cout);

////////////////////////////////////////////////////////////
//
// Calculate pi(u inv) (bob)
//
////////////////////////////////////////////////////////////

cout << "Computing pi(u inv to sig u sig b)\n";
Blob bob_pi_u_inv = space.u_inv_compute(vm, bob_p1);
bob_pi_u_inv.print("pi(u_inv_to_sig_u_sig_b), bob_p1",
                   cout);

////////////////////////////////////////////////////////////
//
// Calculate pi(u inv)
//
////////////////////////////////////////////////////////////

cout << "Computing pi(u inv)\n";
Blob pi_u_inv = space.u_inv_compute(vm,
                    PermutationVector(matrix_size*2, true));
pi_u_inv.print("pi(u inv), sig_u_inv", cout);

////////////////////////////////////////////////////////////
//
// Calculate P2 (alice)
//
////////////////////////////////////////////////////////////

// Dummy data
PermutationVector alice_p2 = alice_key_blob.m_permutation;

pi_u_inv.m_permutation.Permute(alice_p1, alice_p2);

cout << "\nP2 (alice)\n";
alice_p2.print(cout);
cout << endl;

////////////////////////////////////////////////////////////
//
// Calculate P2 (bob)
//
```

```
//////////////////////////////////////////////////////////////

// Dummy data
PermutationVector bob_p2 = bob_key_blob.m_permutation;

pi_u_inv.m_permutation.Permute(bob_p1, bob_p2);

cout << "\nP2 (bob)\n";
bob_p2.print(cout);
cout << endl;

//////////////////////////////////////////////////////////////
//
// Compute pi(u)^(-1)
//
//////////////////////////////////////////////////////////////

// Init ZZ_p class
ZZ p;
p = (long) 13;
ZZ_p::init(p);

// Determinant holder (don't really need)
ZZ_p d1;

// Our matrix and matrix holder
mat_ZZ_p m1;
mat_ZZ_p X;
m1.SetDims(matrix_size*2, matrix_size*2);
X.SetDims(matrix_size*2, matrix_size*2);

// Fill matrix with data
for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      m1[i][j] = pi_u.m_matrix.cell(i,j);

// Calculate inverse
NTL::inv(d1, X, m1);

// Copy data back out
ValueMatrix inverse_pi_u (matrix_size*2, false);

for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      for (int k = 0; k < rep(X[i][j]).size(); k++)
         inverse_pi_u.cell(i, j, inverse_pi_u.cell(i,j)
```

```
                                + digit(rep(X[i][j]), k));

cout << "inv of pi(u)\n";
inverse_pi_u.print(cout);

////////////////////////////////////////////////////////////
//
// Compute pi(u^p1)^(-1) (alice)
//
////////////////////////////////////////////////////////////

Blob alice_m2_blob = space.make_m2(vm, alice_p1,
                                   matrix_size);

// Determinant holder (don't really need)
ZZ_p alice_d2;

// Our matrix and matrix holder
mat_ZZ_p alice_m2;
mat_ZZ_p alice_Y;
alice_m2.SetDims(matrix_size*2, matrix_size*2);
alice_Y.SetDims(matrix_size*2, matrix_size*2);

// Fill matrix with data
for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      alice_m2[i][j] = alice_m2_blob.m_matrix.cell(i,j);

// Calculate inverse
NTL::inv(alice_d2, alice_Y, alice_m2);

// Copy data back out
ValueMatrix alice_m2_inverse (matrix_size*2, false);

for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      for (int k = 0; k < rep(alice_Y[i][j]).size(); k++)
         alice_m2_inverse.cell(i, j,
                               alice_m2_inverse.cell(i,j)
                               + digit(rep(alice_Y[i][j]), k));

cout << "inv of pi(u_inv_to_sig_u_sig_a)\n";
alice_m2_inverse.print(cout);

////////////////////////////////////////////////////////////
//
```

```
// Compute pi(u^p1)^(-1) (bob)
//
////////////////////////////////////////////////////////////

Blob bob_m2_blob = space.make_m2(vm, bob_p1, matrix_size);

// Determinant holder (don't really need)
ZZ_p bob_d2;

// Our matrix and matrix holder
mat_ZZ_p bob_m2;
mat_ZZ_p bob_Y;
bob_m2.SetDims(matrix_size*2, matrix_size*2);
bob_Y.SetDims(matrix_size*2, matrix_size*2);

// Fill matrix with data
for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      bob_m2[i][j] = bob_m2_blob.m_matrix.cell(i,j);

// Calculate inverse
NTL::inv(bob_d2, bob_Y, bob_m2);

// Copy data back out
ValueMatrix bob_m2_inverse (matrix_size*2, false);

for (int i = 0; i < matrix_size*2; i++)
   for (int j = 0; j < matrix_size*2; j++)
      for (int k = 0; k < rep(bob_Y[i][j]).size(); k++)
         bob_m2_inverse.cell(i, j, bob_m2_inverse.cell(i,j)
                           + digit(rep(bob_Y[i][j]), k));

cout << "inv of pi(u_inv_to_sig_u_sig_b)\n";
bob_m2_inverse.print(cout);

////////////////////////////////////////////////////////////
//
// Recover key
//
////////////////////////////////////////////////////////////

ValueMatrix alice_temp = (inverse_pi_u
                        * alice_key_blob.m_matrix)
                        * alice_m2_inverse;
cout << "\nShould be upper-block (alice)\n";
alice_temp.print(cout);
```

```
ValueMatrix bob_temp = (inverse_pi_u
                        * bob_key_blob.m_matrix)
                        * bob_m2_inverse;
cout << "\nShould be lower-block (bob)\n";
bob_temp.print(cout);

// Compose sigma_a, sigma_b

// Dummy data
PermutationVector result1 = alice_p1;
PermutationVector result2 = alice_p1;

alice_p1.Permute(bob_p2, result1);
result1.Permute(pi_u_inv.m_permutation, result2);

cout << "\nalice_p1\n";
alice_p1.print(cout);
cout << "\nalice_p2\n";
alice_p2.print(cout);
cout << "\nbob_p1\n";
bob_p1.print(cout);
cout << "\nbob_p2\n";
bob_p2.print(cout);
cout << "\nresult 1\n";
result1.print(cout);
cout << "\nresult 2\n";
result2.print(cout);

Blob rhs = space.u_inv_compute(vm, result1);

ValueMatrix found_secret = pi_u.m_matrix * alice_temp
                           * bob_temp * rhs.m_matrix;
cout << "\n\nsecret is\n";
found_secret.print(cout);
result2.print(cout);

//////////////////////////////////////////////////////////
//
// Stop timer and print
//
//////////////////////////////////////////////////////////

t.stop();
cout << "\n\nRun-time: " << t << endl;
```

```
    ////////////////////////////////////////////////////////////
    //
    // Cleanup and exit
    //
    ////////////////////////////////////////////////////////////

    return 0;
}
```