IY3840 Malicious Software

Botnet Threat

* Network of compromised devices (bots) controlled by bot master

Responsible:

* Large-scale network probing
* Large-scale DDOS
* Sending emails (SPAM)
* Click-fraud campaign
* Information theft (PII, financial, IP)

Shift from (FUN) to (PROFIT)

Torpig (Trojan horse)

* Distributed via mebroot
* inject via 29 different app as DLL
* steals sensitive info
* http injection for phishing
* use encrypted http as C&C protocol
* use domain flux to locate C&C server

Torpig (Mebroot)

* Spread via drive-by download
* Sophiscated rootkit (overwrite master boot record)

Drive-by download = just visit website, need not click on anything, automatically run exploitation

C&C = vulnerable to take down

Domain Flux = periodically generate new C&C domain name

* Same DGA
  + Weekly domain
  + Daily domain
* 3 fixed domains (if all else fails)
* Attempts to connect every 20 min

RE DGA and C&C protocol

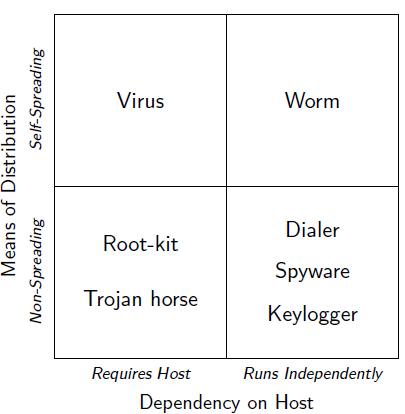
* Got the domain
* Registered domains
* Pushed new binary after that
* Controlled the botnet for 10 days
* Collected data

Data Collection principle

* Operated so harm/damage minimized
* Enable notification and remediation of affected parties

Botnets

Malware = unwanted software & executable code used to perform unauthorized, often harmful, action on a computing device



Virus

* Self replicating
* Needs Host to infect

Worm

* Self replicating
* Spread autonomously over network
  + Exploits vuln affecting large number of hosts
  + Sends itself via email

Trojan horse

* Malicious program disguised as legitimate
* Many different malicious actions
  + Spy, hide, remote access

Root kit

* Used to keep access to compromised system
* Hide files, processes, network connections
  + User & kernel level

Fighting malware

Goals

1. Understand malware behaviours
2. Identify + classify malware
3. Generate effective malware detection models

How

* Collect malware samples (infection strat)
* Analyse samples
  + Static analysis  
    (anti RE, obfuscation, encryption)
    - Linear sweep
    - recursive traversal
  + Dynamic analysis (environment-limited analysis)
    - Symbolic execution
* Extract (generalise) malicious behaviour
  + Host
  + Network
* Generate & deploy detection models

Problems hard

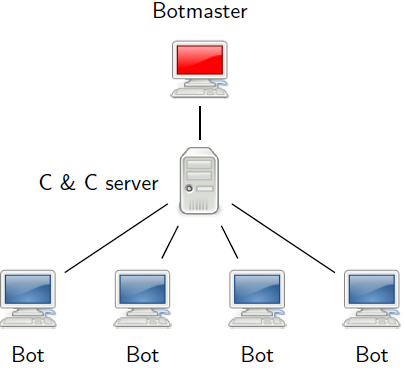
* Cat & mouse game
* Lack of general definition of malicious behaviour
* Victims often help attackers

Bot

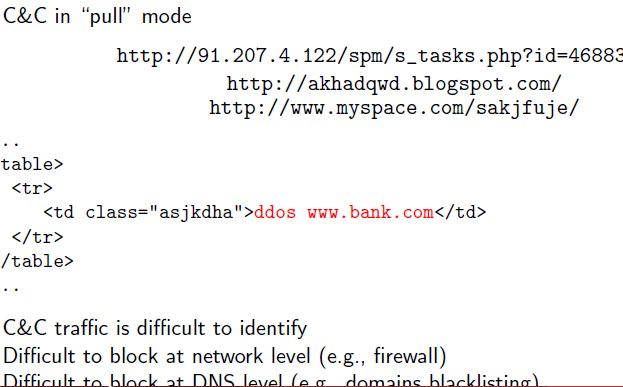
* Autonomous programs performing tasks
* Trend in malicious dev
* Benign (in IRC), eggdrop
* Malicious IRC bots
  + Takeover wars (control)
  + Trash talking
  + DoS to force net split
  + Proxies to hide atker origin
* Parallel, malicious dev
* Diff modules to carry out malicious tasks
* Remote controlled by attacker (bot master)
  + C2 channels, robust   
    (fastflux, domain flux, push/pull, P2P)  
    Search keys in P2P network, Hardcoded
* Incorporated in network of compromised machines
  + Botnets
* $$$, and main vehicle for carrying out criminal activities
* Infection & spreading
  + Network worm (vuln)
  + Email
  + Trojan
  + Drive-by download
    - Malicious scripts (SQL, JS, ads)
    - Drive-by downloads (JS, VB)
    - Redirection
  + Existing backdoor
  + Specialized service (ExploitAAS)

C & C

Centralized control (HTTP, IRC)  
Distributed Control (P2P)  
Push vs Pull (from commander)

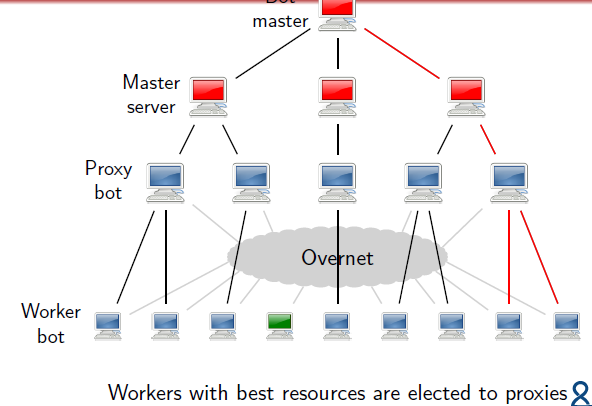


HTTP based

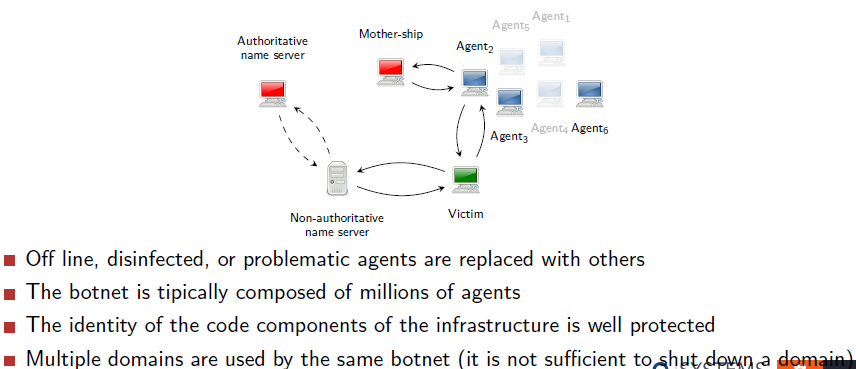
* 

Prevent the rendezvous

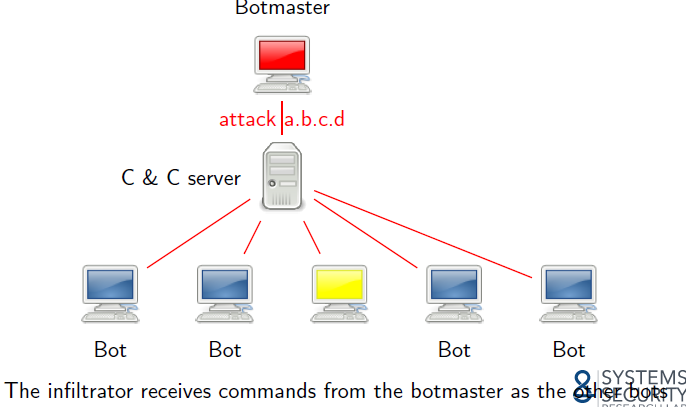
* Network level ACL
* DNS ACL
* HTTP ACL
* DDoS C2 server

P2P based botnet

Fast Flux



Infiltrate botnet (IRC=can, HTTP!=can)



Botnet detection

* Network traffic analysis
  + Payload dependent
  + DGA modeeling
  + Payload agnostic approaches
* Group similar comms tgt to identify outliers
  + Cluster analysis

**Cluster analysis  
Unsupervised learning**- collection of objects without class labels.  
- methods for building model that capture struct of data exists  
- training without teacher  
- extremely useful (costly, labels not known beforehand, large datasets = compressed into small set of prototypes)

Clustering = organizing objects whose members similar in some ways

* High intra-cluster similiarity
* Low inter-cluster similiarity

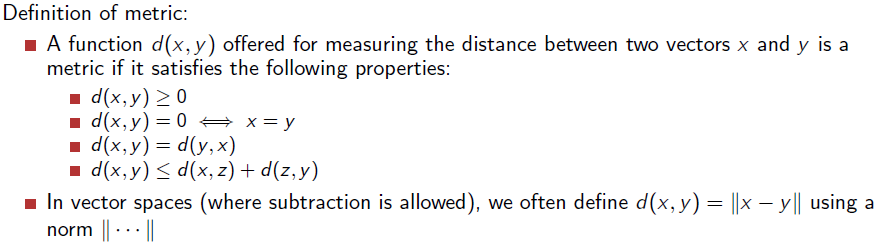
No right/wrong (different clustering = different things)

Some clustering algo have probabilistic interpretations

Non-parametric Clustering

* Define a measure of (dis)similiarity
  + The different distance (Minkowski)
* Define criterion function for clustering
* Define algo to minimize/maximize criterion function

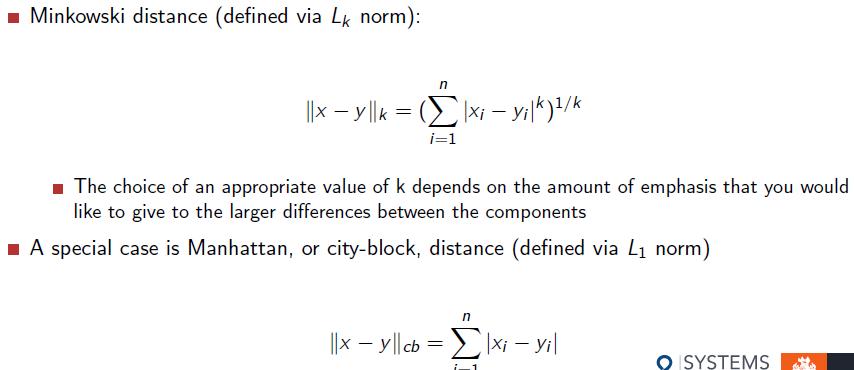
Distance measure



Try to make Z closer to the line (x,y)

Z

X Y



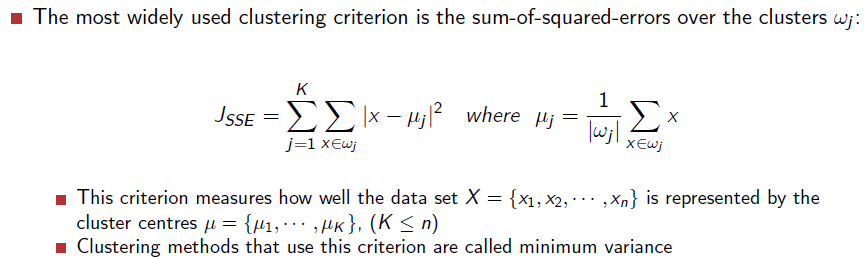
Minkowski = multiple features, n

If n = 2, then its Euclidean distance

If n=1 = Manhattan distance

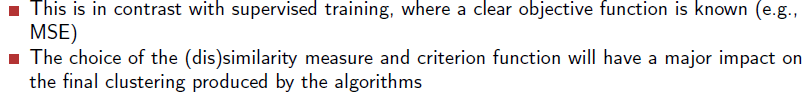
If n=(infinity), = Chebyshev distance

Criterion Function for clustering (for optimization)

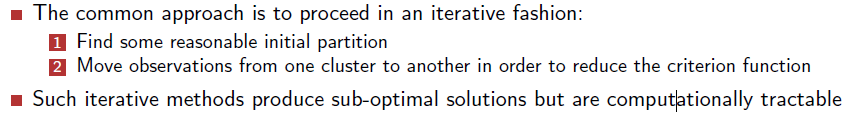


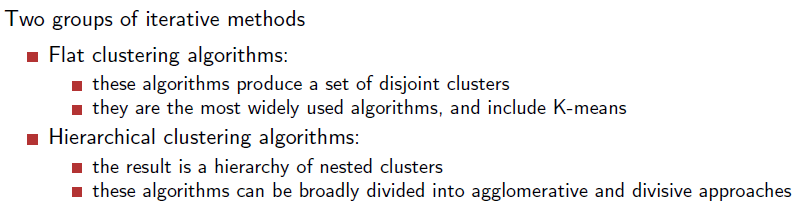
=Measure distance of Points are close to the center

Validity = highly subjective



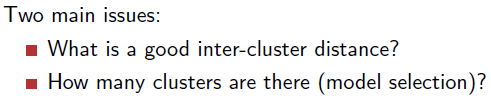
Iterative optimization



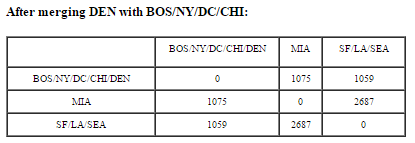
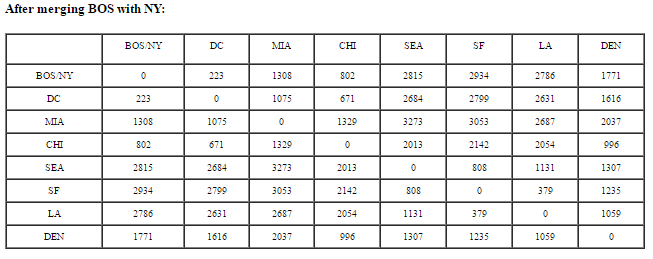
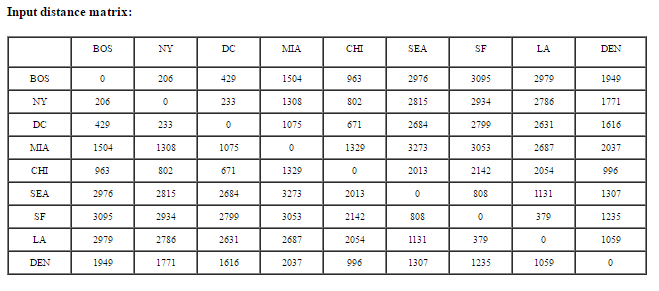


Hierarchical clustering

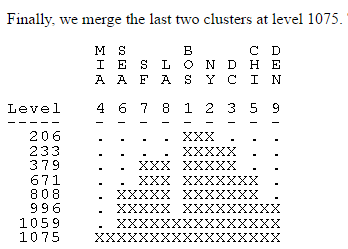
Tree-like structure (2 ways, agglomerative, divisive)



Dendograms



Min = will create a tail



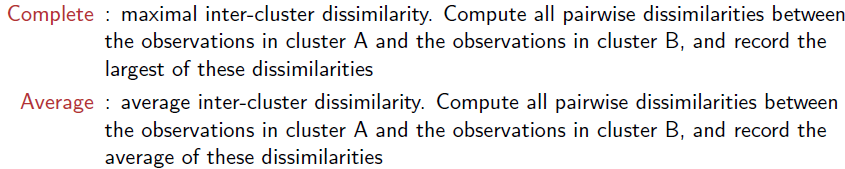
Cutting Tree:

* Hierarchical method = produce several partitions

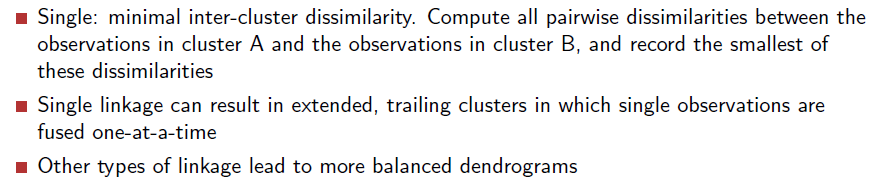
Cutting Criterion = using a threshold

2 popular types of linkage

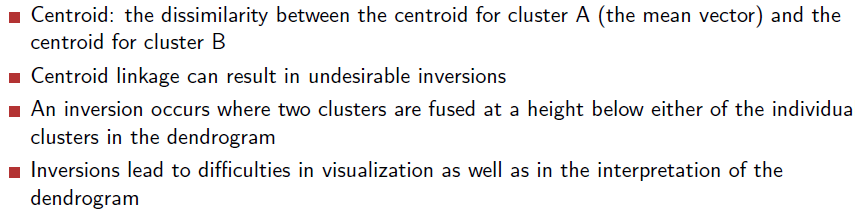
* Complete
* Avg



Single linkage:

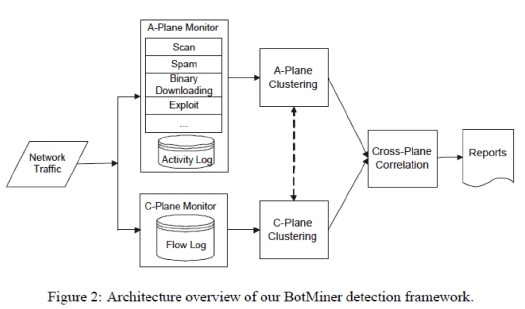


Centroid Linkage:



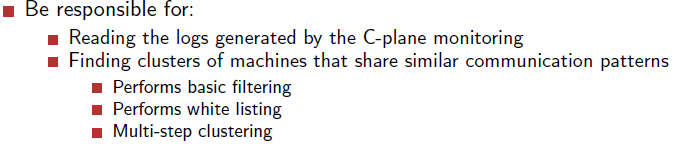
Botminer

* Via C&C/malicious activities
* Structure = centralized / p2p

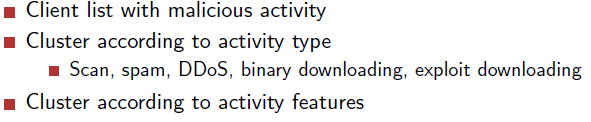


A=activity C=comm

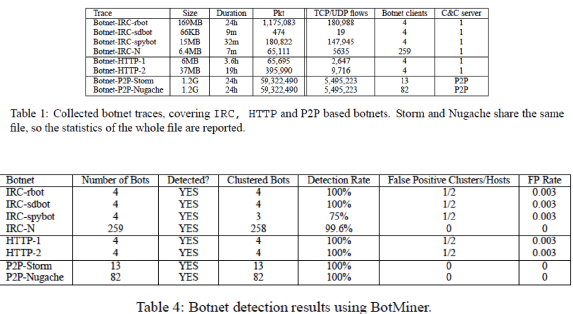
C-plane clustering:



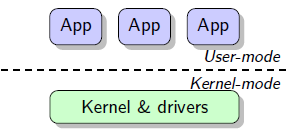
A-plane clustering



Cross-plane correlation = find intersections, get score



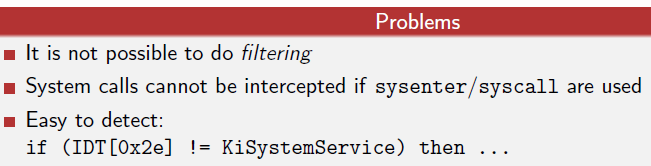
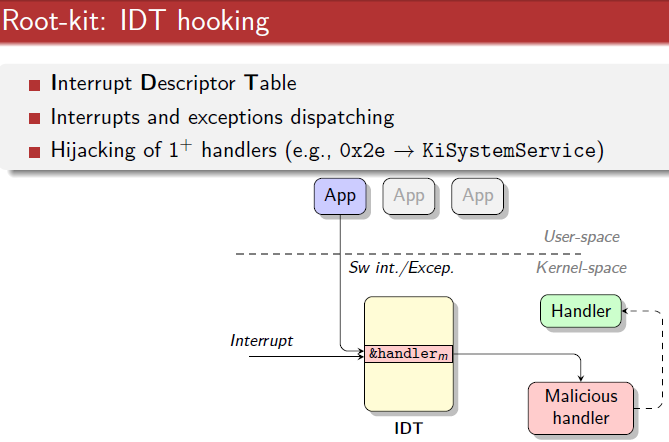
**Rootkits**

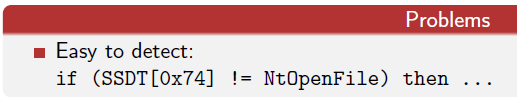
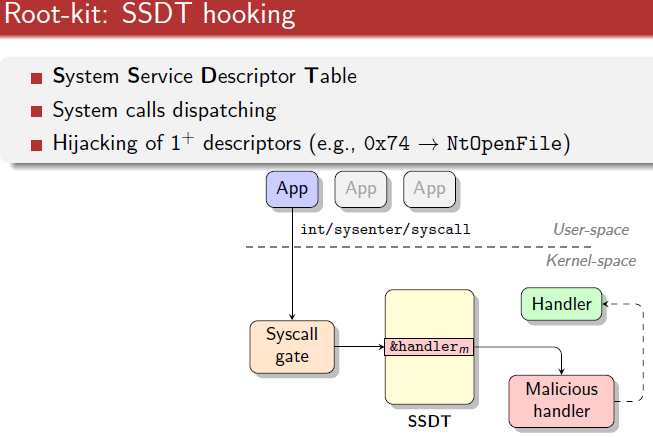
Hide

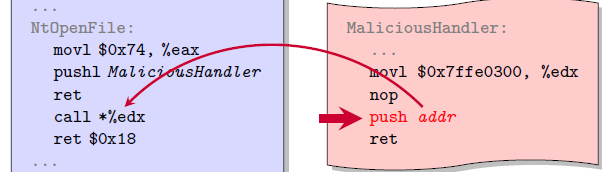
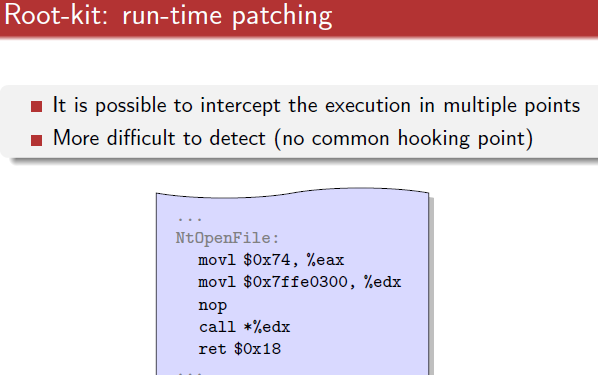
* Files
* Registry
* Services
* Network connections
* Processes

Hijack flow of exec by modifying code pointer

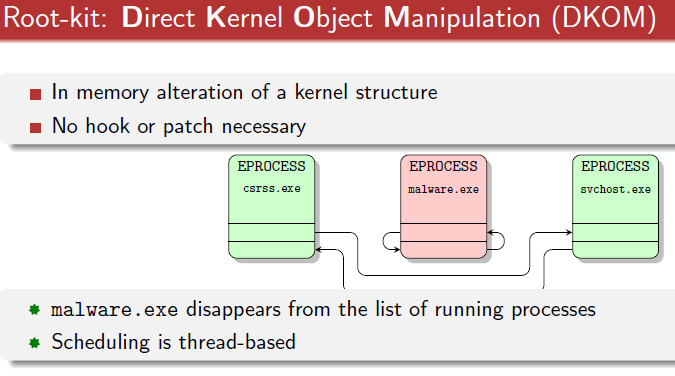
* User space: IAT
* Kernel space: IDT, MSR, SSDT
* VTX - At ring -1 (hypervisor level)
* VTX-root mode (ULTI!)







K-tracer – NDSS (identify interesting flows)

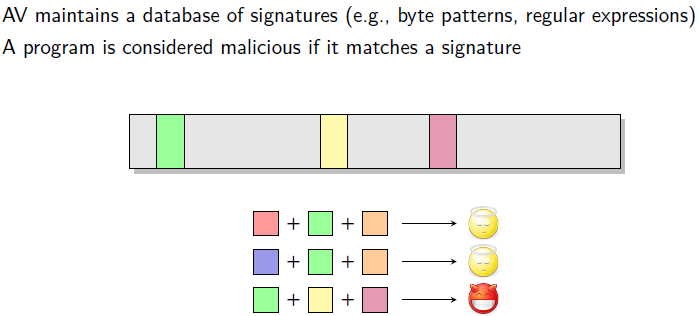
Static Analysis

AV

* No big effort to collect sample
* RE to statically analyse a sample
* Simple = sig-based static detection
  + Hash sign, md5

Early

* Byte-lvl / instruction-lvl signature
* Wildcard = regex
* Heurististics
  + Code exec at last section
  + Incorrect header size
  + Suspicious section name
  + Patched table of import func
* Assembly/C/macro
* Spread via file infection, network, USB
* SOLVED
  + Easy
  + Neither protected/obfuscated
  + Sig-based detection

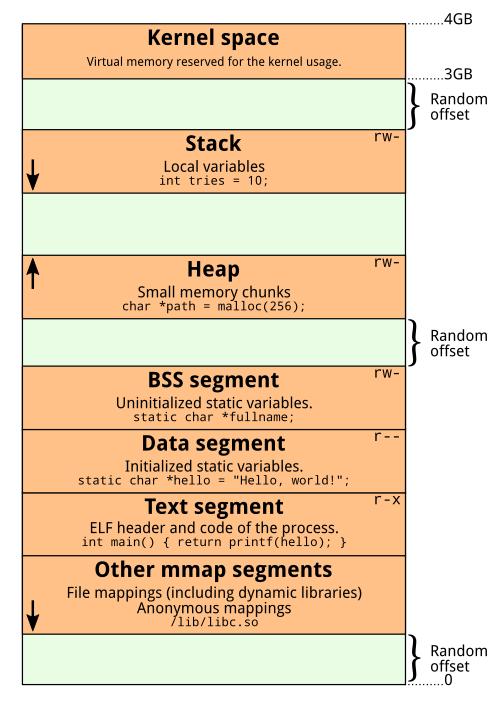


* NOW
  + Prevalent still
    - Reputation
    - String sig
    - Suspicious behaviour
    - Behaviour based

ASSEMBLY

* Low lvl symbolic lang
* Processor specific

Mem addressing

* User access = 32 bit addr
  + Flat mem (4GB)
  + Start = 0
  + End = 2^32 – 1
  + Legacy VA layout

User space

0xbfffffff

0xc0000000

Kernel space

.text 0x8048000  
.data

.bss

Heap(mmap)

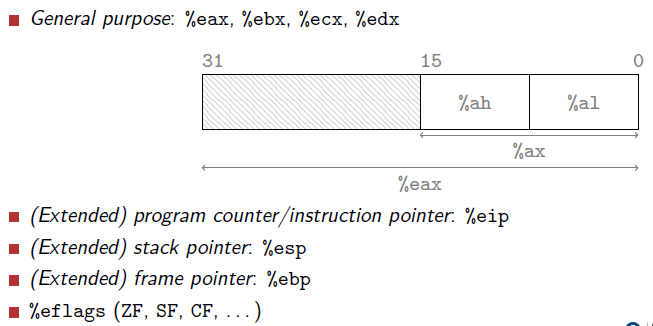
V(growing down)

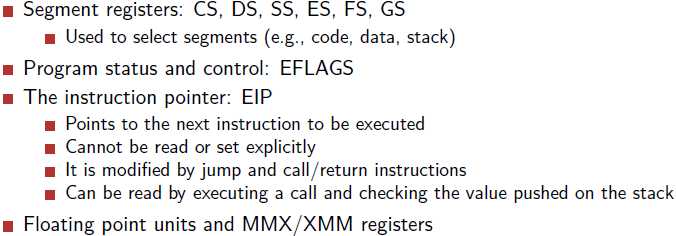
^ (growing up)

Stack

* Segmented mem model
  + Segment = Each addressable separately
* Page = 4K/4M pages

IA-32 Registers



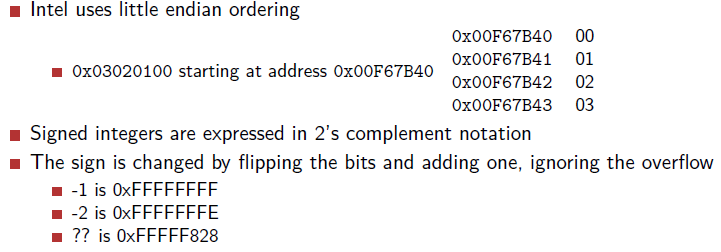


Change EIP value

* JMP
* RET
* CALL
* INT

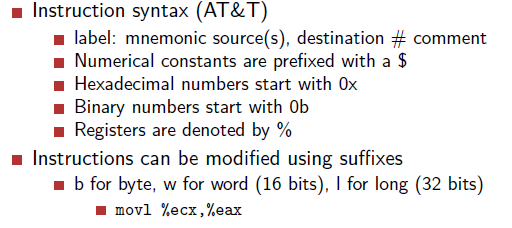
Endianess

* Little
  + 0x12345678  
    78563412 stored
* Big = 12345678 (easy)



Program =:

* Directives: commands for assembler  
  .data = section with variables
* Instructions: actual ops  
  jmp <addr>
* 2 possible Intel or AT&T
* Mov ebx, eax OR mov %eax, %ebx



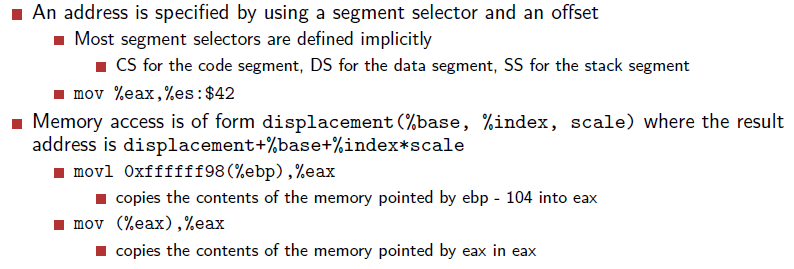
$ = constant/value, no $ = address

RW cannot same time, hence =

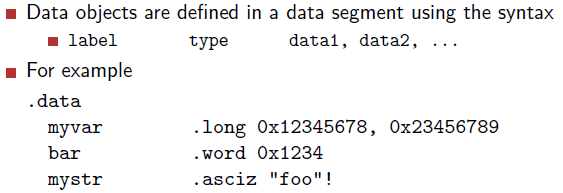
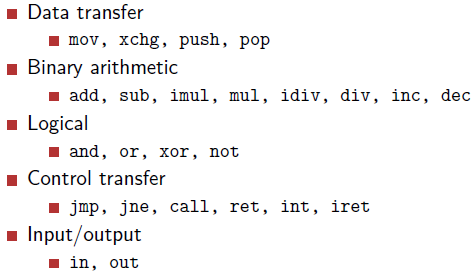
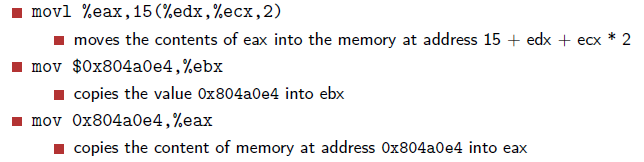
* Move value from 1 addr to another

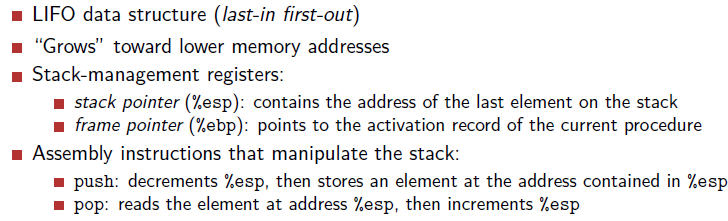
mov 0x1234, %eax  
mov %eax, 0x2345

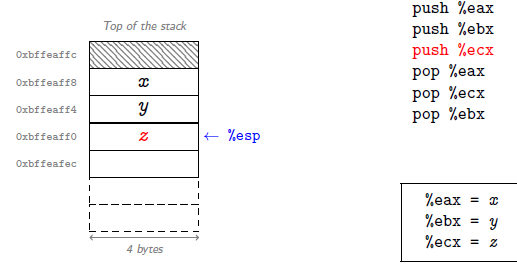
* Move addr into register  
  mov (%ebp), %eax



Disp = used usually in array to loop through stuff







**Prologue**

**push %ebp =** save %ebp on stack

**mov %esp, %ebp =** %ebp = %esp

**sub $n, %esp =** Allocate space for local variables

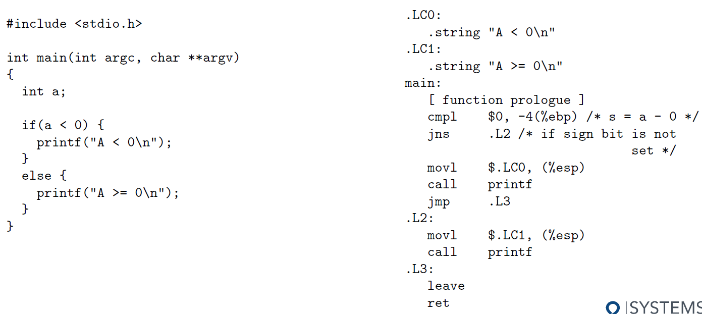
**Epilogue**

**leave =** place the %ebp and %esp

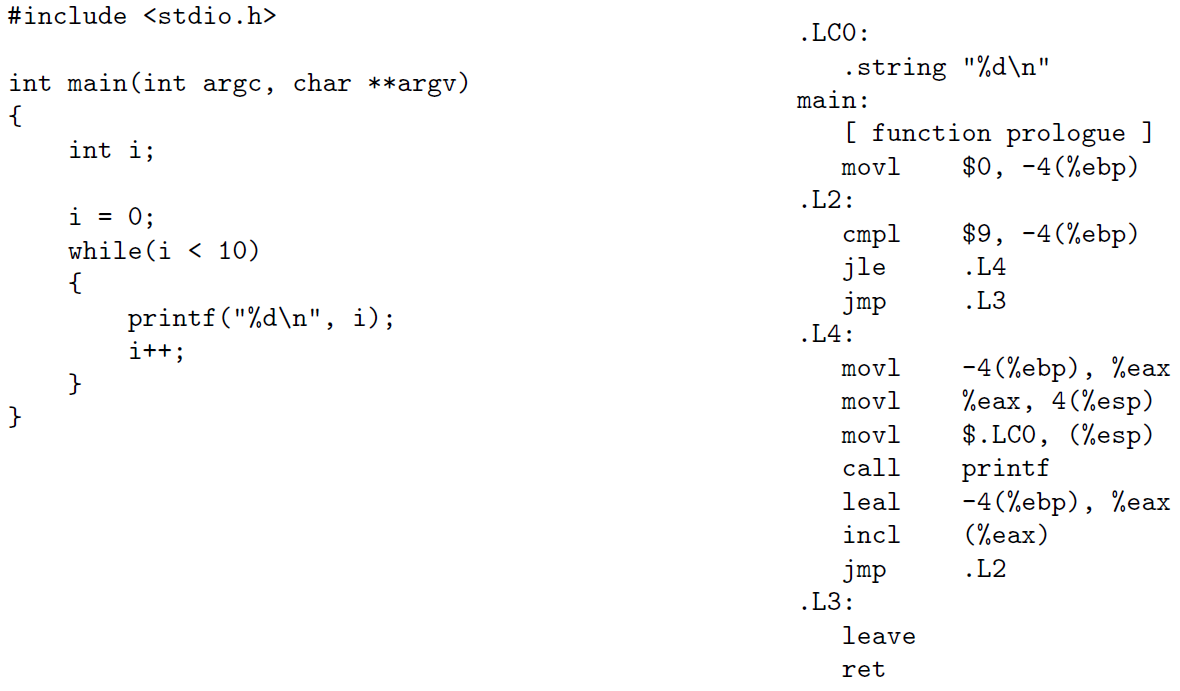
**mov %ebp, %esp  
 pop %ebp**

**ret =** jump %eip

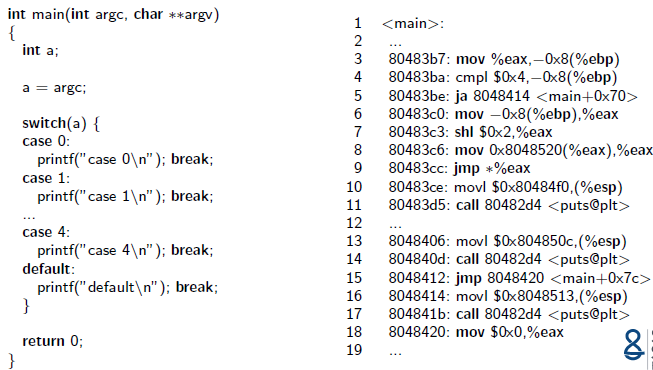
**CALL =** will push next instruction to the stack

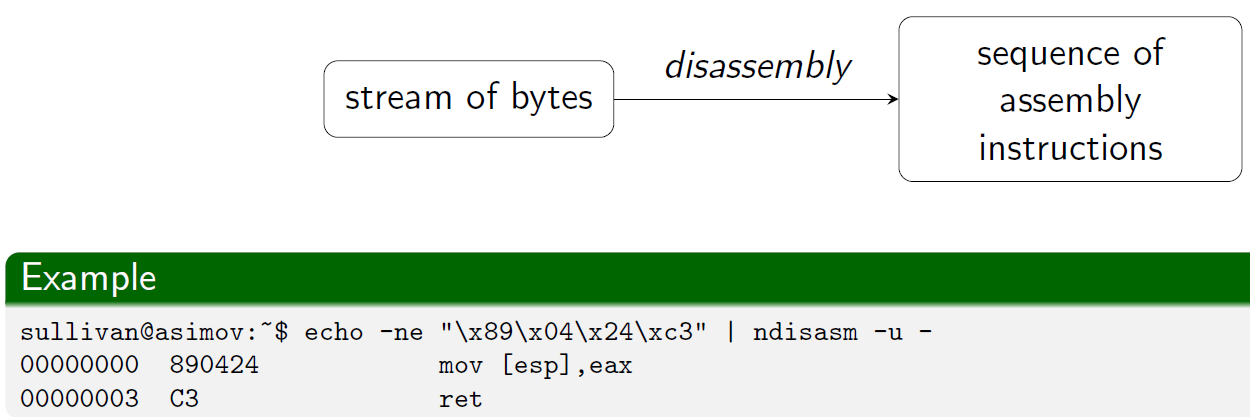
**IF** STATMENT****

**WHILE** STATEMENT



**SWITCH statement**

****



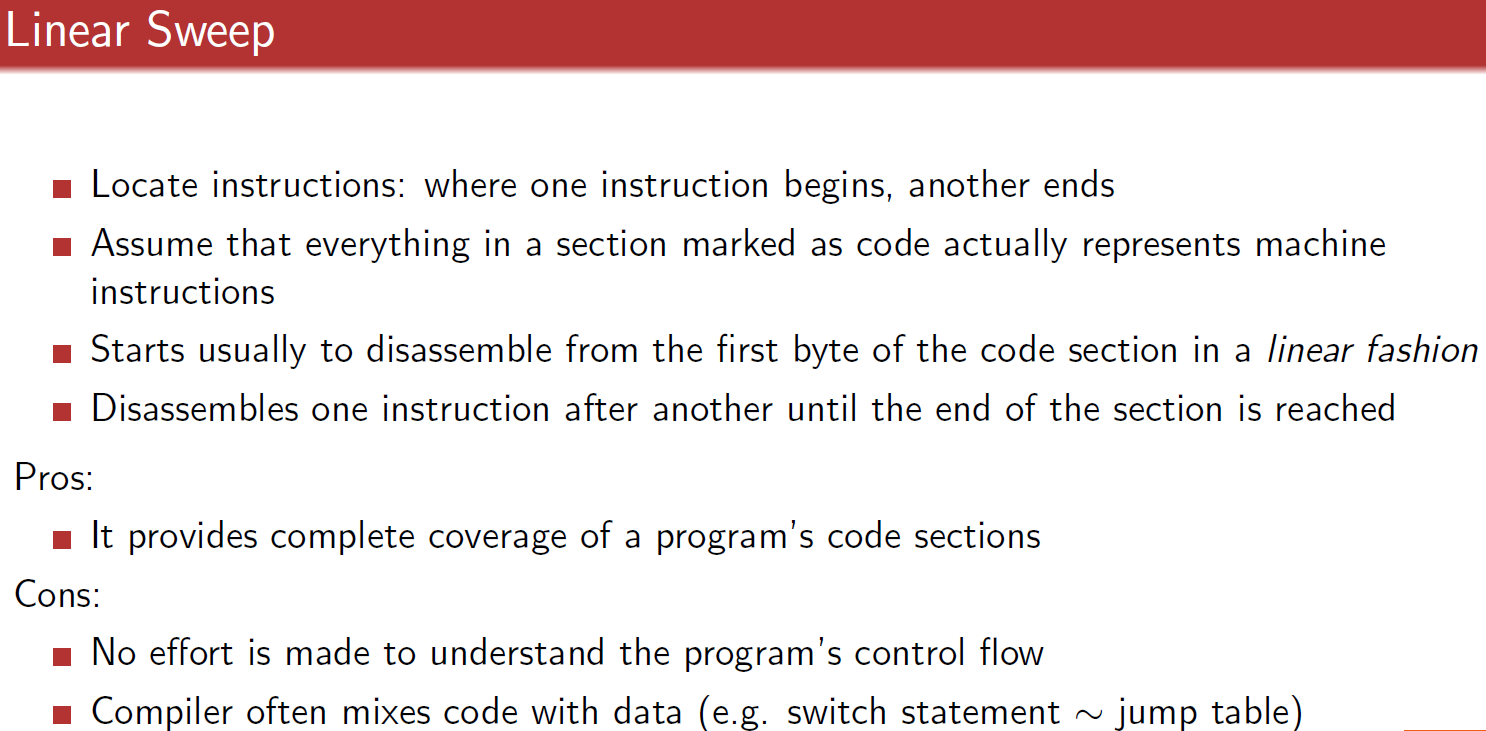
**Issues**

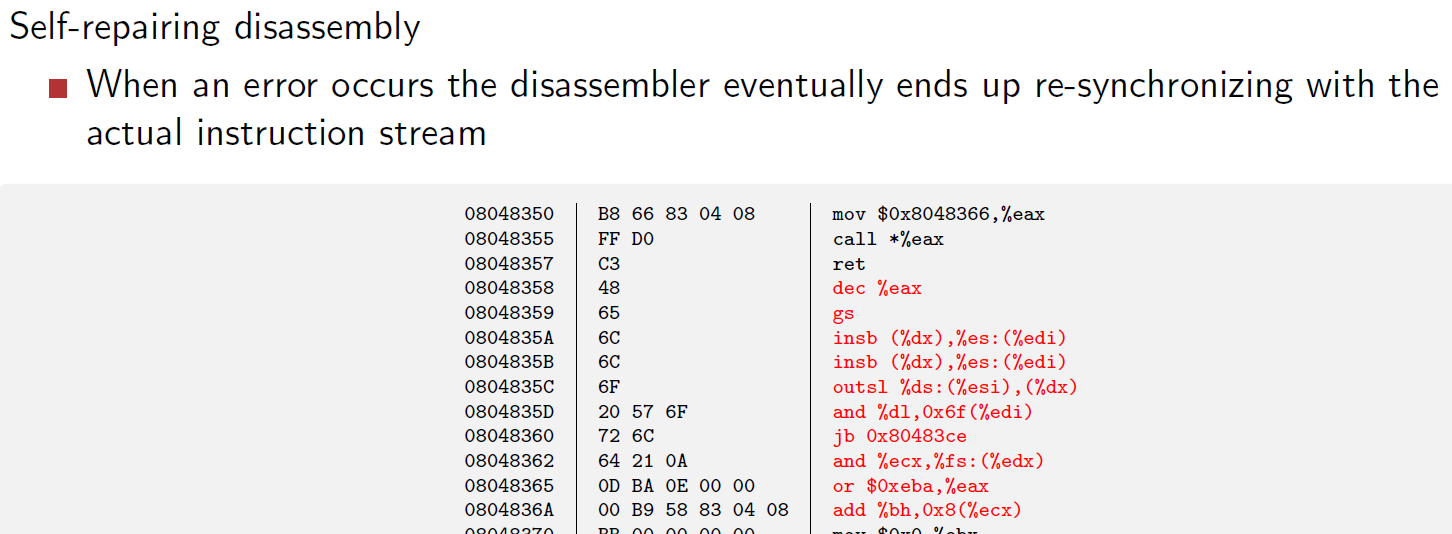
* Code and data in the same address space (how to distinguish them?)
* Variable-length instruction
* Indirect control transfer instructions
* Basic blocks
* At compile-time some information may disappear, e.g.
  + Variable names
  + Type information
  + Macro & comments
* How to identify
  + Functions?
  + Function parameters?

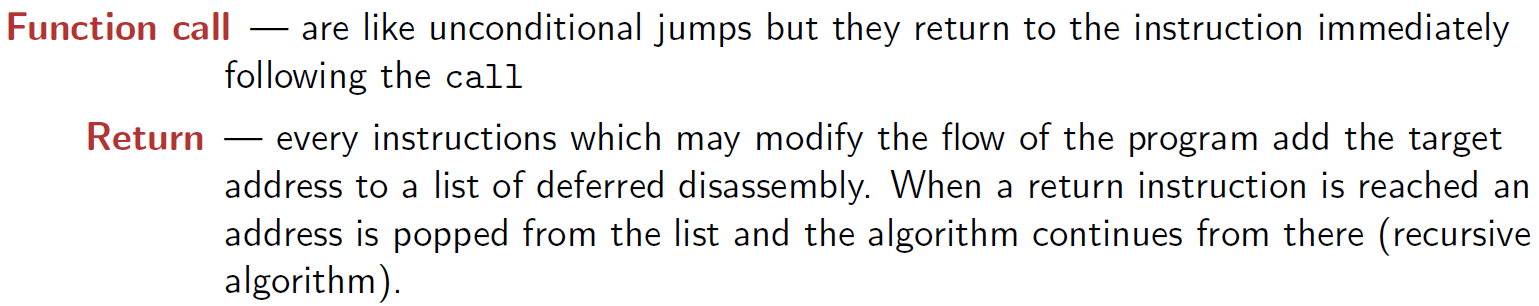
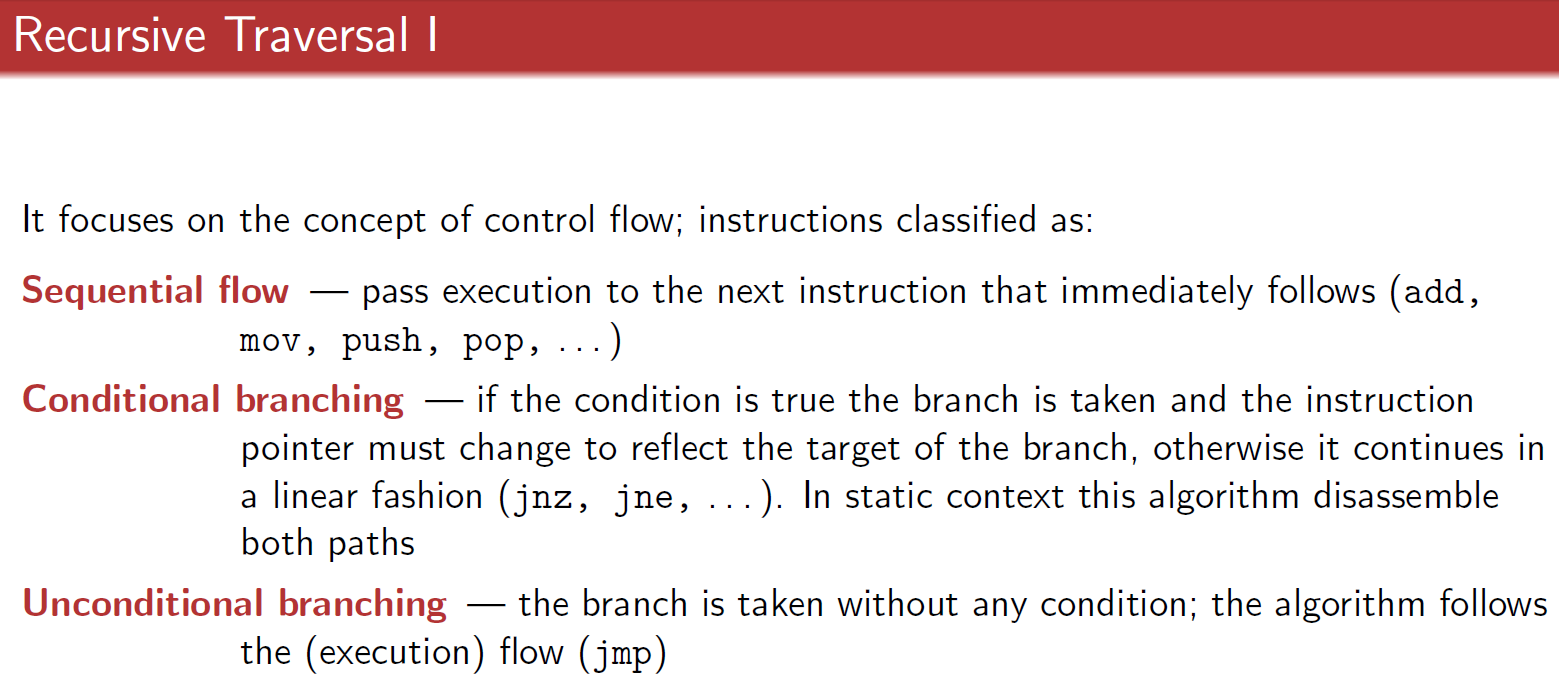


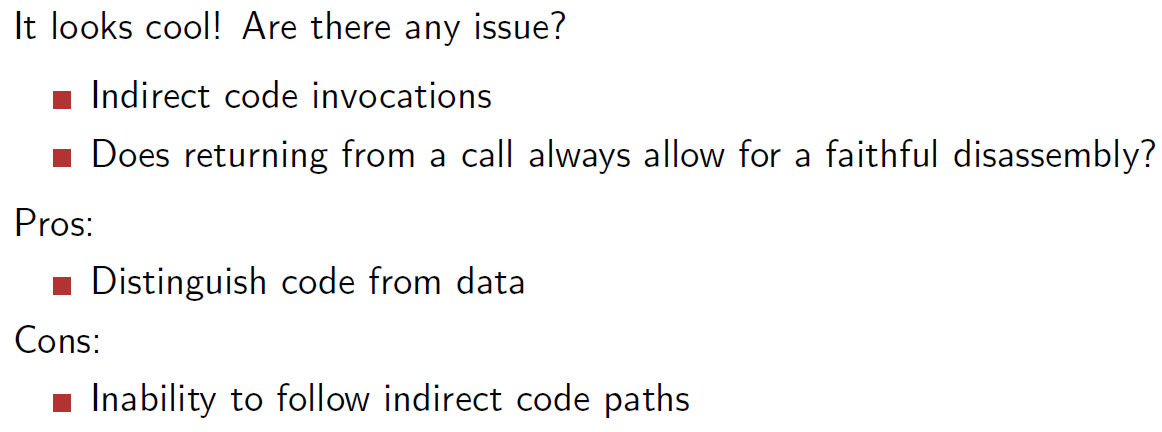
**Two main algorithms**

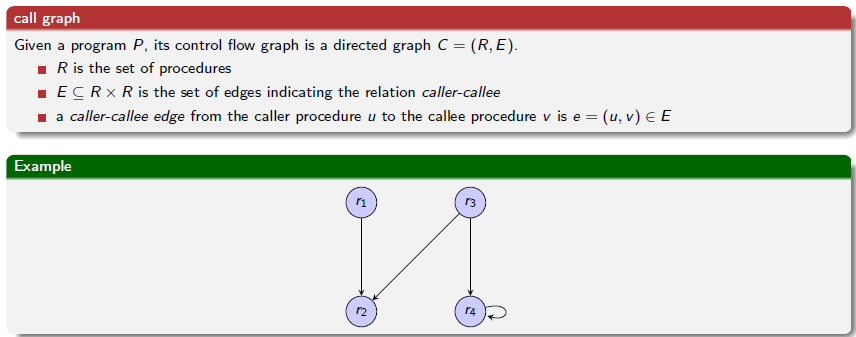
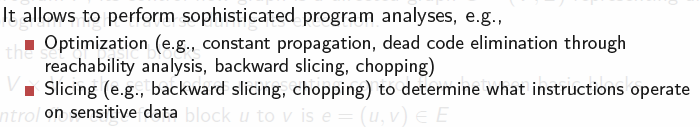
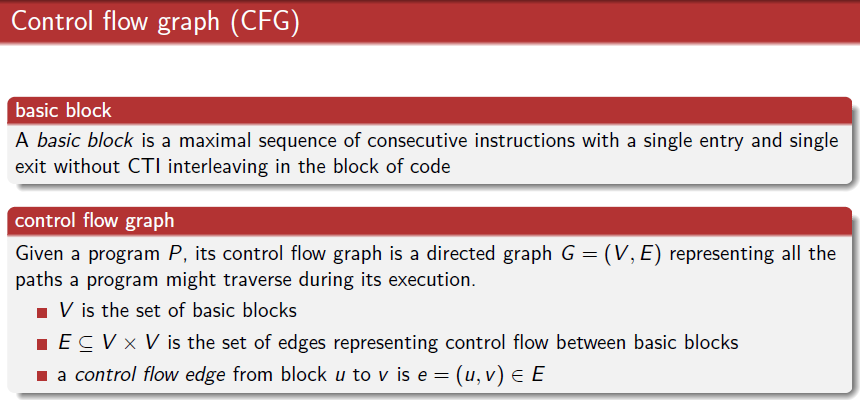
* Linear sweep
  + Objdump
* Recursive traversal
  + IDA Pro

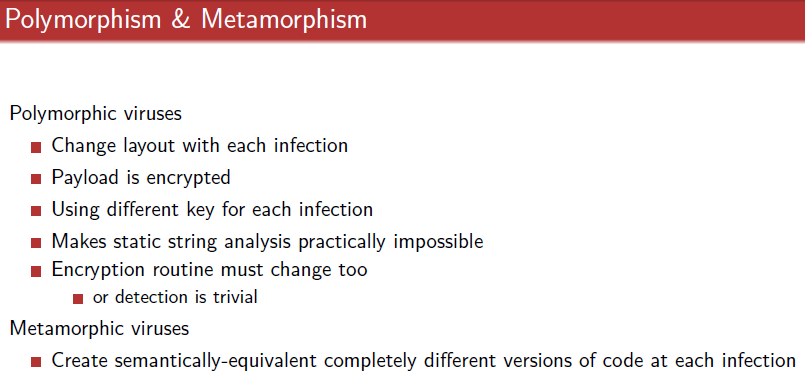


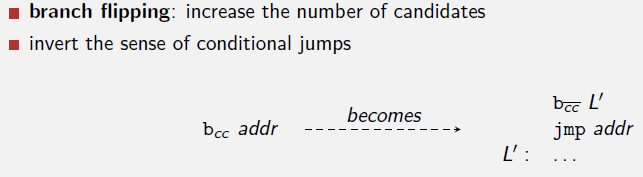
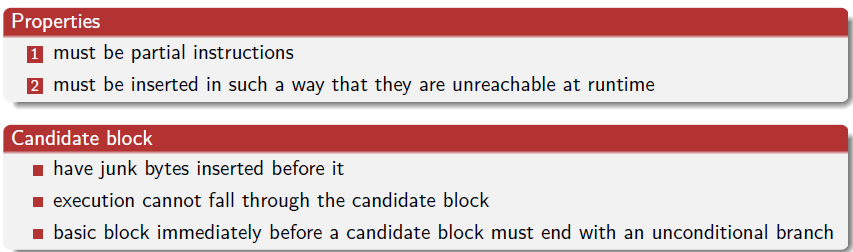
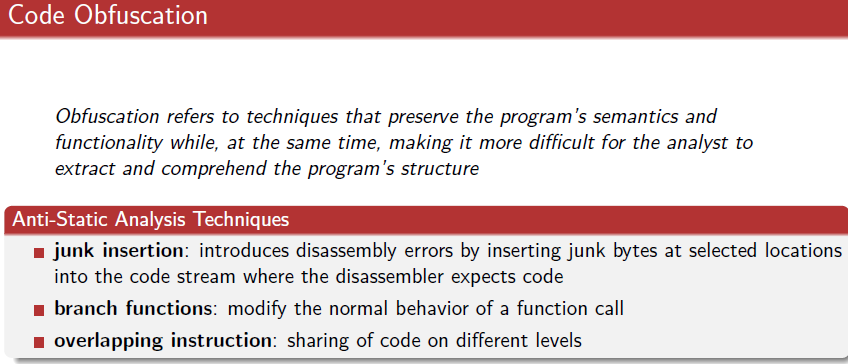


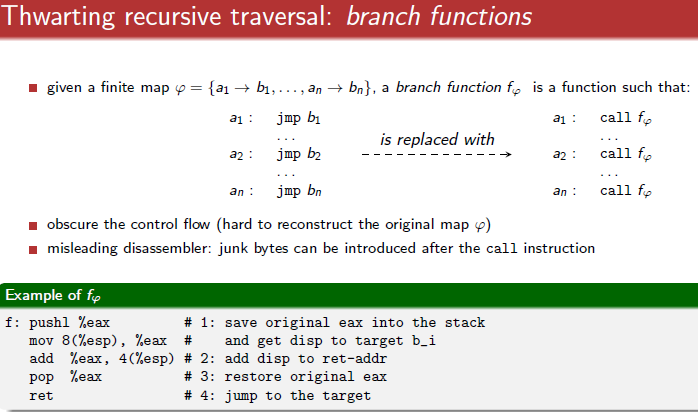
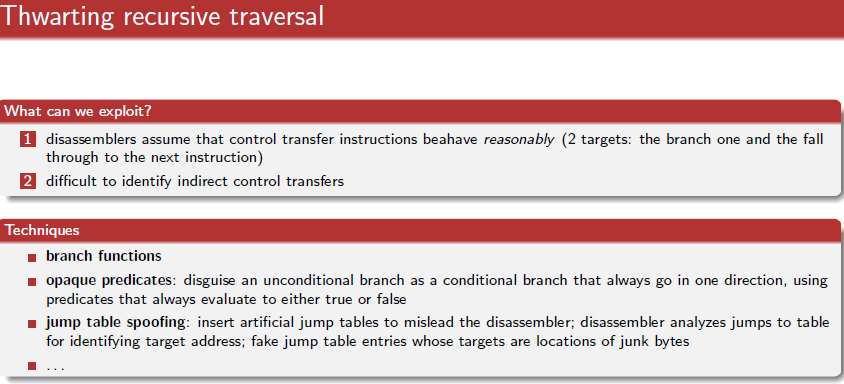


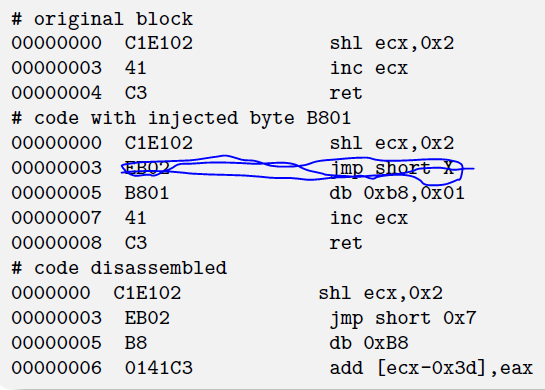


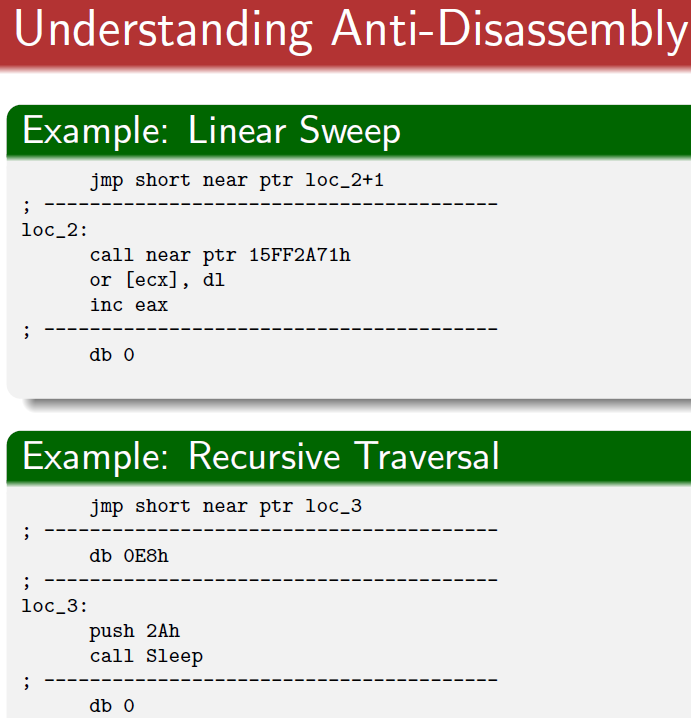
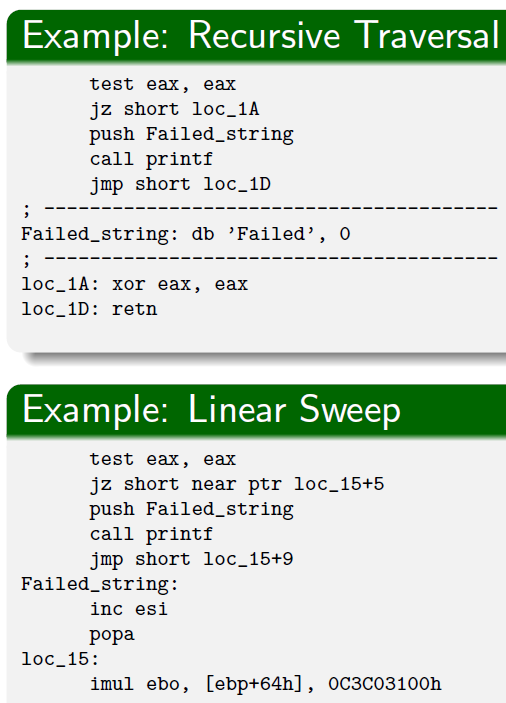
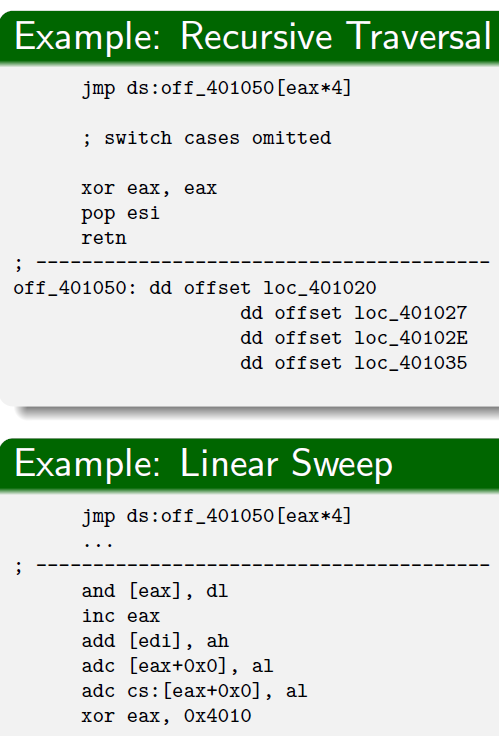
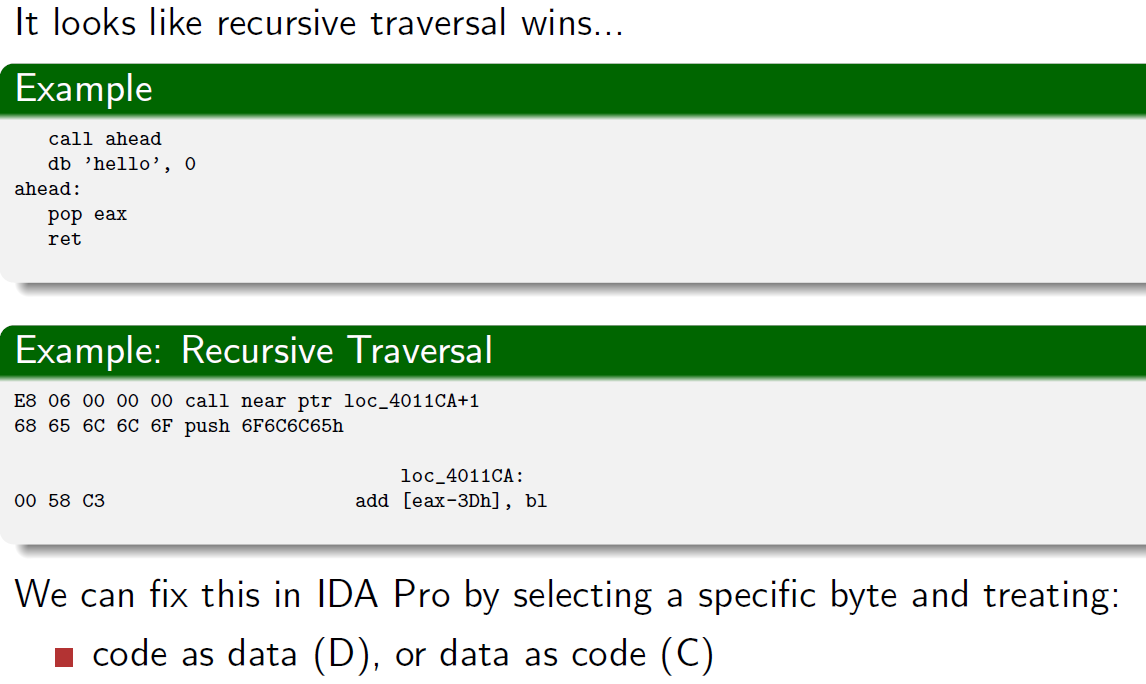
****

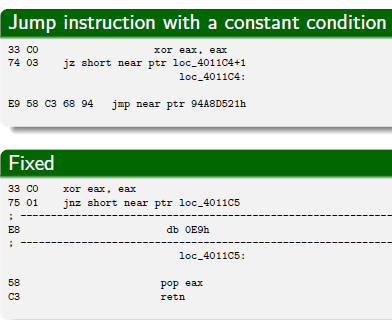
****

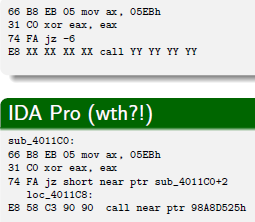
****

****

**Code overlapping**

********

**will sure follow the path**- if loc+1, it won’t be “jmp near ptr”****

-only 1 version of the instructions,   
below = will have 2 versions  
****

**Malware on Smart Devices**

**Increase in Platform adoption**

* Exceeds num of personal com
* Android = main market share

Move from Symbian to android (malware)

Malware As a Service = Cost(atk)<Potential Rev

Rise = privacy problems

* Leaking of private info
* Traceability
* Cyber crime

Security Features

* Market Protection -implemented in market lvl
* Platform Protection
* Enhancing proposals –from academics

Market protection

* Application review (analyse submitted apps)
* Application signing (authorship, integrity chk)
* Official market = a lot of consumers, but need to wait, sloppy in analysis

Platform Protection (Permissions)

* Effectiveness = good, but…
* Apps over privileged
* Increase in consumer exp

Platform Protection (Sandboxing)

* Ineffective = if user overlook permissions
* Do not prevent apps from exploiting sys, kernel vuln
* Propose use of Hypervisor (heavy)
* Virtualisation = trade off security VS performance

Platform Protection (Interaction)

* IPC = intercepted/stopped/replaced
* Allows collusion attack
  + >2 mal app can collude, to violate permissions
* Activity hijacking

Platform Protection (remote management)

**Enhancing Security Model**

* Rule driven policy (richer language, detailed permissions)
* High level policy
* Platform hardening
* Multiple users

Malware

Increase

* Evolution – annoyance to complex
  + PC, palm malware
* Characterisation – multipurpose, profit, sabotage, espionage
* Examples- piggyback malware, botnets, root-exploits

Evolution

* Mobile Malware
  + New infection vector (3G, wifi, edge, BT)
  + Pay-per-use, premium rate SMS
* Smartphone
  + steal personal info
  + over wide range of mobile platform (ios, android)
* Future
  + IoT, smart tv
  + Medical, implantable medical devices
* Challenge =complex, reuse-oriented pcs of SW

Characterization

* Traditional = virus,Trojan,spyware
* New, depends on
  + Attack goals, behaviour
  + Distribution & infection
  + Privilege acquisition

Malware detection

* Monitoring, analysis, identification
* Types of detections (Static,Dynamic)
* Type of monitoring (kernel, user)
* Granularity of detection (per app, per device, group of apps)
* Type of analysis (expert learning, machine learning)
* Type of identification (anomaly, misuse, specs)
* Place of monitoring, identification(cloud, phone)

Monitor-able Features

* Comms, sensor, systems, hardware, user
* Calls, GPS, camera
* Diff features= diff types of malware

Market Protection (Static Analysis)

* Droid Sieve: Automated Android Analysis

Permissions and API calls

* + Effective for malware fingerprinting
  + Obfuscation
    - Encryption, reflection, native code, hidden code

Taxonomy (meta information, types of resources, certs, incognito features)

Vuln to motivated adversaries

Market Protection (Dynamic Analysis)

* Virtual environment

**Static analysis in Android**

Dalvik

* Highlevel = java (dalvik bytecode)
* Different from Java bytecode
  + Register based (DEX)
  + Stack based (class)
* Optimization(space, less instructions, lots of semantics)

**Java SourceCode -> Java ByteCode -> Dalvik Bytecode**

**JavaByteCode -> javap -> Java Assembly**

Method info, definition

Types

* Reference types
  + Objects/Lib = Start with Lpackage/name/ObjectName
  + Array = [I = int[], [[I = int[][], [Ljava/lang/String
* Primitive
  + V = void, Z = Boolean, C= char, J =long

Code

* Dalvik opcode = 218 types
* Interesting = 6E invoke-virtual
* Long instructions

Dalvik

* Register based
* 2^64 registers

Stack based: JVM

Register based : Dalvik (faster)

Disassembling (smali language)

Dexdump classes.dex

Assets folder (check files in assets folder)

Apktool (decode/build)

View AndroidManifest.xml (note package name, activity, intents, permissions, intent-filter)

Jd-GUI

Honeypot

Rogue AV distribution infrastructure   
(fake affected system, install fake AV)

Botnet target specific IP range, but all connect back to same C&C

More targeted targets (stuxnet)

Main message = more complicated,  
last time = just plant honeypots around

Impact of change:

* Good enough is not enough
* How to detect what I can’t see

Traffic -> suspicious artifacts -> analysis -> take (malware traffic, C&Cs, behav info)

Automation malware analysis (sandbox)

* Dynamic, run.
* Use full system emulation (see every instruction)
* Taint analysis (data flow)
* Monitor system calls

High res anaylsis

* C&C traffic
* Replacing image of another process
* Disabling system restore
* Wiping out pe image in memory
* Deleting volume shadow copies

Evading dyn analysis

* Sandbox = popuplar
* Code executed, malware authors hv control
* Do nothing during sandbox

Evasion

* Detect virtualized env
* Detect signs of specific analysis env

(geoloc, specific OS artifacts)

* Avoid being analysed

(requires human, timeout analysis)

Check if username = andy (simple loophole)

Even stalling can be mitigated

* Program does not make progress
* Passive mode (find loop, reduce logging\_
* Active mode (reduce logging is not sufficient, interrupt loop)

Packets

* Detect patterns
* User agents, headers

Block either domain-names/ip would block.

Detect a DGA

DGA reversing

Fast flux, (domain name, to a lot of ip)

* Instead use lousy webserver

P2p ring = comm adds resiliency, if u join botnet, joins the p2p ring

* Example = zeus gameover
* Servers that populate the p2p ring
* C2C down, can derive a new C&C
* Can use fake node to fake a new C&C server

Can use simple mistakes to detect custom clients

* E.g. missing a simple ‘space’
* Custom user-agent

Detect HTTP, data exchange

* Put specific string (username/password) for the application to steal and detect them

If use custom encryption (hard to revert, easy to spot)

* Size and metadata of message flow
* First message = size of XXX
* Second message = size of XXX

If use SSL/TLS (easy to implement, easier to study/FP)

TLS C&C

* Auto gen cert
* Easy setup, achieve large infra
* BUTTTT
* TLS, stripping
* TLS cert reputation

TLS blacklist certs

* Sslbl.abuse.ch

Reputation

* Server used for several malware to do reflection/teslacrypt-checkin

Suricata rule

Havent seen before

1. Use reputation   
   (use bad servers to detect new threats)
2. Threat hunting

(large scale data collection, concept of interesting activity, tools to correlate/dig info)

**Dynamic Analysis**

Wants to look at interesting behaviour, not everything

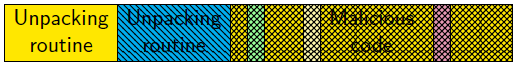
**Limits of Static Analysis**

* Opaque predicates: condition outcome already set, hard to deduce using static (complex CFGs)
* Anti-analysis: anti disassem, CFG flattenting, packing (incomplete CFG)
* Indirect calls/jumps (partial CFG exploration)

**DYN = can observe ACTUAL executions**

Packing = mal code (hidden by compress/encrypt)

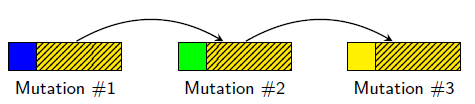
Decompress/decrypt at runtime



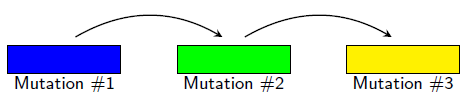
**Algo unpacking**

* used in AV routine
* Recover original code
* A lot of packers families

Polymorph (semantics same, after unpacking)

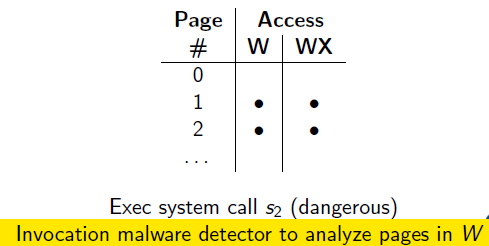
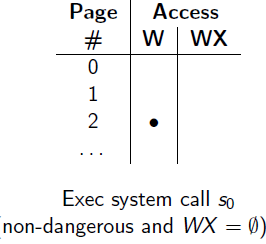
****

Metamorph (whole payload change)

****

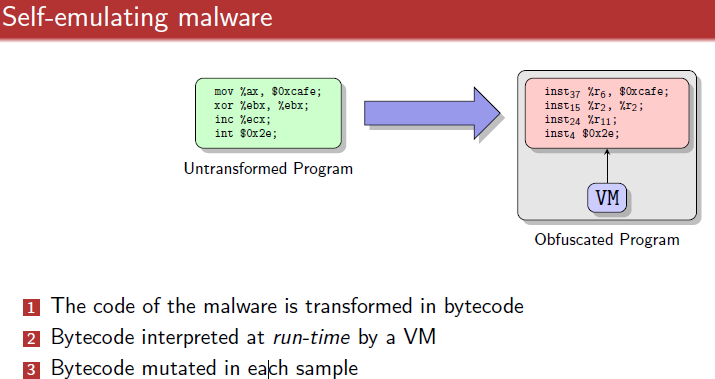
Algo-agnostic unpacking = DYN ANALYSIS

* Emulate/tracing of sample execution until termination of packing routine

****

Dangerous syscalls = activate AV to check**Heuristics =** detect end of unpacking (based on exec of written code)

Hard to differentiate (malware app, JVM) running



Packer = can be used for good apps

Signature != sufficient

Behavior-based malware detection

* Signature, static detection easy to defeat

Moving -> dyn behaviour-based tech

* Monitor events characterize exec of prgm
* Infer behaviour of program
* Detect high-lvl mal behaviour
* Can detect novel malware – share same high lvl behaviour

DYNAMIC TECH

* EXECUTED + MONITORED
* INTERACTION with ENV
* INTERACTION with OS

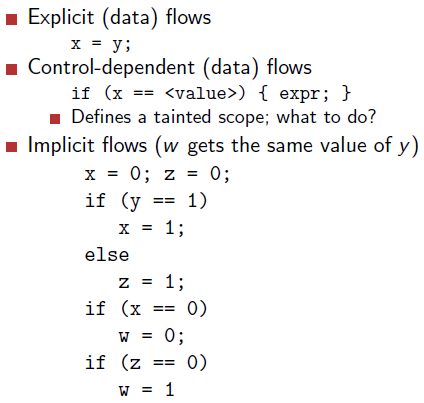
DYNAMIC ANALYSIS

* Abiiity to monitor code (data flow) as execution
* Allow reasoning about execution
* Allow perform security analysis (run time info)
* Debugging
  + Finding bugs
  + (whats going on) – sw interrupt, API tracing
* Instrumentation
  + Add extra semantic-preserving code to program/process
  + Taint tracking

TAINT ANALYSIS

* Track how INTERESTING data flow thru prgm
* **Taint SOURCE**
* Propagation rules (how-to)
* **Taint SINKS** (where data flows to, allow security policy)
* **TAINT SCOPE** (if tainted(x), then change z value)
* **TAINT LEAK =** miss out taints

= Basis of many DYN behaviour malware analysis framework

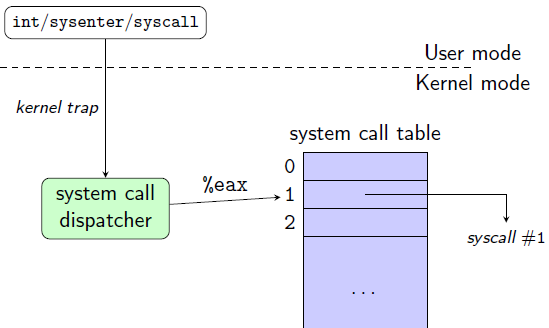
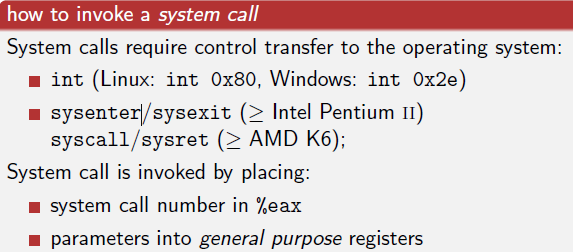


**Interaction with env**

* Lsof
* Netstat
* Ltrace = intercepts and records dynamic lib-calls

System calls

* Only kernel has priv to do
* Provide useful info about process behaviour
* N-gram analysis -> on seq of sys calls



Strace = intercepts + records system calls

* Trace syscall, param, signals, child processes

Behavior-based malware detection

* Fine grained = too much for endhost
* Coarse gained = at end host

Sandbox, Debugging

* Step by step
* Breakpoint
* Watchpoint (value of var changes)
* Catchpoint = (particular event occurs)

Anti-debugging

* Int3 & SIGTRAP
* Checksum of code = (breakpoint)
* Ptrace() = ptrace yourself (cos u cannot ptrace a ptraced process) if error = ptrace active
  + Redefine ptrace return val to always 0
  + Set EAX to zero

Use sandbox

GOALS of dyn-analysis

* Visibility = sandbox see as much as possible
* Resistance to detection
* Scalability = analysis must scale up

Emulator = software program simulates functionality of program/hardware (reconstruct sys calls)

* P runs on emulated hardware = System collects detailed info about exec of P
* Detect evasion attempt
* Performance penalty

Virtualization = P runs on hardware

* Software = control + mediates access of program
* VM = independent, isolated
* Execution = use actual physical resources
* Bad = hard to hide hypervisor to mal code
* Good speed

Implication of WHOLE SYS EMULATION

1. Can install + run an actual OS on top of emulator
   1. Malware exec top of real OS
   2. Fingerprinting = hard
2. Interface of processor = simpler than interface provided by modern OS
3. System emulator = great visibility
   1. Visibility and mem access = ability to reconstruuct low & high semantics
   2. Challenges =
      1. semantic gap (reconstruct sys call) = VM introspection
      * attribute process to sys call
      1. performance