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

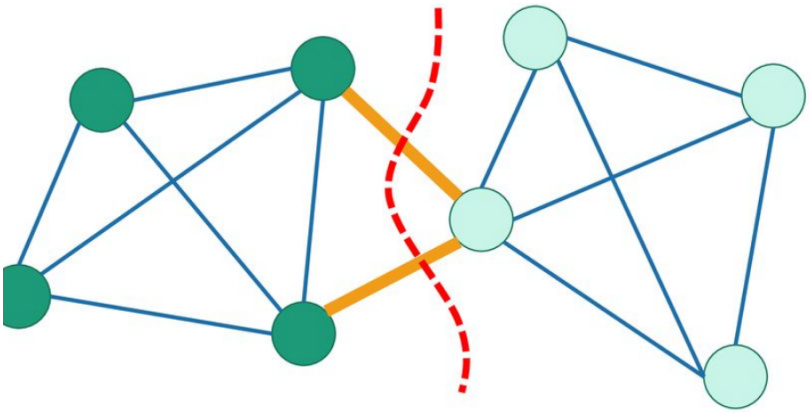


Figure 1. Example of global min-cut. Note that the global min-cut is not necessarily unique

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

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

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.

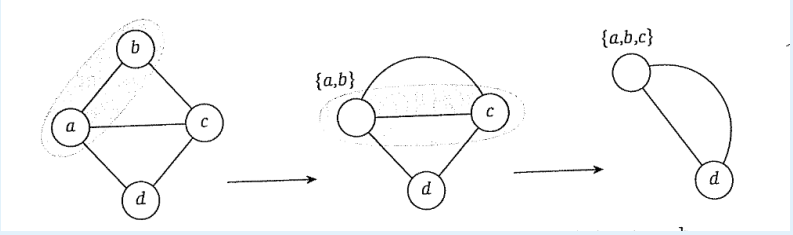


Figure 2. Example of a series of contractions

CONTRACTION$(G = (V, E))$

1 **while** G has more than 2 supernodes
2 Pick an edge $e = (u, v)$ uniformly at random
3 Contract e, remove self-loops
4 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. So the probability than an edge in F is contracted at any step is

$$\frac{k}{\binom{k}{2}} = \frac{2}{n} \tag{1.3}$$

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

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

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

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

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

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...

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

From what we found previously we know that this is

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

□

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*.

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$(G = (V, E))$

1 **if** G has two supernodes corresponding to $S, \hat{S}$
2 **return** $S, \hat{S}$
3 Run the contraction algorithm until $\frac{n}{\sqrt{2}} + 1$ supernodes remain
4 Let G' be the resulting contracted multigraph
5 $(S_1, \hat{S}_1) =$ MIN-CUT(G')
6 $(S_2, \hat{S}_2) =$ MIN-CUT(G')
7 **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 $\frac{1}{O(\log n)}$[2]

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 [3] be solved with the master theorem.

□

[2] *So repeat the algorithm $O(\log(n))$ times, leading to $O(n^2 \log^2 n)$ runtime*

[3] *I think?*

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[4]

- 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.

[4] *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[5] 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.

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

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 $d(p,q)$[6] 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.

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

rabins-algorithm(P)

```
1  Randomly order P as p_1, p_2 ... p_n
2  cp = {p_1, p_2}
3  Δ = dist(p_1, p_2)
4  for i = 3 to n
5      if ∃ q ∈ P_{i-1} = p_1 ... p_{i-1} s.t. dist(p, q) < Δ
6          cp = {p, q}
7          Δ = 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)

Note that there are two types of randomized algorithms:

- Monte Carlo algorithms: bound on worst-case time & probability of 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

- 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 $min\{dist(p, q) : q \in Q\} \geq \delta$
  - INSERT-SLOW(p) inserts p into Q even if $min\{dist(p, q) : q \in Q\} < \delta$
  - CHECK-CLOSEST(p) checks if $min\{dist(p, q) : q \in Q\} < \Delta$ 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 ... p_n
2  cp = {p_1, p_2}
3  Δ = 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         Δ = 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.

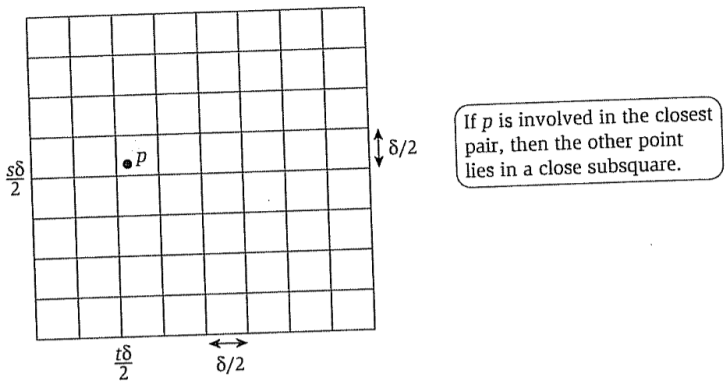


Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{ij}.

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. [7]

[7] *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.*

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.

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

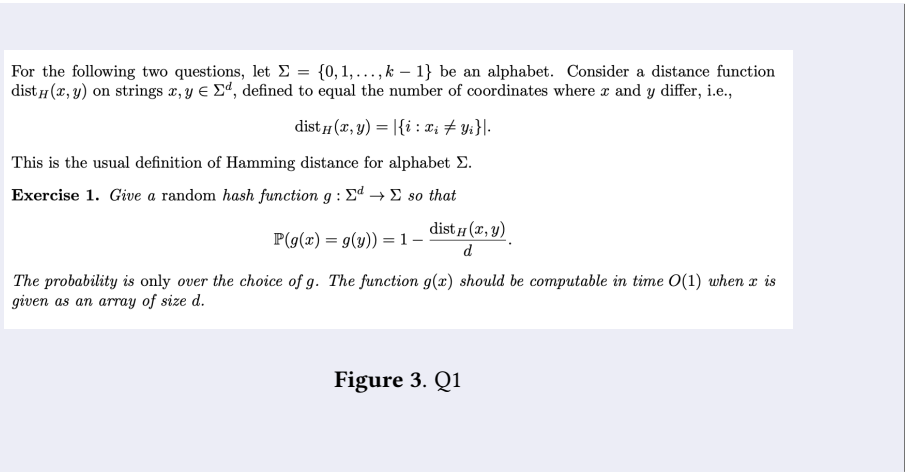
- Now, what's $P(Z_i) = 1$?
  - Let $\{p_i, p_k\}$ be a closest pair in P_i
  - If $Z_i = 1$ then p_i or p_k is p_i
  - $P(Z_i = 1) \leq \frac{2}{i}$ (There are two bad options out of i options)

- 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 $\text{dist}_H(x, y)$ on strings $x, y \in \Sigma^d$, defined to equal the number of coordinates where x and y differ, i.e.,

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

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*

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

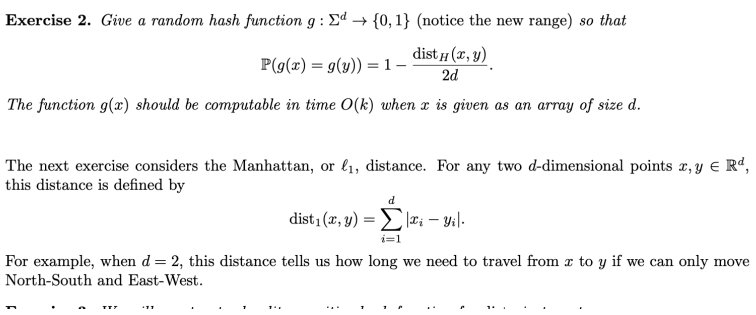
The probability is only *over the choice of g. The function $g(x)$ should be computable in time $O(1)$ when x is given as an array of size d.*

Figure 3. Q1

Comment

Randomly sample a dimension[8] and then the probability of the hash functions being equal to each other is proportional to the hamming distance between the strings (consider random...).

[8] *O(1)*



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

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

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

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

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 $P(g(x) \neq g(y) | \text{selected dimension i}) = \frac{|\{ix_i \neq y_i\}}{2d}$.
O(k) since you have to sample it k times

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.* $0 \leq a \leq b \leq 1$*, give an expression for the probability* $\mathbb{P}(a \leq t < b)$ *in terms of* a *and* b*.*
2. *Give a random hash function* $g : [0,1]^d \to \{0,1\}$ *so that*

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

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 $0 \leq a \leq b \leq 1$, $P(a \leq t < b)$ is

Subsection 1.5

Nearest Neighbours & Clustering

Suppose we have a data set P of n entries[9], 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?

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

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:

- INSERT(P,x): insert x into P
- 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 APXNEARESTNEIGHBOUR(P,x): output a string $y \in P$ such that

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

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.

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.[10]
- 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.

[10] *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.*

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[11].

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

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.[12] Then,

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

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

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

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

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

And that they don't collide

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

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 $[d]$,
So,

$$dist(x,y) \leq r : P(g_I(x) = g_I(y)) \geq 1 - \left(\frac{r}{d}\right)^k = p_1^k \tag{1.21}$$

$$dist(x,y) \geq Cr : P(g_I(x) = g_I(y)) \leq 1 - \left(\frac{Cr}{d}\right)^k = p_2^k \tag{1.22}$$

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

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

$$g_I(x) = (0,1,0) \tag{1.23}$$

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

Universal hash function: Each h_i is a random hash function that such that

$$\forall\, u, v = \{0,1\}^k, u \neq v \tag{1.24}$$

The probability of collision

$$P(h_i(u) = h_i(v)) \leq \frac{1}{m} \leq \frac{1}{n} \tag{1.25}$$

i.e. is small.
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 ≠ ∅
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 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.

Analysis

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

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

$$F = \{x \in P : dist(q, x) > Cr\} \tag{1.26}$$

For the analysis we're interested in:

1. q collides with points in $F \leq 12L$ times (with multiplicity)
2. q collides with $x*$

What is the probability of both happening?

1. Expected number of collisions with far points
(From prior)

$$\forall\, l, \forall\, x \in F : P[g_{I_l}(x) = g_{I_l}(q)] \leq p_2^k = (1 - \frac{Cr}{d})^k \tag{1.27}$$

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.

Let us choose k such that $p_2^k = (1 - \frac{Cr}{d})^k \leq \frac{1}{n}$.

$$k \equiv \frac{\log n}{\log \frac{1}{p_2}} \tag{1.28}$$

TODO: Test out the algebra

$$z_{x,l} = \begin{cases} 1 & \text{if x collides with q in T\_l} \\ 0 & \text{otherwise} \end{cases} \tag{1.29}$$

The number of collisions with far points is

$$\sum_{l=1}^{L} \sum_{x \in F} z_{x,l} \tag{1.30}$$

NOTE: shouldn't this be w.r.t F?
I.e. we get a 1 if x collides with q and 0 otherwise
Taking the expectation,

$$E[\text{num collisions with far points}] = \sum_{l=1}^{L} \sum_{x \in F} E[X_{x,l}] = \sum_{l=1}^{L} \sum_{x \in F} P[z_{x,l} = 1] \tag{1.31}$$

$$P[z_{x,l} = 1] = P[g_{I_l}(x) = g_{I_l}(q)] + P[g_{I_l}(x) \neq g_{I_l}(q), h_l(g_{I_l}(x)) = h_l(g_{I_l}(q))] \leq \frac{1}{n} + \frac{1}{n} \leq \frac{2}{n} \tag{1.32}$$

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

Comment

Markov's Inequality
For any random variable $Z \geq 0$, $P[Z \geq t] \leq \frac{E[Z]}{t}$
There are cases where

$$P[x \geq EZ] \leq \frac{1}{z} \tag{1.33}$$

Let Z be the total number of collisions with far points.

$$P[Z \geq 12L] \leq \frac{2L}{12L} = \frac{1}{6} \tag{1.34}$$

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 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.

$$P(\text{x* collides with q in some hash table}) = 1 - P(\text{x* do}$$
$$= 1 - P[\text{x* does not collide with q in T1}] \cdot P[\text{x* does not collide with q in T2}] \dots$$
$$\geq 1 - \prod_{l=1}^{L} P[g_{I_l}(x*) \neq g_{I_l}(q)]$$
$$\geq 1 - (1 - p_1^k)^L$$
$$\geq 1 - e^{-2p_1^k} \tag{1.35}$$

Let

$$\rho = \frac{\log\left(\frac{1}{p_1}\right)}{\log\left(\frac{1}{p_2}\right)} \tag{1.36}$$

Via some inequality (look it up later)

So $p_1 = p_2^\rho \Leftrightarrow p_1^k = p_2^{k\rho} \Rightarrow \rho \approx \frac{1}{c}$
Let's define a parameter $L = 2n^\rho$ Then, the probability is bounded by $P[\dots] \geq 1 - \frac{1}{e^2} > \frac{5}{6}$
Time is dominated by $O(d * L) = O(dn^\rho)$

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.

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,

```
1 unsigned long int h = 0x6f6c6c6548; // ascii for hello
2 unsigned long int w = 431316168567; // ascii for world
3 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 2$\hat{}$128 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:

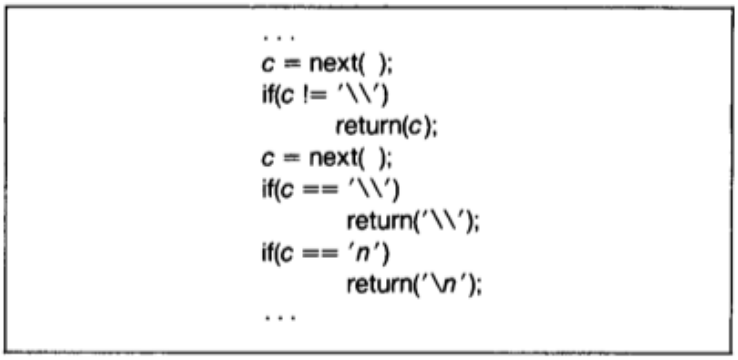
- Another security concept is the **threat**, or any method that can breach security.
- An exercise of a threat is called an **exploit** and a successful exploit causes the system to be compromised. Common threats include internet connections/open ports.
- **Vulnerabilities** are flaws that that weaken 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

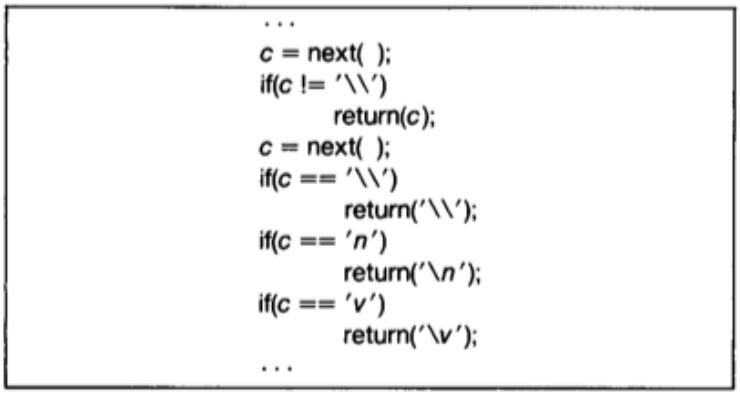
Comment

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



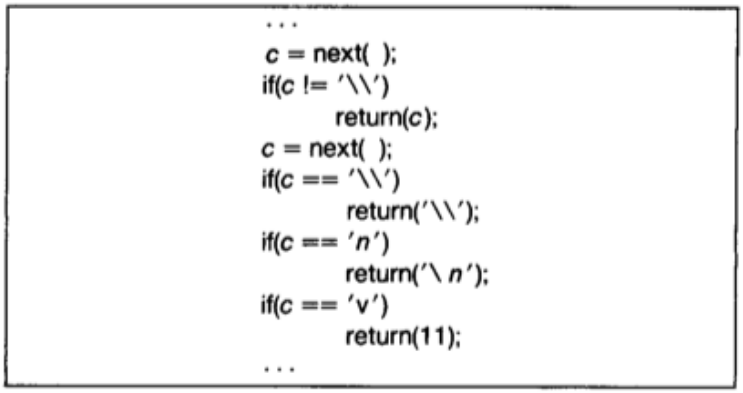
```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

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 6. 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.

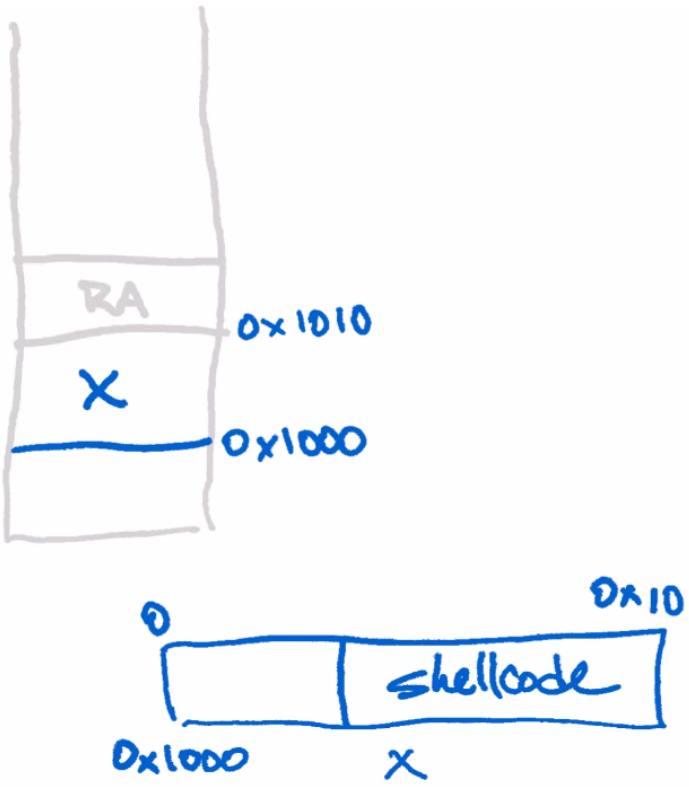


Figure 7. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

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.

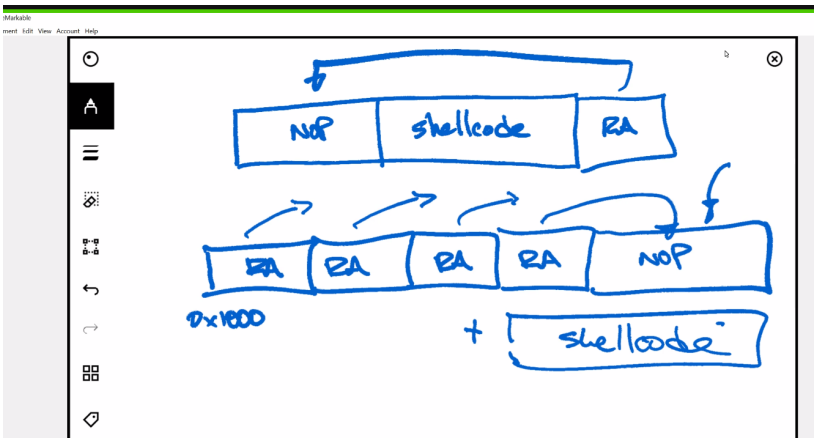


Figure 8. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

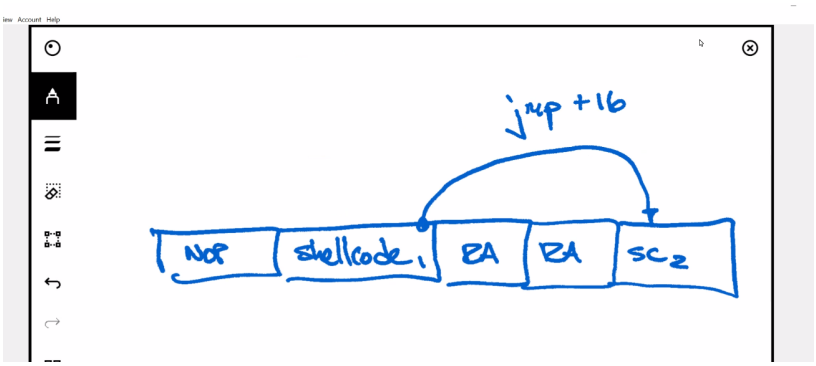


Figure 9. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.3

Format string Vulnerabilities

```
1 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:

```
1 char* str = "Hello world";
2 printf(str); // 1
3 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
}
```

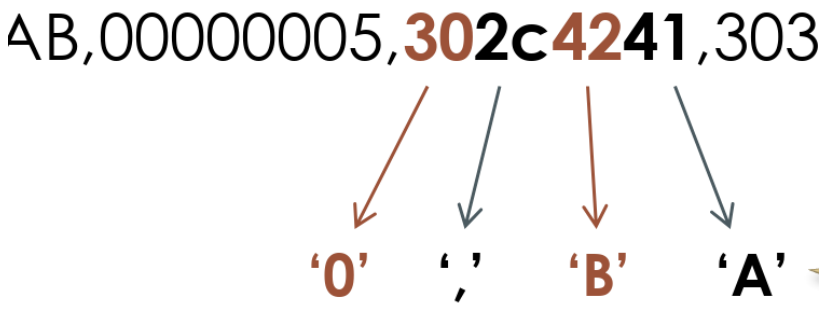
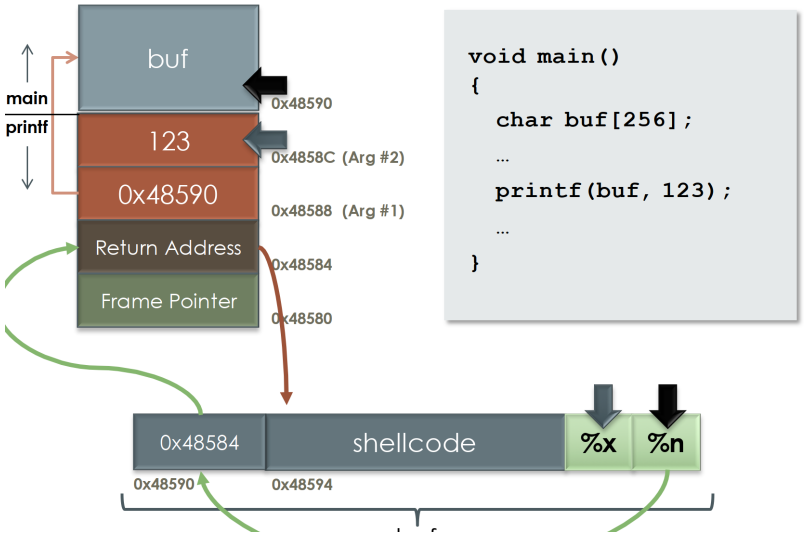


Figure 10. ASCII decoding

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:



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 11. The printf count increments by 243 with %243d. Shorthand

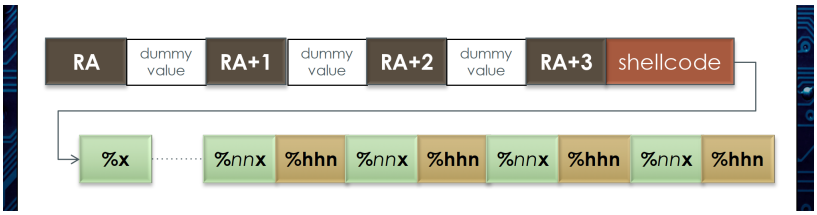


Figure 12. The gaps are there because

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

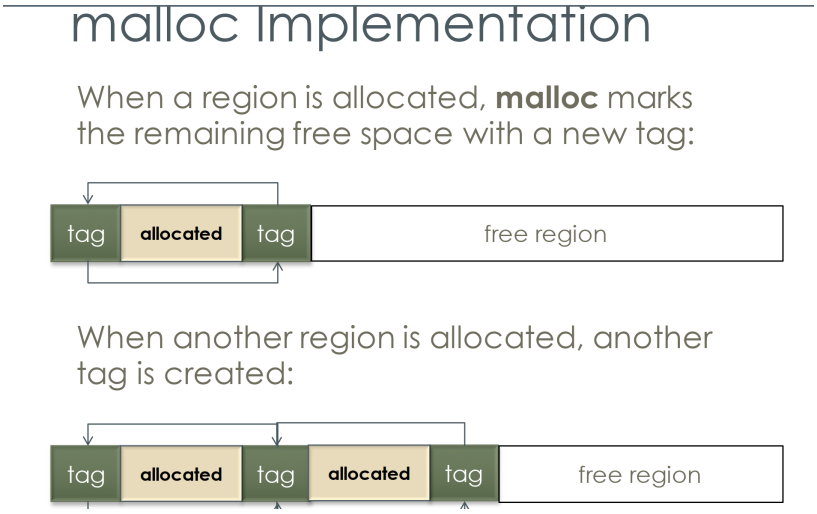
```
1  p = malloc(128);
2  q = malloc(128);
3  free(p);
4  free (q);
5  p = malloc(256);
6  // this is where the attack happens; the fake tag, shell code, etc
7  strcpy(p, attacker_string);
8  free(q);
```

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 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.[13]

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

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.

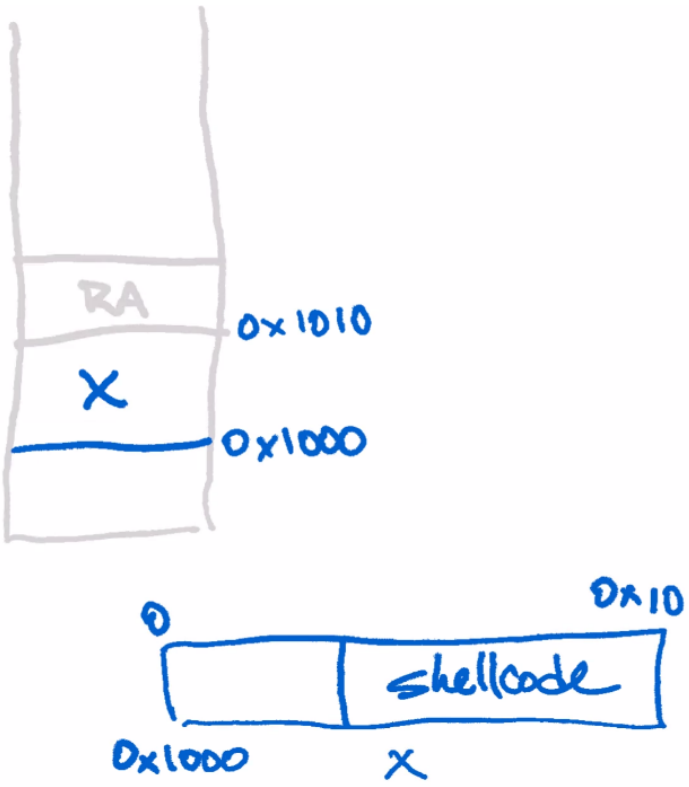


Figure 13. Bufferoverflow to write to the return address. Shellcode is a sequence of instructions that is used as the payload of an attack. It si called a shellcode because they commonly are used to start a shell from which the attacker can do more.

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.

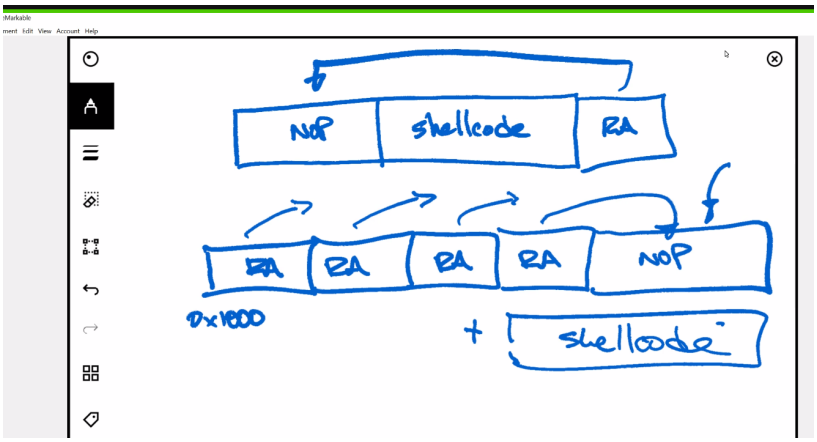


Figure 14. Can create a RA sled with a NOP leading to shellcode and then try it from e.g. 0x1000, 0x2000 and so forth to find where to attack from.

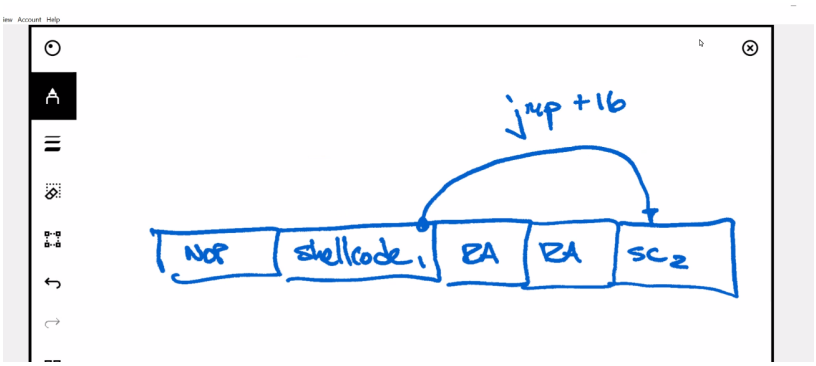


Figure 15. Another technique may involve placing shellcode all over the place, of which each one may be a valid entrypoint into the shellcode.

Subsection 2.7

Format string Vulnerabilities

```
1 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:

```
1 char* str = "Hello world";
2 printf(str); // 1
3 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
}
```

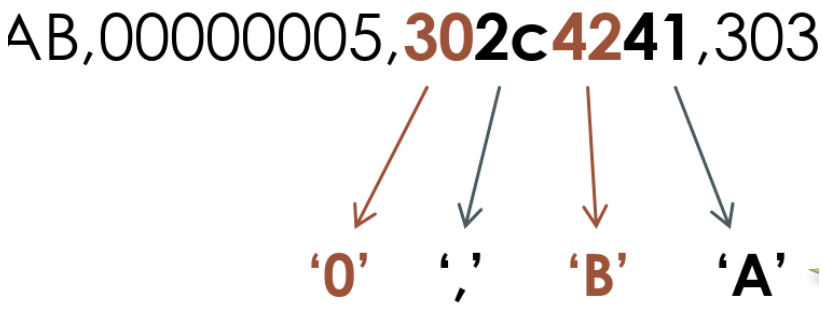
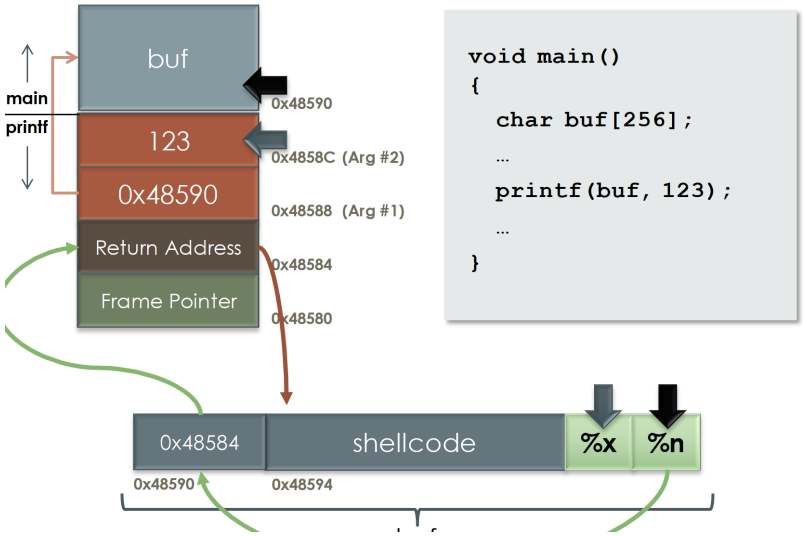


Figure 16. ASCII decoding

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:



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

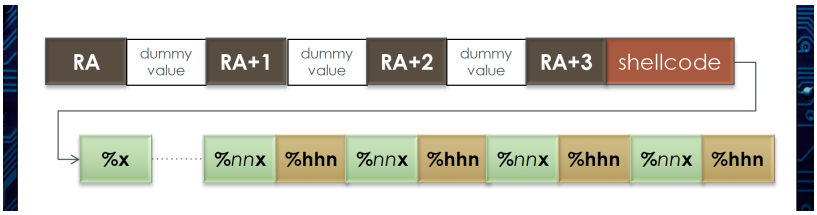


Figure 18. The gaps are there because

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 2.8

Double-Free vulnerability

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

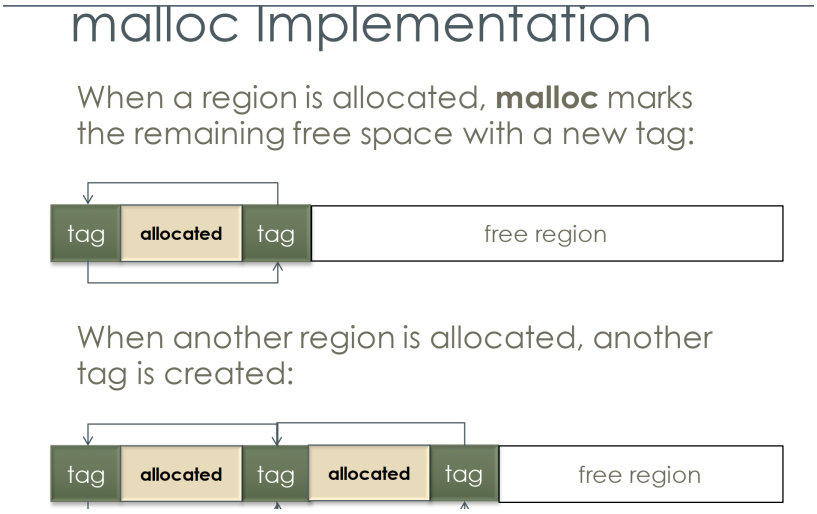
```
1  p = malloc(128);
2  q = malloc(128);
3  free(p);
4  free (q);
5  p = malloc(256);
6  // this is where the attack happens; the fake tag, shell code, etc
7  strcpy(p, attacker_string);
8  free(q);
```

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 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.[14]

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

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.

```
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 19. 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 be 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

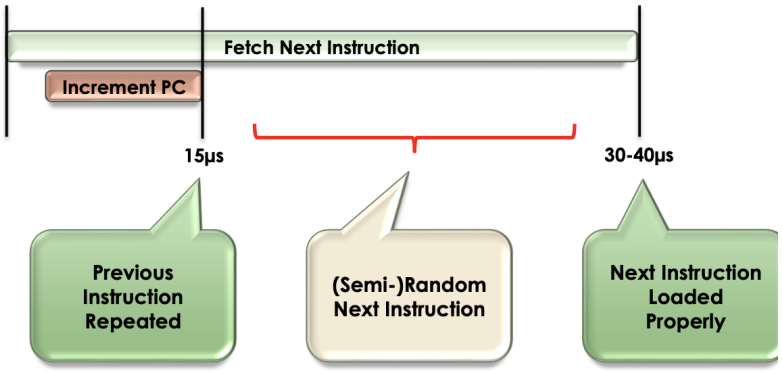


Figure 20. With some careful timing we can cause the CPU to skip or repeat an instruction.

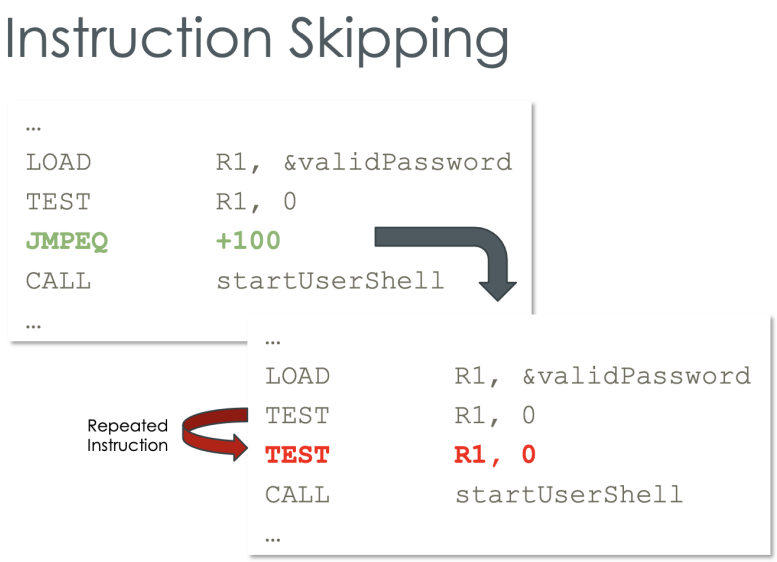
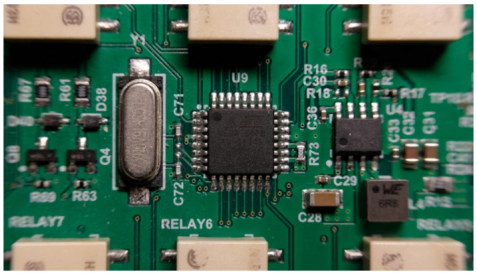


Figure 21. An example of where this can be useful: skipping the JMP instruction of an IF statement

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.");
L: SKIP THIS!  return;
}

// Login correct
…
```

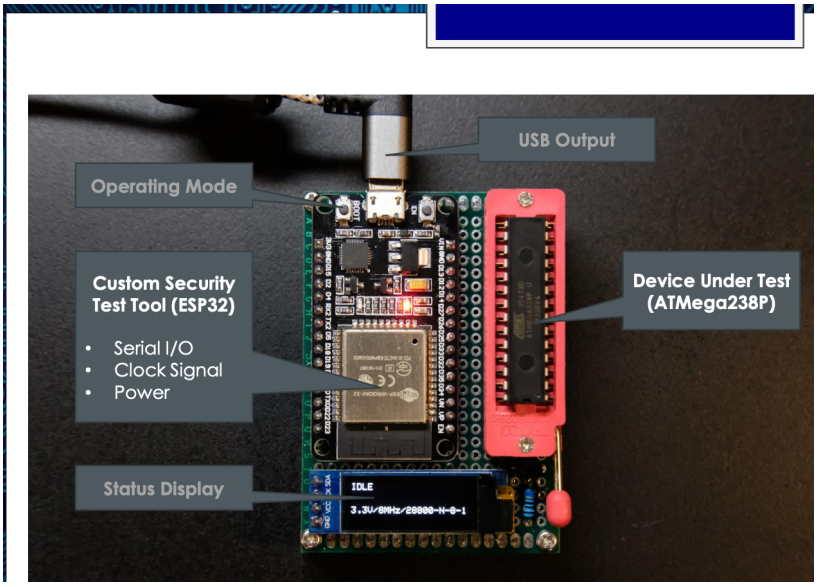


Figure 22. The ATMega238P hooked up to a custom security test tool built on top of a ESP32

Our attack is to use a **clock glitch**[15] 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.

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

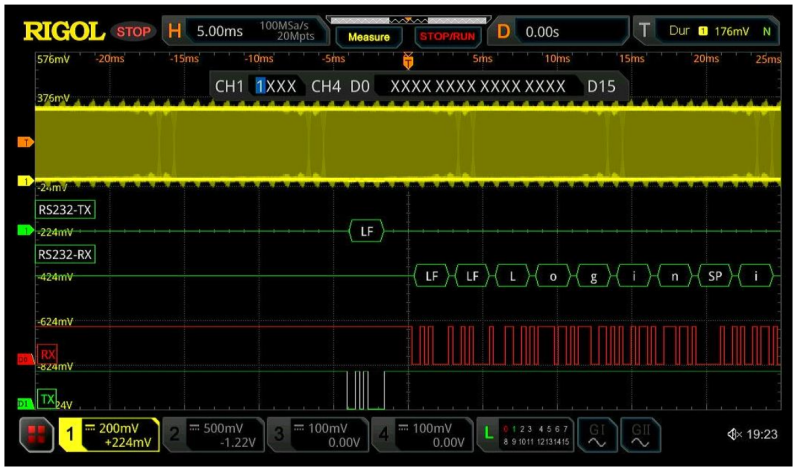


Figure 23. Looking at the oscilloscope to show the sweeped input

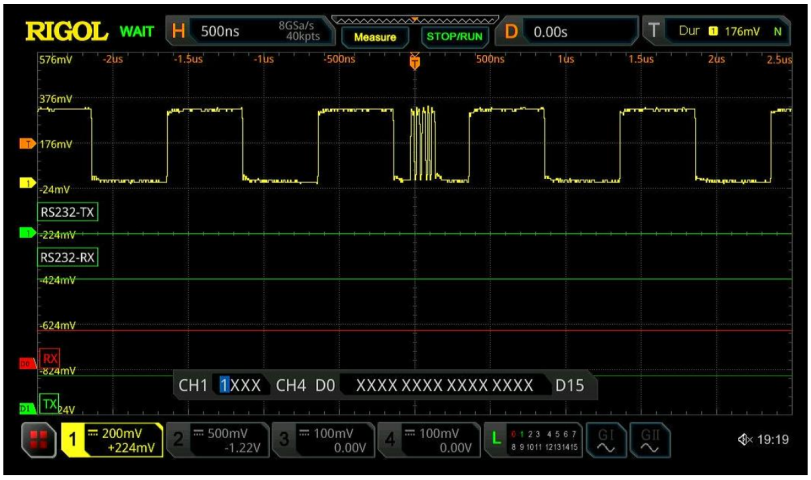


Figure 24. Note crazy clock pulses which we try to line up with the return instruction. If we have a really good chip we can try it with only 1 pulse, but here we use 5 pulses because we're on a cheaper chip. We also don't happen to care too much about whether or not if we disrupt too many of the other 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.

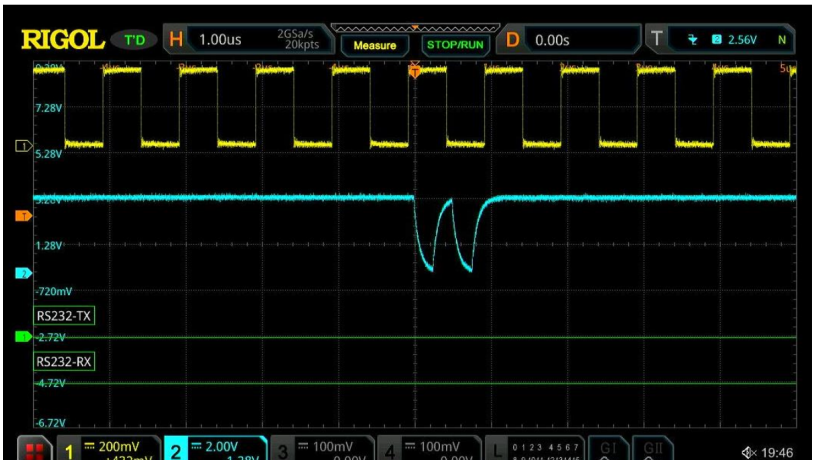


Figure 25. A power glitch attack disrupts the fetch instruction or the decode

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).

```
1  unsigned int printhelloworld() {
2      printf("Hello World!");
3      return 5;
4  }
5  int main (int argc, char *argv[]) {
6      unsigned int result = 0;
7      result = printhelloworld();
8      if (result == 4) {
9          printf("super secrete string\n");
10     }
11     return 0;
12 }
```

Figure 26. 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 [16]

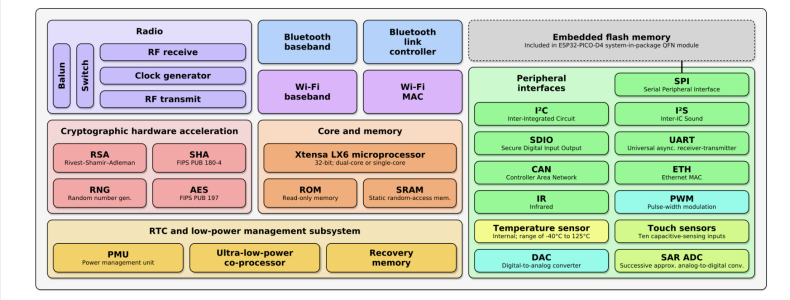
[16] *if we stack a lot of hole-y cheese on top of each other it will be opaque*

Subsection 2.15

Cryptography

Comment

Case Study: Espressif ESP32



- 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

Section 3

ECE353 Operating Systems

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.

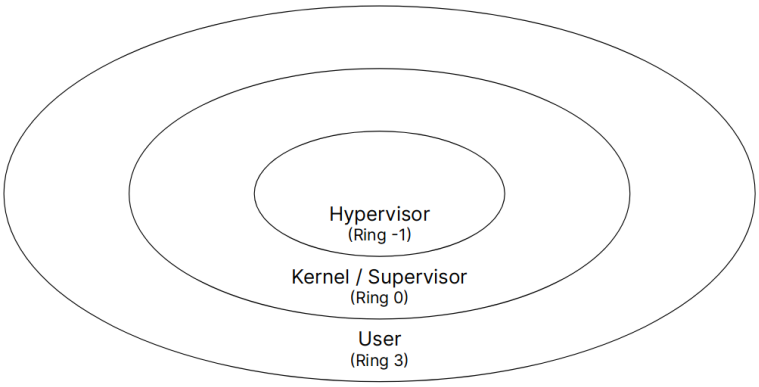


Figure 27. x86 Instruction access rings. Each ring can access instructions in its outer rings.

The kernel runs in, well, Kernel mode. **System calls** offer an interface between user and kernel mode[17].

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

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:

```
1 ssize_t write(int fd, const void* buf, size_t count);
2 // writes bytes to a file descriptior
```

[17] *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

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
```

This output is followed by information about the program and section headers

`strace` can be used to trace systemcalls. For example let's look at the 168-byte hello-world example

```
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 28. 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.

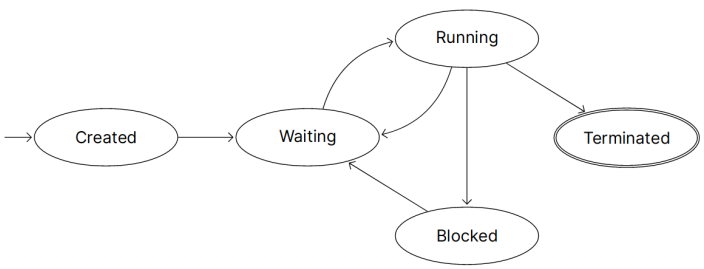


Figure 29. A possible process state diagram

These state changes are managed by the Process and OS[18] 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.

[18] *I think*

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[19].

Process state can be read in `/proc` for linux systems.

[19] *completely fair scheduler*

In `c` most system calls are wrapped to give additional features and to put them more concretely in the userspace.

```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 30. 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 zomibe 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

```c

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`[20]

[20] *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 8

```c
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 9 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.

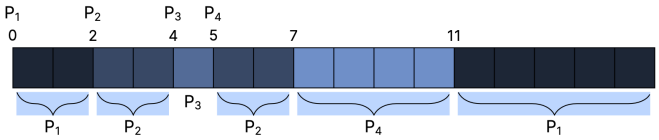
Definition 10 SJF (Shortest job first): schedule the job with the shorts execution time first. Though it's optimal at minimizing average wait times, since we don't know how long each process takes it may not be practically optimal. It also has the downside of potentially starving longer jobs.

Definition 11 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. [21]

[21] *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):

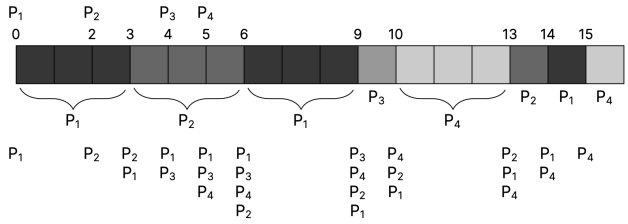


Figure 31. RR example with quanta of 3 units. Average number of switches is 7, average waiting time is $\frac{8+8+5+7}{4} = 7$, average response time is $\frac{0+1+5+5}{4}$. Note that ties are handled by favouring new processes.

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 12

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[22] at a higher priority level, then returns to its original level.

[22] *The blocking portion*

- 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 13 **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[23]. 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[24]
  - 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[25] 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.

[23] *to deal with cache locality*

[24] *Also, there are hard and soft real-time systems. Linux also implements FCFS and RR scheduling which you can select for tasks.*

[25] *completely fair scheduler*

Definition 14 **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 15 **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...

```
1
2  int main(int argc, char* argv[]) {
3      if (argc != 2) { return EINVAL; }
4      // will have 3 fd open: 0, 1, 2 for stdin, stdout, stdrr
5
6      int in_pipefd[2] = {0};
7      int out_pipefd[2] = {0};
8      check_error(pipe(out_pipefd), "outpipe");
9      check_error(pipe(id_pipefd), "inpipe");
10     // 0 is read end, 1 is write end
11     // want to use the pipe to communicate with the child
12     // pipe before fork, so that both parent and child have access to
   ↪   the pipe
13     // replace child stdout to out_pipefd[1]
14     // and replace parent std
15     pid_t pid = fork();
16     if (pid > 0) { parent(in_pipefd, out_pipefd, pid); }
17     else { child(in_pipefd, out_pipefd, argv[1]); }
18     return 0;
19 }
```