[2] The Vector

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

- I have a secret: the midterm exam.
- I've represented it as an n-vector $\mathbf{v}$ over $GF(2)$.
- I want to provide it to my TAs Alice and Bob (A and B) so they can administer the midterm while I take vacation.
- One TA might be bribed by a student into giving out the exam ahead of time, so I don't want to simply provide each TA with the exam.
- **Idea:** Provide pieces to the TAs:
  - the two TAs can jointly reconstruct the secret, but
  - neither of the TAs all alone gains any information whatsoever.
- **Here's how:**
  - I choose a random n-vector $\mathbf{v}_A$ over $GF(2)$ randomly according to the uniform distribution.
  - I then compute

$$\mathbf{v}_B := \mathbf{v} - \mathbf{v}_A$$

  - I provide Alice with $\mathbf{v}_A$ and Bob with $\mathbf{v}_B$, and I leave for vacation.

Vectors over $GF(2)$: All-or-nothing secret-sharing using $GF(2)$

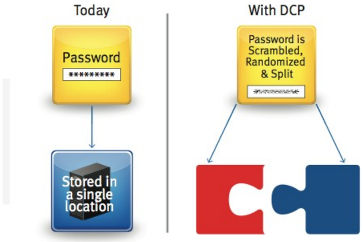
- What can Alice learn without Bob?
- All she receives is a random n-vector.
- What about Bob?
- He receives the output of $f(\mathbf{x}) = \mathbf{v} - \mathbf{x}$ where the input is random and uniform.
- Since $f(\mathbf{x})$ is invertible, the output is also random and uniform.

Vectors over GF (2): All-or-nothing secret-sharing using GF (2)

Too simple to be useful, right?
RSA just introduced a product based on this idea:

RSA® DISTRIBUTED
CREDENTIAL PROTECTION

Scramble, randomize and split credentials



- Split each password into two parts.
- Store the two parts on two separate servers.

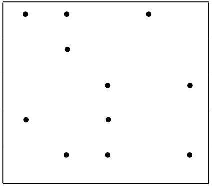
Vectors over *GF* (2): *Lights Out*

- *input*: Configuration of lights
- *output*: Which buttons to press in order to turn off all lights?

Computational Problem: Solve an instance of *Lights Out*

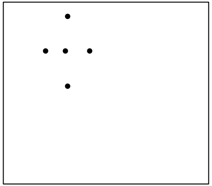
Represent state using `range(5)`$\times$`range(5)`-vector over *GF* (2).

Example state vector:



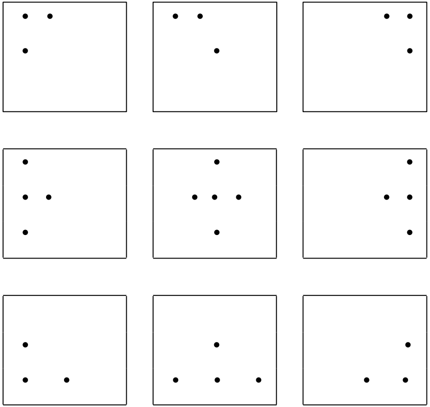
Represent each button as a vector (with ones in positions that the button toggles)

Example button vector:



Vectors over $GF(2)$: *Lights Out*

Button vectors for 3×3:



Computational Problem: Which sequence of button vectors sum to $\mathbf{s}$?

Vectors over *GF* (2): *Lights Out*

Computational Problem: Which sequence of button vectors sum to **s**?

Observations:

- By commutative property of vector addition, order doesn't matter.
- A button vector occuring twice cancels out.

Replace Computational Problem with: Which *set* of button vectors sum to **s**?

Vectors over *GF* (2): *Lights Out*

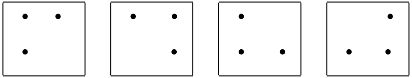
Replace our original Computational Problem with a more general one:

$\boxed{\text{Solve an instance of } \textit{Lights Out}}$ $\Rightarrow$ $\boxed{\text{Which set of button vectors sum to } \mathbf{s}\text{?}}$

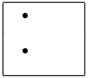
$\Rightarrow$ $\boxed{\begin{array}{l}\text{Find subset of } GF(2) \text{ vectors} \\ \mathbf{v}_1, \ldots, \mathbf{v}_n \text{ whose sum equals } \mathbf{s}\end{array}}$

Vectors over *GF* (2): *LightsOut*

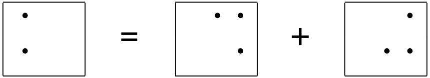
Button vectors for 2×2 version:



where the black dots represent ones.

Quiz: Find the subset of the button vectors whose sum is 

Answer:



Dot-product

Dot-product of two D-vectors is sum of product of corresponding entries:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{k \in D} \mathbf{u}[k]\,\mathbf{v}[k]$$

Example: For traditional vectors $\mathbf{u} = [u_1, \ldots, u_n]$ and $\mathbf{v} = [v_1, \ldots, v_n]$,

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Output is a scalar, not a vector
Dot-product sometimes called *scalar product*.

Dot-product

Example: Dot-product of $[1, 1, 1, 1, 1]$ and $[10, 20, 0, 40, -100]$:

$$
\begin{array}{ccccccccccc}
 & 1 & & 1 & & 1 & & 1 & & 1 & & \\
\cdot & 10 & & 20 & & 0 & & 40 & & -100 & & \\
\hline
 & 10 & + & 20 & + & 0 & + & 40 & + & (-100) & = & -30
\end{array}
$$

Quiz: Dot-product

Quiz: Write a procedure `list_dot(u, v)` with the following spec:

- *input:* equal-length lists `u` and `v` of field elements
- *output:* the dot-product of `u` and `v` interpreted as vectors

Hint: Use the `sum(·)` procedure together with a list comprehension.

Answer:

```
def list_dot(u, v): return sum([u[i]*v[i] for i in range(len(u))])
```

or

```
def list_dot(u, v): return sum([a*b for (a,b) in zip(u,v)])
```

Dot-product: Total cost or benefit

Suppose D consists of four main ingredients of beer:

$$D = \{\text{malt, hops, yeast, water}\}$$

A *cost* vector maps each food to a price per unit amount:

$$cost = \{\text{hops} : \$2.50/ounce,\ \text{malt} : \$1.50/pound,\ \text{water} : \$0.06/gallon,\ \text{yeast} : \$.45/g\}$$

A *quantity* vector maps each food to an amount (e.g. measured in pounds).
quantity = {hops:6 oz, malt:14 pounds, water:7 gallons, yeast:11 grams}

The total cost is the dot-product of *cost* with *quantity*:

$$cost \cdot quantity = \$2.50 \cdot 6 + \$1.50 \cdot 14 + \$0.006 \cdot 7 + \$0.45 \cdot 11 = \$40.992$$

A *value* vector maps each food to its caloric content per pound:

$$value = \{\text{hops} : 0,\ \text{malt} : 960,\ \text{water} : 0,\ \text{yeast} : 3.25\}$$

The total calories represented by a pint of beer is the dot-product of *value* with *quantity*:

Dot-product: Linear equations

Example: A sensor node consist of hardware components, e.g.

- CPU
- radio
- temperature sensor
- memory

Battery-driven and remotely located so we care about energy usage.

Suppose we know the power consumption for each hardware component.
Represent it as a D-vector with $D = \{radio, sensor, memory, CPU\}$

$$\textbf{rate} = \text{Vec}(D, \{memory : 0.06\text{W}, radio : 0.06\text{W}, sensor : 0.004\text{W}, CPU : 0.0025\text{W}\})$$

Have a test period during which we know how long each component was working.
Represent as another D vector:

$$\textbf{duration} = \text{Vec}(D, \{memory : 1.0\text{s}, radio : 0.2\text{s}, sensor : 0.5\text{s}, CPU : 1.0\text{s}\})$$

Total energy consumed (in Joules):

Dot-product: Linear equations

Turns out: We can only measure *total energy consumed by the sensor node* over a period

Goal: calculate rate of energy consumption of each hardware component.

Challenge: Cannot simply turn on memory without turning on CPU.

Idea:

- Run several tests on sensor node in which we measure total energy consumption
- In each test period, we know the duration each hardware component is turned on. For example,

$$\textbf{duration}_1 = \{radio : 0.2\text{s}, sensor : 0.5\text{s}, memory : 1.0\text{s}, CPU : 1.0\text{s}\}$$

$$\textbf{duration}_2 = \{radio : 0\text{s}, sensor : 0.1\text{s}, memory : 0.2\text{s}, CPU : 0.5\text{s}\}$$

$$\textbf{duration}_3 = \{radio : .4\text{s}, sensor : 0\text{s}, memory : 0.2\text{s}, CPU : 1.0\text{s}\}$$

- In each test period, we know the total energy consumed: $\beta_1 = 1, \beta_2 = 0.75, \beta_3 = .6$
- Use data to calculate current for each hardware component.

Dot-product: Linear equations

A *linear equation* is an equation of the form

$$\mathbf{a} \cdot \mathbf{x} = \beta$$

where **a** is a vector, β is a scalar, and **x** is a vector of variables.

In sensor-node problem, we have linear equations of the form

$$\mathbf{duration}_i \cdot \mathbf{rate} = \beta_i$$

where **rate** is a vector of variables.

Questions:

- Can we find numbers for the entries of **rate** such that the equations hold?
- If we do, does this guarantee that we have correctly calculated the current draw for each component?

Dot-product: Linear equations

More general questions:

- Is there an algorithm for solving a *system of linear equations*?

$$\begin{aligned}
\mathbf{a}_1 \cdot \mathbf{x} &= \beta_1 \\
\mathbf{a}_2 \cdot \mathbf{x} &= \beta_2 \\
&\vdots \\
\mathbf{a}_m \cdot \mathbf{x} &= \beta_m
\end{aligned}$$

- How can we know whether there is only one solution?

- What if our data are slightly inaccurate?

These questions motivate much of what is coming in future weeks.

Dot-product: Measuring similarity: Comparing voting records

Can use dot-product to measure similarity between vectors.

Upcoming lab:

- Represent each senator's voting record as a vector:

$$[+1, +1, 0, -1]$$

+1 = *In favor*, 0 = *not voting*, -1 = *against*

- Dot-product $[+1, +1, 0, -1] \cdot [-1, -1, -1, +1]$
    - very positive if the two senators tend to agree,
    - very negative if two voting records tend to disagree.

Dot-product: Vectors over $GF(2)$

Consider the dot-product of 11111 and 10101:

$$\begin{array}{ccccccccccc} & 1 & & 1 & & 1 & & 1 & & 1 & & \\ \cdot & 1 & & 0 & & 1 & & 0 & & 1 & & \\ \hline & 1 & + & 0 & + & 1 & + & 0 & + & 1 & = & 1 \end{array}$$

$$\begin{array}{ccccccccccc} & 1 & & 1 & & 1 & & 1 & & 1 & & \\ \cdot & 0 & & 0 & & 1 & & 0 & & 1 & & \\ \hline & 0 & + & 0 & + & 1 & + & 0 & + & 1 & = & 0 \end{array}$$

Dot-product: Simple authentication scheme

- Usual way of logging into a computer with a password is subject to hacking by an eavesdropper.

- **Alternative:** Challenge-response system
  - Computer asks a question about the password.
  - Human sends the answer.
  - Repeat a few times before human is considered authenticated.

  Potentially safe against an eavesdropper since probably next time will involve different questions.

- Simple challenge-response scheme based on dot-product of vectors over $GF(2)$:
  - Password is an n-vector $\hat{\mathbf{x}}$.
  - Computer sends random n-vector $\mathbf{a}$
  - Human sends back $\mathbf{a} \cdot \hat{\mathbf{x}}$.

Dot-product: Simple authentication scheme

- **Example:** Password is $\hat{\mathbf{x}} = 10111$.
- Computer sends $\mathbf{a}_1 = 01011$ to Human.
- Human computes dot-product $\mathbf{a}_1 \cdot \hat{\mathbf{x}}$:

$$\begin{array}{rccccccccccc} & 0 & & 1 & & 0 & & 1 & & 1 & & \\ \cdot & 1 & & 0 & & 1 & & 1 & & 1 & & \\ \hline & 0 & + & 0 & + & 0 & + & 1 & + & 1 & = & 0 \end{array}$$

and sends $\beta_1 = 0$ to Computer.

Dot-product: Attacking simple authentication scheme

How can an eavesdropper Eve cheat?

- She observes a sequence of challenge vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ and the corresponding response bits $\beta_1, \beta_2, \ldots, \beta_m$.
- Can she find the password?

She knows the password must satisfy the linear equations

$$\begin{aligned} \mathbf{a}_1 \cdot \mathbf{x} &= \beta_1 \\ \mathbf{a}_2 \cdot \mathbf{x} &= \beta_2 \\ &\vdots \\ \mathbf{a}_m \cdot \mathbf{x} &= \beta_m \end{aligned}$$

Questions:

- How many solutions?
- How to compute them?

Answers will come later.

Dot-product: Attacking simple authentication scheme

Another way to cheat?

Can Eve derive a challenge for which she knows the response?

Algebraic properties of dot-product:

- **Commutativity:** $\mathbf{v} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{v}$
- **Homogeneity:** $(\alpha\, \mathbf{u}) \cdot \mathbf{v} = \alpha\, (\mathbf{u} \cdot \mathbf{v})$
- **Distributive law:** $(\mathbf{v}_1 + \mathbf{v}_2) \cdot \mathbf{x} = \mathbf{v}_1 \cdot \mathbf{x} + \mathbf{v}_2 \cdot \mathbf{x}$

Example: Eve observes

- challenge 01011, response 0
- challenge 11110, response 1

$$\begin{aligned}(01011 + 11110) \cdot \mathbf{x} &= 01011 \cdot \mathbf{x} + 11110 \cdot \mathbf{x}\\ &= \quad 0 \quad + \quad 1\\ &= \quad 1\end{aligned}$$

For challenge 01011 + 11110, Eve can derive right response.

Dot-product: Attacking simple authentication scheme

More generally, if a vector satisfies equations

$$\begin{aligned}
\mathbf{a}_1 \cdot \mathbf{x} &= \beta_1 \\
\mathbf{a}_2 \cdot \mathbf{x} &= \beta_2 \\
&\vdots \\
\mathbf{a}_m \cdot \mathbf{x} &= \beta_m
\end{aligned}$$

then what other equations does the vector satisfy?
Answer will come later.

Solving a triangular system of linear equations

How to find solution to this linear system?

$$
\begin{aligned}
[1, 0.5, -2, 4] \cdot \mathbf{x} &= -8 \\
[0, 3, 3, 2] \cdot \mathbf{x} &= 3 \\
[0, 0, 1, 5] \cdot \mathbf{x} &= -4 \\
[0, 0, 0, 2] \cdot \mathbf{x} &= 6
\end{aligned}
$$

Write $\mathbf{x} = [x_1, x_2, x_3, x_4]$.
System becomes

$$
\begin{array}{ccccccccc}
1x_1 & + & 0.5x_2 & - & 2x_3 & + & 4x_4 & = & -8 \\
 & & 3x_2 & + & 3x_3 & + & 2x_4 & = & 3 \\
 & & & & 1x_3 & + & 5x_4 & = & -4 \\
 & & & & & & 2x_4 & = & 6
\end{array}
$$

Solving a triangular system of linear equations: Backward substitution

$$
\begin{aligned}
1x_1 + 0.5x_2 - 2x_3 + 4x_4 &= -8 \\
3x_2 + 3x_3 + 2x_4 &= 3 \\
1x_3 + 5x_4 &= -4 \\
2x_4 &= 6
\end{aligned}
$$

Solution strategy:

- Solve for x_4 using fourth equation.
- Plug value for x_4 into third equations and solve for x_3.
- Plug values for x_4 and x_3 into second equation and solve for x_2.
- Plug values for x_4, x_3, x_2 into first equation and solve for x_1.

Solving a triangular system of linear equations: Backward substitution

$$1x_1 + 0.5x_2 - 2x_3 + 4x_4 = -8$$
$$3x_2 + 3x_3 + 2x_4 = 3$$
$$1x_3 + 5x_4 = -4$$
$$2x_4 = 6$$

$$2x_4 = 6 \rightarrow x_4 = \frac{6}{2} = 3$$
$$1x_3 = -4 - 5x_4 = -4 - 5(3) = -19$$
$$\rightarrow x_3 = -19$$
$$3x_2 + 3x_3 + 2x_4 = 3$$
$$3x_2 + 3(-19) + 2(3) = 3x_2 - 57 + 6 = 3x_2 - 51 = 3$$
$$\rightarrow 3x_2 = 54 \rightarrow x_2 = \frac{54}{3} = 18$$
$$1x_1 + 0.5x_2 - 2x_3 + 4x_4 = x_1 + 0.5(18) - 2(-19) + 4(3) = -8$$
$$x_1 + 9 + 38 + 12 = x_1 + 59 = -8 \rightarrow x_1 = -67$$

Solving a triangular system of linear equations: Backward substitution

Quiz: Solve the following system by hand:

$$
\begin{array}{rcrcrcr}
2x_1 & + & 3x_2 & - & 4x_3 & = & 10 \\
 & & 1x_2 & + & 2x_3 & = & 3 \\
 & & & & 5x_3 & = & 15
\end{array}
$$

Answer:

$$x_3 = 15/5 = 3$$

$$x_2 = 3 - 2x_3 = -3$$

$$x_1 = (10 + 4x_3 - 3x_2)/2 = (10 + 12 + 9)/2 = 31/2$$

Solving a triangular system of linear equations: Backward substitution

Hack to implement backward substitution using vectors:

- Initialize vector `x` to zero vector.
- Procedure will populate `x` entry by entry.
- When it is time to populate x_i, entries $x_{i+1}, x_{i+2}, \ldots, x_n$ will be populated, and other entries will be zero.
- Therefore can use dot-product:
  - Suppose you are computing x_2 using $[0, 3, 3, 2] \cdot [x_1, x_2, x_3, x_4] = 3$
  - So far, vector `x` $= [x_1, x_2, x_3, x_4] = [0, 0, -19, 3]$.
  - $x_2 := 3 - ([0, 3, 3, 2] \cdot \mathtt{x})$

```
def triangular_solve(rowlist, b):
    x = zero_vec(rowlist[0].D)
    for i in reversed(range(len(rowlist))):
        x[i] = (b[i] - rowlist[i] * x)/rowlist[i][i]
    return x
```

Solving a triangular system of linear equations: Backward substitution

```
def triangular_solve(rowlist, b):
    x = zero_vec(rowlist[0].D)
    for i in reversed(range(len(rowlist))):
        x[i] = (b[i] - rowlist[i] * x)/rowlist[i][i]
    return x
```

Observations:

- If `rowlist[i][i]` is zero, procedure will raise `ZeroDivisionError`.
- If this never happens, solution found is the *only* solution to the system.

Solving a triangular system of linear equations: Backward substitution

```
def triangular_solve(rowlist, b):
    x = zero_vec(rowlist[0].D)
    for i in reversed(range(len(rowlist))):
        x[i] = (b[i] - rowlist[i] * x)/rowlist[i][i]
    return x
```

Our code only works when vectors in `rowlist` have domain $D = \{0, 1, 2, \ldots, n-1\}$.

For arbitrary domains, need to specify an ordering for which system is "triangular":

```
def triangular_solve(rowlist, label_list, b):
    x = zero_vec(set(label_list))
    for r in reversed(range(len(rowlist))):
        c = label_list[r]
        x[c] = (b[r] - x*rowlist[r])/rowlist[r][c]
    return x
```