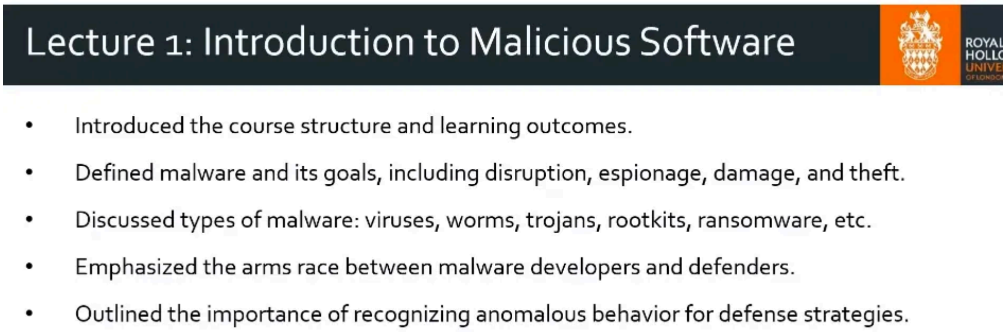
# Lecture 1

# Introduction to Malicious Software




## Malware Definition and Goals

- **Malware**: Software designed to violate a system's security policy
- Goals include disruption, espionage, damage and theft
- Examples shown through malicious scripts that create unauthorised privileges

## Malware Taxonomy

### Types of Classification

- **Functional**: Based on distinguishing features (virus, worm, etc)
- **Behavioural**: Based on exhibited behaviour
- **Authorship**: Based on creators/tools used (focuses on attribution)

### Classification Units

- **Malware types**: Broad categories (worms, viruses, trojans)
- **Malware families**: Specific groups (GandCrab, Ryuk, Sodinokibi)
- **Samples**: Specific instances with unique signatures

## Major Malware Types

- **Trojan Horse**: Program with both an overt (documented) and covert (hidden) purpose
  - Often uses command-and-control servers
- **Rootkit**: Pernicious trojan that hides itself on systems
  - Changes system reporting programs
  - Can operate at kernel level
  - Difficult to detect using standard tools

- **Virus**: Program that inserts itself into files and performs actions
  - Has insertion and execution phases
  - Types include:
    - Overwriting viruses
    - Companion viruses
    - Parasitic viruses
    - Memory-resident viruses
    - Boot-sector viruses
    - Multi-partite viruses
    - File infectors (including macro and script viruses)
- **Worm**: Self-replicating program that copies between computers
  - No need for human interaction
  - Can spread exponentially (e.g., Code-Read infected 359,000 computers in <14 hours)
- **Other Types**:
  - **Downloaders/Droppers**: Download or extract additional malware
  - **Backdoors/Remote Access Tools (RATs)**: Bypass authentication
  - **Rabbit viruses**: Consume all resources
  - **Logic bombs**: Trigger on specific events
  - **Spyware**: Record user information
  - **Botnets**: Networks of infected computers
  - **Ransomware**: Inhibits resource use until payment
    - Locker-Ransomware: locks computer to prevent access
    - Crypto-Ransomware: encryption of files to make them inaccessible
  - **Wipers**: Destroy data
  - **Cryptominers**: Use resources for cryptocurrency mining
  - **Grayware**: Annoying but less serious than malware
    - **Adware**: Display advertisements, often targeted

# Defense Strategies

- Emphasised recognising anomalous behaviour
- Ongoing arms race between developers and defenders

# Lecture 2

# Anatomy of a Binary

## Lecture 2: Anatomy of a Binary

- Explained how source code in C is compiled into binaries through preprocessing, compiling, assembling, and linking.
- Discussed ELF (Executable and Linkable Format) and its structure.
- Introduced basic assembly concepts and disassembly of object files.
- Covered how binaries are loaded and executed in memory and how to interpret binary contents for malware analysis.

## The C Compilation Process

- **Four phases of compilation**:
  1. **Preprocessing**: Expands directives, macros
  2. **Compilation**: Translates to assembly language
  3. **Assembly**: Converts to object files (machine code)
  4. **Linking**: Combines object files into exectuable
- **Object Files vs Executables**:
  - Object files are relocatable (not bound to specific addresses)
  - Executables are ready to load at a particular memory address
  - Static libraries merge into binary
  - Dynamic libraries resolve at runtime

## ELF (Executable and Linkable Format)

- Standard binary format on Linux
- Structure includes:
  - Executable header (first)
  - Program headers
  - Sections
  - Section headers (last)

### ELF Components

- **Executable Header**: Describes format and structure

    - Contains "magic value" (0\x7f followed by "ELF")
    - Specifies entry point address
- **Section Headers**: Describe contiguous, non-overlapping chunks of code/data
    - `.init` : Initialisation code
    - `.text` : Main program code
    - `.data` : Initialised variables
    - `.bss` : Uninitialised variables
    - `.rodata` : Read-only data (constants)
- **Program Headers**: Used by OS for loading and execution
    - Define segments for runtime
    - Map sections to memory segments

# Binary Loading and Execution

- OS sets up process with virtual address space
- Interpreter (e.g., ld-linux.so) loads binary
- Controls transfers to interpreter which handles relocations
- Then jumps to program entry point

# Assembly Language Basics

- **Registers**: Small storage locations on CPU
    - General purpose (rax, rbx, etc.)
    - Special purpose (rip, rflags)
- **Common Instructions**:
    - Data movement: mov, xchg, push, pop
    - Arithmetic: add, sub, inc, dec, neg
    - Logical: and, or, xor, not
    - Comparison: cmp, test
    - Control flow: jmp, call, ret
- **Stack Operations**:
    - LIFO (Last In First Out) structure
    - Used for function calls, local variables, return addresses
    - Frame pointers (rbp) and stack pointers (rsp)
    - Function prologues and epilogues

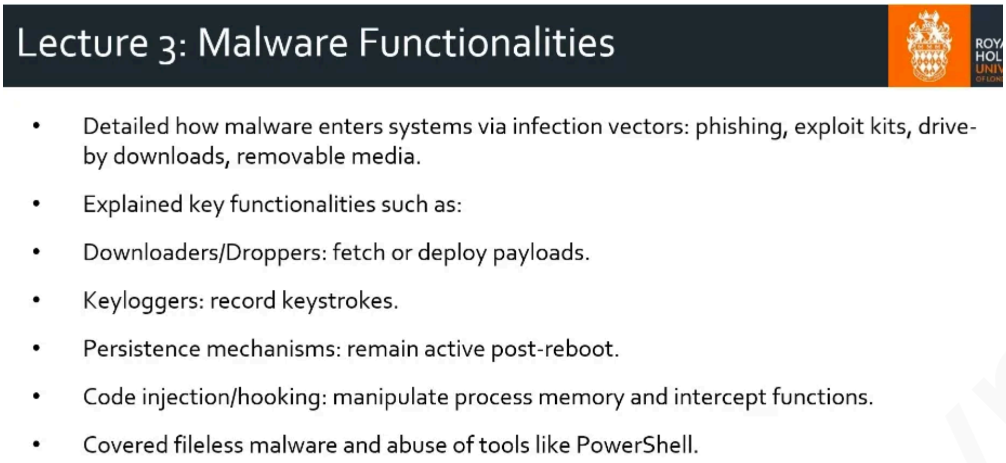
# Binary Analysis Challenges

- Lack of symbolic information
- No type information
- No high-level abstractions

- Mixed code and data
- Location-dependent code and data

# Lecture 3

# Malware Functionalities




## Infection Vectors

- **Phishing**: Impersonating legitimate entities (to obtain information)
  - Homograph attacks (using similar-looking characters)
  - Spearphishing (tailored for specific victims)
  - Spam email with malicious links/attachments
- **Web Vulnerabilities**:
  - Malvertising (malicious advertising)
  - Compromised websites
    - SQL injection, XSS
  - Drive-by downloads (unintentional download of malicious code)
  - Watering hole attacks (infecting sites visited by targets)
- **Common Delivery Channels**:
  - Windows macros and scripts
  - Exploit kits: all-in-one tool to launch exploits against vulnerable programs
  - Fileless malware: misuses existing utilities to avoid detection

## Malware Components and Functionality

- **Downloader**: Downloads additional malware from internet
- **Dropper**: Embeds and extracts additional malware components
- **Keylogger**: Intercepts keystrokes
  - Methods: GetAsyncKeyState(), SetWindowsHookEX()
- **Replication**: Spreading mechanisms

    - Via removable media
    - Network propagation
- **Command and Control** (C2):
    - Communication with attacker-controlled servers
    - Protocol types (IRC, HTTP/HTTPS, P2P, DNS tunneling)
    - Botnet structures (centralised, hierarchial, peer-to-peer)
- **Persistence Mechanisms**:
    - Registry modifications
    - DLL search order hijacking
    - COM hijacking
    - Creating services
    - Startup folder items

# Code Injection Techniques

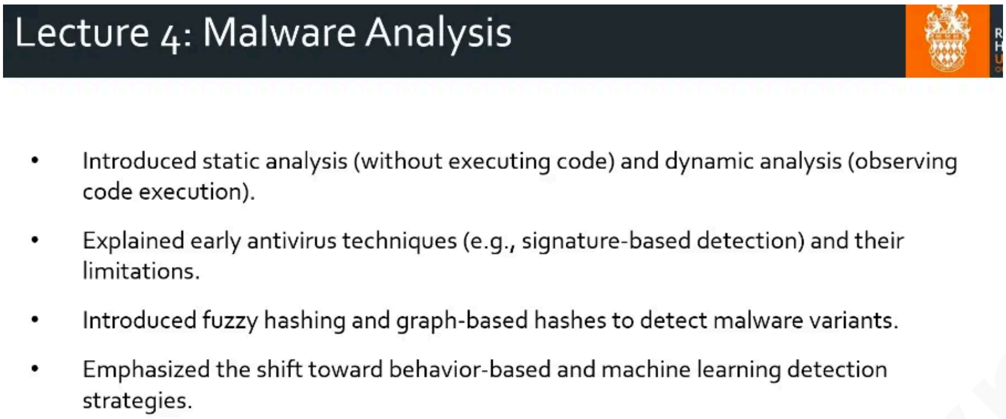
- **Process Injection Methods**:
    - Remote DLL Injection
        - Target process forced to load malicious DLL into memory space
    - Remote Executables/Shellcode Injection
        - Malicious code injected directly into memory with no trace on disk
    - Hollow Process Injection
        - Executable section of legitimate process is replaced with malicious version
    - Code Injection via Buffer Overflow
- **Hooking Techniques**:
    - IAT Hooking (Import Address Table)
    - Inline Hooking

# Fileless Malware

- Uses existing utilities to avoid footprints
    - "Living off the land"
    - Uses PowerShell, WMI, registry
    - Resides in volatile memory
    - Harder to detect with traditional methods
- **PowerShell** commonly abused:
    - Provides access to OS functions
    - Leaves few traces
    - Can execute code directly from memory

# Lecture 4

# Malware Analysis




## Early Malware Analysis Approaches

### Early Days

- Minimal effort to collect samples
- Manual reverse engineering for analysis
- Simple signature-based detection was effective
- Used hash signatures (e.g., MD5) for identification

### Traditional Malware Characteristics

- Written in assembly/C/macro code
- Spread via file infection, network, or removable media
- Typically unprotected and non-obfuscated
- Easily detected with signature-based methods

## Signature-Based Detection

### Types of Signatures

- **Byte-Stream signatures**: Specific patterns of bytes
  - Simple but prone to false positives
  - Easily evaded with minor changes
- **Checksums** (e.g., CRC32):
  - Applied to byte-streams
  - Weak against collision attacks
- **Cryptographic hashes** (e.g., MD5, SHA):

    - More resilient against collision attacks
    - Easily defeated by small file changes
- **Fuzzy hash functions**:
    - Detect groups of similar files (same malware family)
    - Use locality-sensitive hashing (LSH)
    - Allow for detecting variants with small changes
- **Graph-based hashes**:
    - Computed from call graphs or control-flow graphs
    - Time-consuming signature generation
    - Growing database size
    - Easily defeated by code protection techniques

# Static Analysis

## Processes & Challenges

- Extracts properties without executing code (over-approximation)
- Complete static analysis identifies all violations but may report false positives
- Sound static analysis under-approximates behaviours (no false positives but may miss violations)

## Disassembly Approaches

- **Linear Sweep**:
    - Used by tools like objdump, WinDbg
    - Processes code sections sequentially
    - Complete coverage but easily confused by data in code
- **Recursive Traversal**:
    - Used by tools like IDA, OllyDbg
    - Follows control paths
    - Better at distinguishing code from data
    - May miss code due to unresolved indirect control flow

## Limitations

- Difficulty separating code from data
- Variable-length instructions (x86)
- Indirect control transfers
- Loss of information (variable names, types, etc.)

# Dynamic Analysis

### Characteristics

- Executes program to monitor behaviour
- Under-approximates behaviours but is sound (no false positives)
- Observes actual execution paths

### Techniques

- **Dynamic Disassembly**: Records instructions during execution
- **Debugging**: Monitors execution with breakpoints
- **Control Flow Analysis**: Creates graphs of execution points
- **System Call Monitoring**: Tracks OS interactions

### Goals & Implementations

- **Visibility**: See as much execution as possible
- **Resistance to Detection**: Hide monitoring from malware
- **Scalability**: Handle large volumes of samples

### Analysis Environments

- **Virtualisation**: Hardware-level VM
- **Emulation**: Software simulation of hardware
- **Simulation**: Imitation of abstract model
- **Sandboxes**: Isolated execution environments

### Code Coverage Strategies

- **Test Suites**: Running with known inputs
- **Fuzzers**: Generate inputs automatically
- **Symbolic Execution**: Represent variables symbolically

## Shift to Advanced Detection

### Behaviour-Based Detection

- Monitors events that characterise program execution
- Infer behaviours from system events
- Focus on high-level malicious behaviours
- Can detect novel malware with similar behaviours

### Machine-Learning Detection

- Automated analysis of patterns

- Adaption to new threats
- Feature extraction from binaries
- Classification of unknown samples

# Analysis Tools

## Categories

- **Disassemblers**: IDA Pro, Hopper, radare
- **Debuggers**: gdb, OllyDbg, windbg
- **Analysis Frameworks**: angr, Pin, Dyninst
- **System Monitors**: strace, 1trace, Wireshark

# Analysis Challenges

- Binary analysis is complex and fundamentally undecidable
- Lack of symbolic information
- No type information
- Loss of high-level abstractions
- Mixed code and data
- Location dependent code

# Lecture 5

# Malware Anti-Analysis

Lecture 5: Malware Anti-Analysis

- Discussed evasion techniques malware uses to resist analysis:
- Static evasion: packing, polymorphism, opaque predicates, control flow flattening.
- Dynamic evasion: anti-debugging, sandbox detection, logic/time-bombs.
- Explained obfuscation techniques such as XOR encryption, junk byte insertion, and overlapping instructions.
- Introduced multi-layer packing and metamorphic malware to highlight analysis challenges.

## Overview of Analysis Limitations

- Static and dynamic analysis both have limitations that malware exploits
- Anti-Analysis techniques aim to prevent proper malware classification or detection
- Arms race between malware authors and security researchers

## Static Analysis Evasion

### Obfuscation Techniques

- **Base64 Encoding**: Converts binary data to ASCII format
  - Used to hide data in plain text protocols (e.g., HTTP)
  - Example: "One" encodes to "T251"
- **XOR Encryption:**
  - Single-byte XOR: Each byte XORed with a key value
  - Multi-byte XOR: More secure against brute force attempts
  - Used to hide strings, code, and signatures

### Anti-Static Analysis Methods

- **Junk Insertion**:
  - Adds unreachable code to confuse disassemblers
  - Junk bytes placed at locations not executed at runtime
  - Particularly effective against linear sweep disassemblers
- **Branch Functions**:
  - Modify normal function call behaviour

    - Redirect control flow to confuse analysis tools
    - Make code unreachable for recursive traversal algorithms
- **Overlapping Instructions**:
    - Creates multiple valid instruction paths in the same code
    - Exploits variable-length x86 instructions
    - Breaks disassembler assumption of non-overlapping code chunks
- **Opaque Predicates**:
    - Conditions with outcome known upfront but hard to deduce statically
    - Creates more complex control flow graphs
    - Example: if (((X^2 + X)mod 2) == 0)
- **Control Flow Flattening**:
    - Obfuscates normal program flow
    - Uses switch statements in infinite loops with multiple code blocks
    - Makes code harder to follow and understand

## Packing Techniques

- **Basic Packing**:
    - Compresses executable content
    - Adds unpacking stub that extracts original binary at runtime
    - Modifies entry point to point to stub
- **Multi-layer Packing**:
    - Hides malicious code under multiple layers of compression/encryption
    - Each layer needs to be unpacked during analysis
- **Algorithmic-Agnostic Unpacking**:
    - Uses dynamic analysis to defeating packing
    - Emulates sample execution until unpacking completes
- **Self-Emulating Malware**:
    - Transforms code into bytecode
    - Uses virtual machine to interpret bytecode at runtime
    - Mutates bytecode in each sample

## Polymorphic Techniques

- **Encrypted Viruses**:
    - Enciphers payload, uses decryptor at runtime
    - Evades signature-based detection
- **Oligomorphic Viruses**:
    - Uses multiple decryptors instead of a single one
    - Changes decryptors between generations
- **Polymorphic Viruses**:

- Changes layout with each infection
    - Uses a different encryption key each time
- **Metamorphic Viruses**:
    - Creates semantically-equivalent but structurally different code versions
    - "Body-polymorphics" - entire code changes while maintaining function
    - Analyses and mutates its own code in blocks

# Dynamic Analysis Evation

## Anti-Debugging Techniques

- **Process Detecting**:
    - Checks if being traced using APIs: IsDebuggerPresent
    - Looks at PEB!NtGlobalFlags
    - Uses ntdll!NtQueryInformationProcess
- **The ptrace Trick**:
    - Attempts to attach to itself (only one process can trace)
    - If fails (returns -1), knows it's being debugged
    - Can be defeated by redefining ptrace() function to always return 0

## Sandbox Evasion Methods

- **Red Pills**: Programs that detect if running in emulated environment
    - Example: SIDT instructions to detect VM
- **System Fingerprinting Categories**:
    - Environmental Artifacts
    - Timing Checks
    - CPU virtualisation detection
    - Process Introspection
    - Reverse Turing tests
    - Network artifacts
    - Mobile sensors
    - Browser-specific checks
- **Sleep Evasion**:
    - Waits before executing malicious code
    - Anti-sleep: Analysis tools may skip sleep calls
- **Human Interaction Detection**:
    - Monitors for mouse/keyboard activity
    - Only activates after detecting human-like behaviour
- **VM/Sandbox Detection**:
    - Checks for VM-specific processes, files, registry keys

    - Looks for analysis tool artifacts
    - Examines hardware characteristics

## Malware Anti-Analysis Tools

- RDG Tejon Crypter: Obfuscation tool
- Pafish: Demonstrates sandbox detection techniques
- al-khaser: Proof of Concept (PoC) tool showing common sandbox evasion methods

# Lecture 6

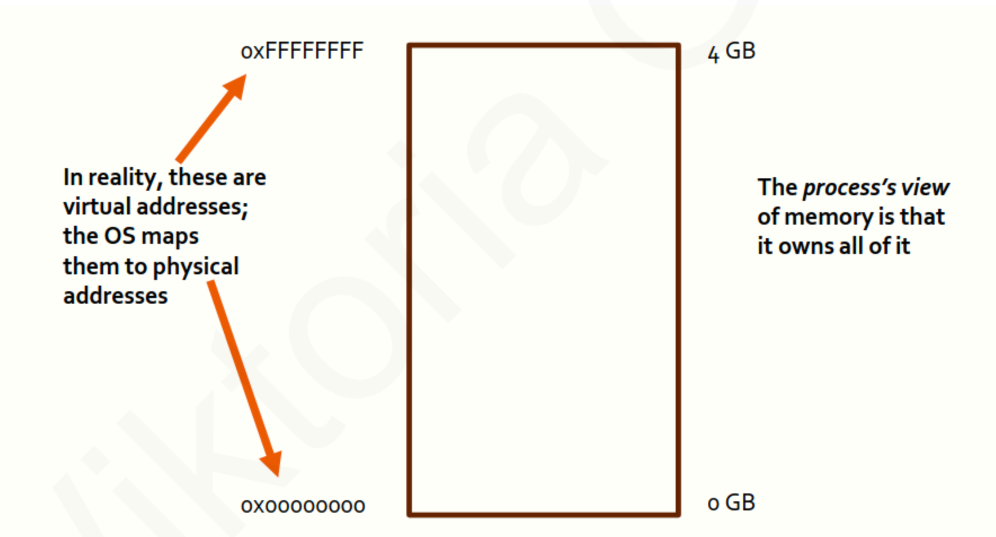
# Buffer Overflow, SQL Injection, and Cross-Site Scripting

Lecture 6: Buffer Overflow, SQL Injection, and XSS

- Focused on exploitation techniques used in malware:
- Buffer overflows: manipulate memory to alter program flow (e.g., return address overwrite).
- Code injection: load and execute attacker-supplied code.
- SQL Injection and Cross-Site Scripting (XSS): inject malicious input into web apps.
- Illustrated stack frames, memory layouts, and defenses like stack canaries and ASLR (Address Space Layout Randomization).

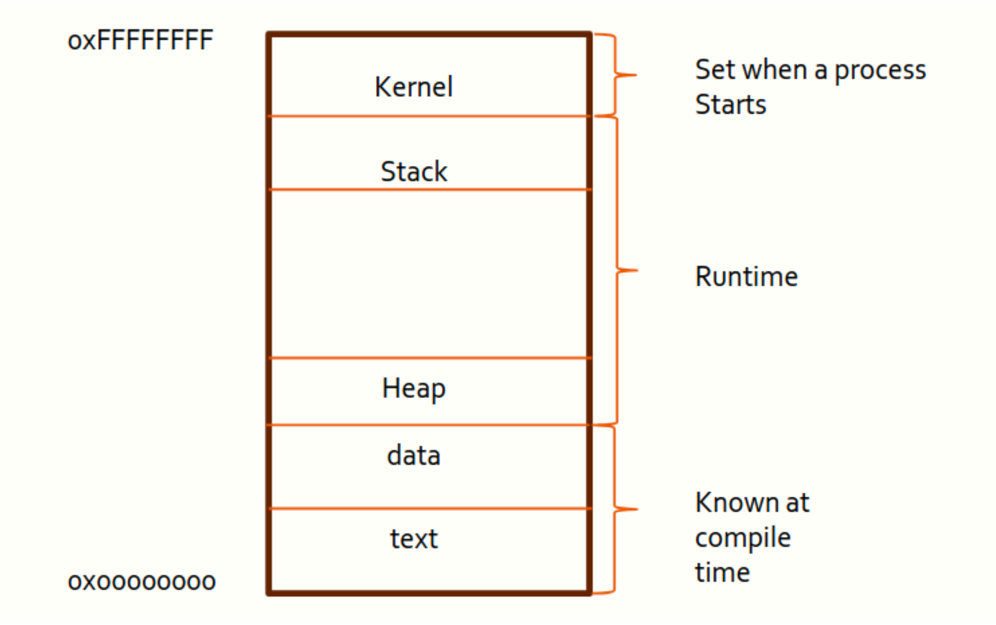
## Memory Layout

### C Call Stack

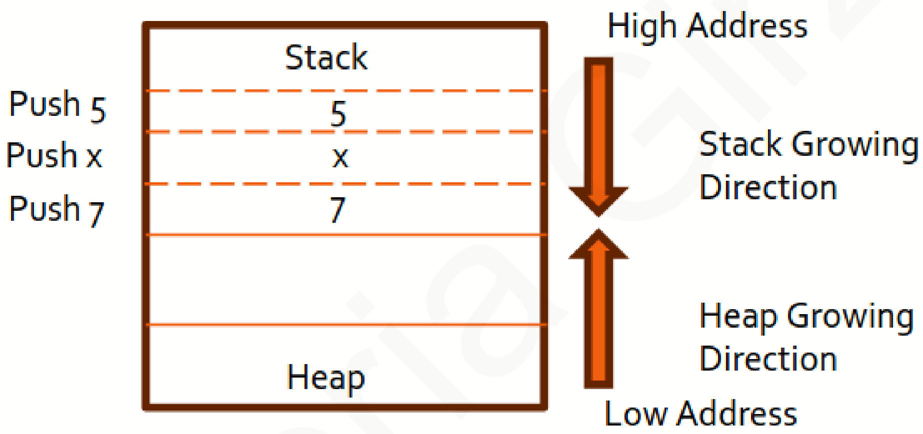



- When a function call is made, return address is put on the stack
- Values of parameters are put on the stack
- Local variables are put on the stack
- Function saves stack frame pointer (on the stack)
- On Linux (x86), stack grows from high addresses to low
- Pushing something on the stack moves Top Of Stack towards address 0

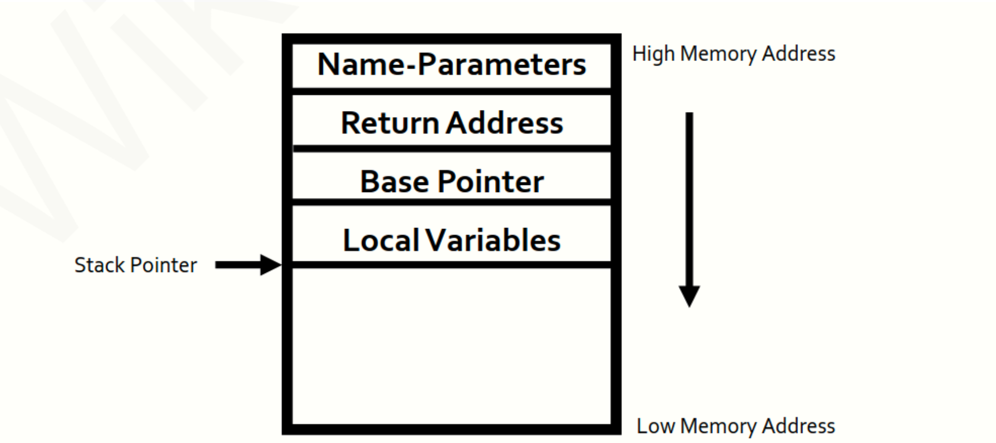
### Stack vs Heap Memory Organisation




- **Stack**: Used for function call management, local variables, return addresses
- **Heap**: Grows in opposite direction, used for dynamic memory allocation
- Both are regions in the process memory space




## Stack Frame Structure




Each function has its own stack frame containing:

- **Function parameters**: values passed to function
- **Local variables**
- **Return address**: address where execution should continue after function completes
- **Saved base pointer**: previous frame's base pointer (saved %ebp)
- **Frame pointer** (%ebp): points to base of current stack frame
- **Stack pointer** (%esp): points to top of stack (growing downwards in x86 structures)

## Function Call Process

1. **Calling function:**
    - Push arguments onto stack (in reverse order)
    - Push return address of instruction to follow after control returns to you
    - Jump to function
2. **Called function:**
    - Push old frame pointer onto stack (%ebp)
    - Set new frame pointer (%ebp) to where the end of the stack is right now (%esp)
    - Push local variables onto the stack
3. **Function return:**
    - Deallocate local variables: %esp = %ebp
    - Restore base pointer: pop %ebp
    - Jump to return address: %eip = 4(%ebp)
4. **Back in calling function:**
    - Remove arguments from stack

# Buffer Overflow

### Buffer

- Contiguous set of a given data type
- Common in C
- All strings are buffers of char's

### Overflow

- Put more into the buffer than it can hold

## Examples of Vulnerable Code

```c
// Example 1: Buffer on stack overflow
char buff[4];
strcpy(buff, "Hello:)");  // Overflow
```

- Buffer is only 4 bytes, but "Hello:)" is 7 bytes (plus null terminator)

- `ebp` gets replaced with ASCII values from overflow
- When restoring the pointer, it will read corrupted value

```c
// Example 2: Dangerous function
char fileData[50];
gets(fileData);  // No bounds checking
```

- Use safer functions like fgets() instead

Buffer overflow inputs can come from:

- Text input fields
- Network packets
- Environment variables
- File input

Unsafe functions like `strcpy()` and `gets()` will copy data until a null terminator without checking buffer size.

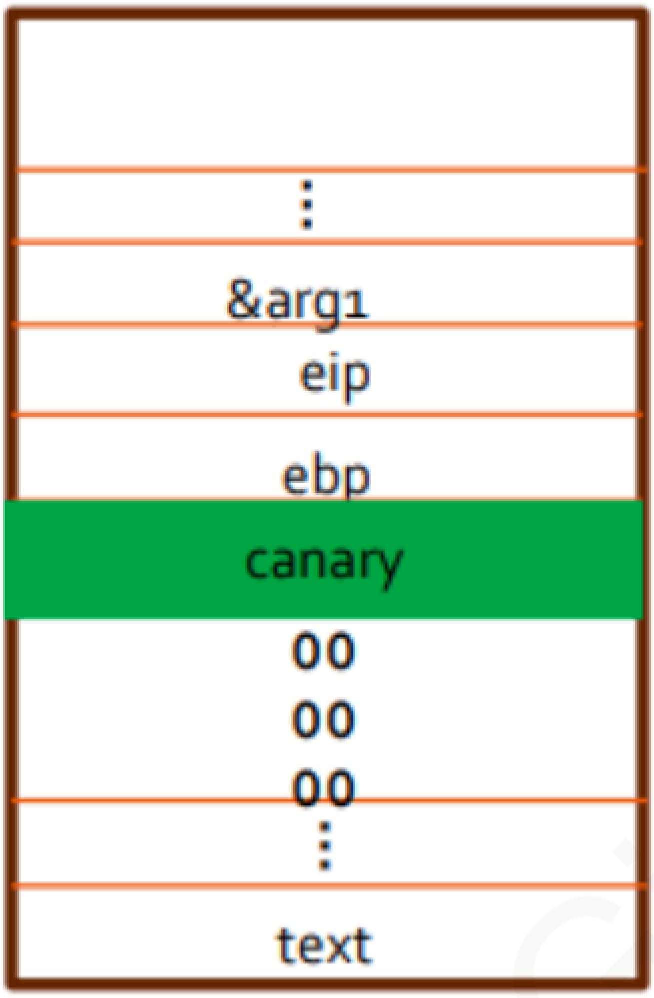
## Code Injection

Buffer overflows can be exploited for code injection by:

1. **Loading code into memory**: Injecting shell code that must:
   - Avoid null bytes (would terminate string functions)
   - Be self-contained (not rely on loader)
   - Not depend on stack integrity
   - **Goal often**: get a shell/privilege escalation
2. **Redirecting execution flow**: Getting code to run:
   - Overwrite return address to point to injected code
   - Can't insert explicit "jump" instructions
3. **Finding the return address**: Determining the exact location to overwrite
   - Without code access, hard to know buffer-to-EBP distance
   - Approach: try many values or exploit predictable addresses
   - With ASLR, this becomes much more difficult

## Defences Against Buffer Overflows

1. **Stack Canaries**: Values placed between buffers and control data




    - Types:
        - **Terminator Canaries** (CR, LF, NULL, -1) - leverages the fact that scanf, etc. don't allow these
        - **Random canaries -** write new random value @ each process start, protecting stored value in memory
        - **XOR canaries -** same as random canaries, but store "canary XOR control info"
    - Checked before function returns to detect corruption
2. **Address Space Layout Randomisation (ASLR)**:
    - Randomises memory locations to make predicting addresses difficult
    - Adoption timeline: Linux (2005), Vista (2007), mac OS (2007/2011), iOS (2011), Android (2011)
3. **Non-executable stack**: Prevents execution of injected code
4. **Proper coding practices**: Using safe functions, bounds checking, input validation

## SQL Injection

- Attackers manipulate SQL queries through unchecked input
- Can lead to unauthorised data access or manipulation
- Examples: entering `' OR '1'='1` instead of valid username

# Cross-Site Scripting (XSS)


- Malicious scripts injected into trusted websties
- Scripts execute in users' browsers
- Can access cookies, session tokens, and sensitive information
- Browser cannot distinguish between legitimate and malicious scripts

# Cross-Site Request Forgery (CSRF)

Example with POST:

```
<a href="http://bank.com/transfer.do?acct=MARIA&amount=100000">View my Pictures!</a>

<img src="http://bank.com/transfer.do?acct=MARIA&amount=100000" width="0" height="0" border="0">
```

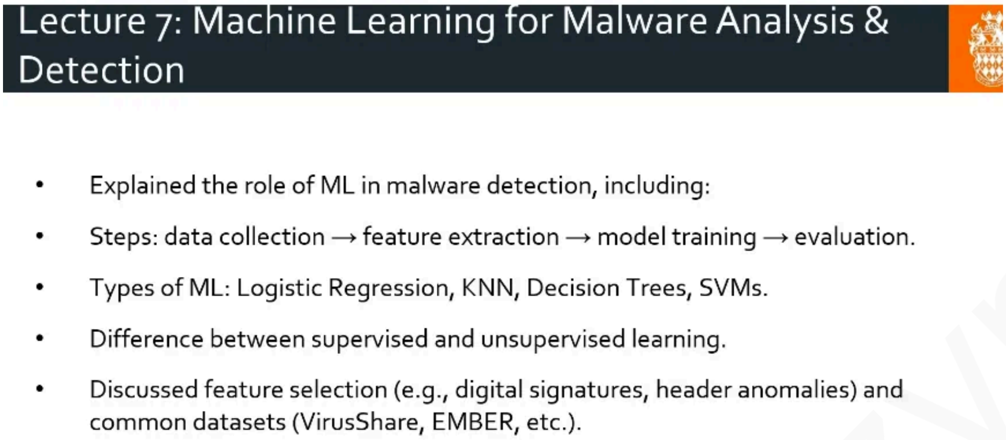
Example with POST:

```
<form action= "<nowiki>http://bank.com/transfer.do</nowiki>" method="POST">
<input type= "hidden" name= "acct" value="MARIA"/>
<input type= "hidden" name= " amount" value="100000"/>
<input type= "submit" value="View my pictures"/>
</form >
```

- Tricks users into performing unwanted actions on sites where they're authenticated
- Exploits the trust a site has in a user's browser
- Unlike XSS which exploits user's trust in a site

# Lecture 7

# Machine Learning for Malware Analysis and Detection




# Basics of Machine Learning

- ML is a **set of mathematical techniques** enabling computers to learn from data
- Helps computers generalise past data to **predict future outcomes**
- Definitions:
  - "Machine Learning is the science of programming computers to learn from data"
  - "Field of study giving computers ability to learn without explicit programming" (Arthur Samuel, 1959)
  - "A program learns from experience E with respect to task T and performance measure P if performance improves with experience" (Tom Mitchell, 1997)

# ML in Cyber Security

### Use Cases

- **Pattern Recognition**: Discover characteristics in data to recognise similar patterns
  - Examples: spam detection, malware detection, botnet detection
- **Anomaly Detection**: Establish baseline normality and identify deviations
  - Examples: network outlier detection, user authentication

# Supervised vs Unsupervised Learning

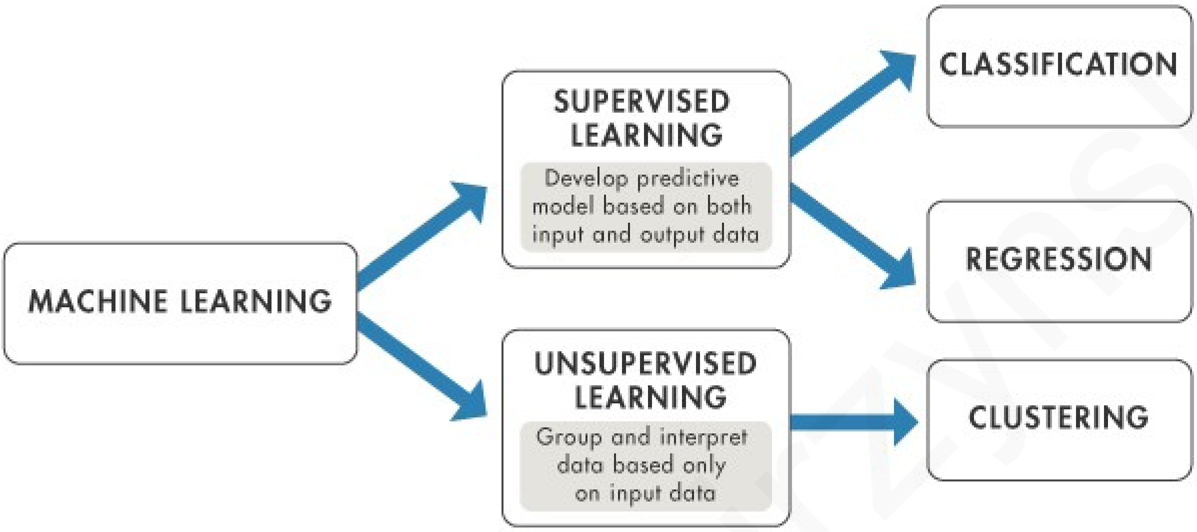
### Supervised Learning

- Known number of classes

- Learning from labelled training data
- Used to classify future observations

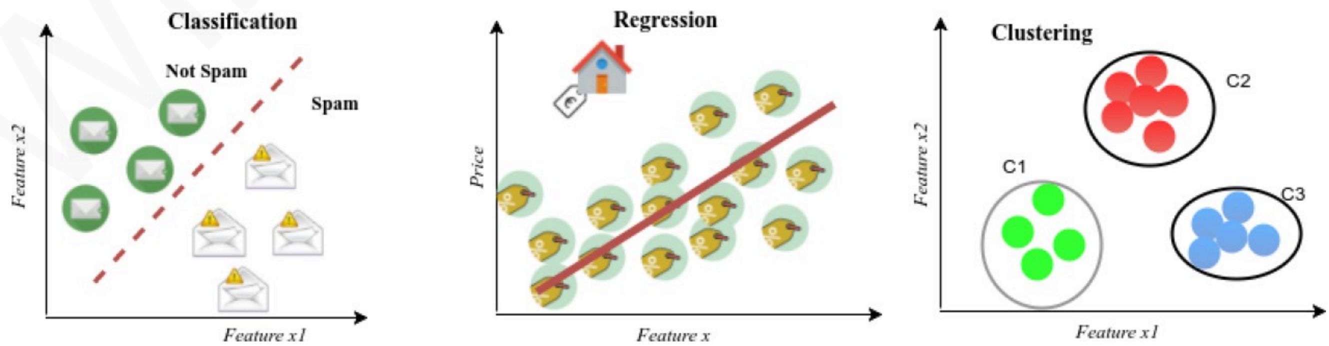
**Unsupervised Learning**

- Unknown number of classes
- No prior knowledge
- Finds "natural" groupings of instances




**ML Tasks**

- **Classification**:
  - Given labelled dataset,
  - Separate instances into predefined classes
- **Regression**:
  - Given some points,
  - Predict numerical values
- **Clustering**:
  - Given an unlabelled dataset,
  - Group similar elements in unlabelled data




# Building ML-Based Malware Detectors

1. **Gathering Training Examples**
    - Quality and quantity of training examples are crucial
    - Need both malware and benignware samples
    - Examples should mirror what the detector will encounter
    - Collection considerations:
        - Freshness
        - Quality/Verifications
        - Quantity
        - Target OS
        - Format (binaries or features)
        - Source (public/private)

**Common Malware Datasets**

- VirusShare
- VirusTotal
- Androzoo
- theZoo (Live Malware Repository)
- Microsoft Malware Classification Challenge
- EMBER dataset

2. **Feature Extraction**
    - Extract distinctive attributes from binaries
    - Good feature examples:
        - Digital signatures
        - Header information
        - Presence of encrypted data
        - Imported tables
        - String features
        - N-grams
    - Feature selection considerations:
        - Choose features that distinguish malware from benignware
        - Avoid too many features (curse of dimensionality)
        - Feature scaling is important
        - Feature representation matters

**Feature Selection Methods**

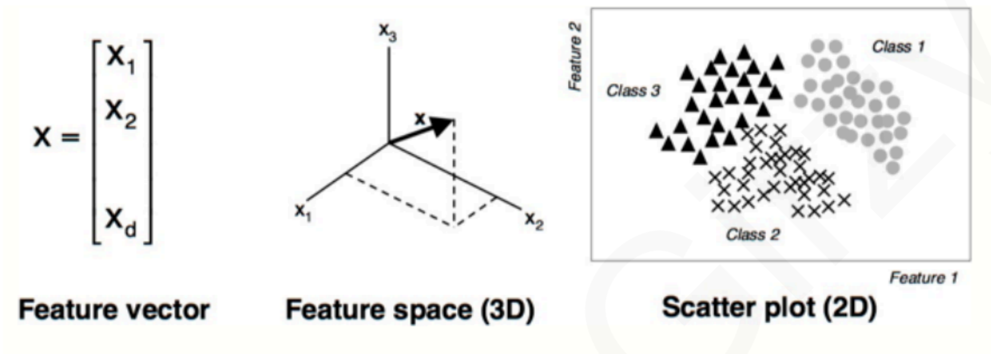
- **Manual Selection**: Based on domain expertise
- **Univariate Analysis**: Evaluate features individually
- **Recursive Feature Elimination**: Start with all features and eliminate iteratively

- **Latent Feature Representations**: PCA, SVD to reduce dimensionality
- **Model-Specific Ranking**: Use weights from trained models

3. **Training ML Systems**
   - Provide algorithm with labelled
   - Allow it to distinguish between malware and benignware
4. **Testing ML Systems**
   - Measure accuracy using data not included in training
   - Evaluate how well it detects new malware and avoids false positives
   - Use appropriate performance metrics

# ML Algorithms for Malware Detection

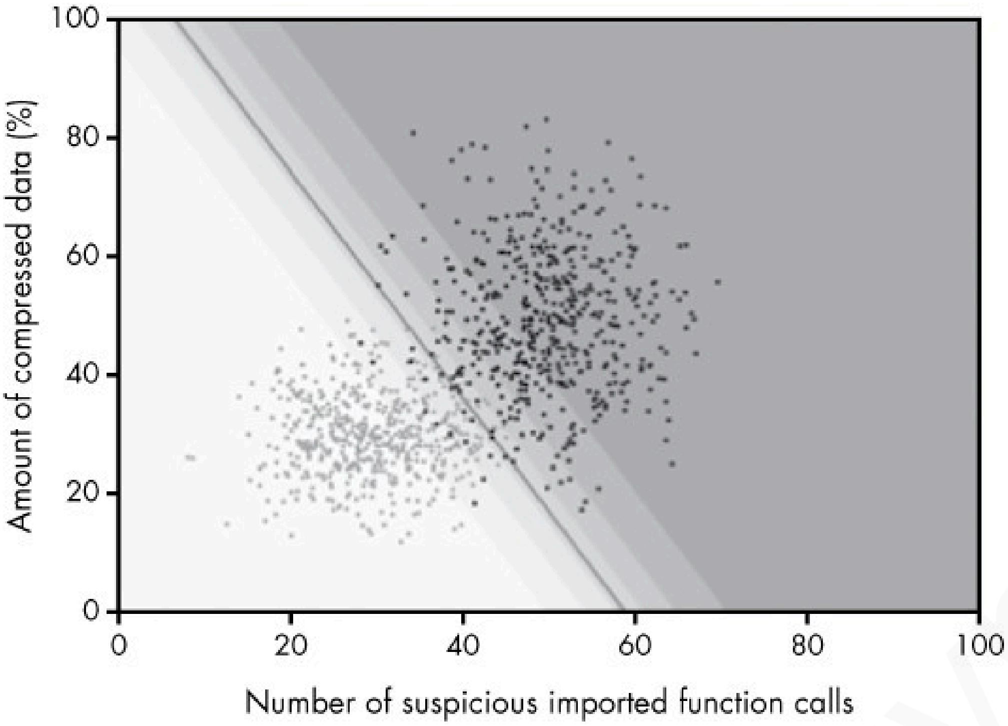
## Feature Spaces and Decision Boundaries



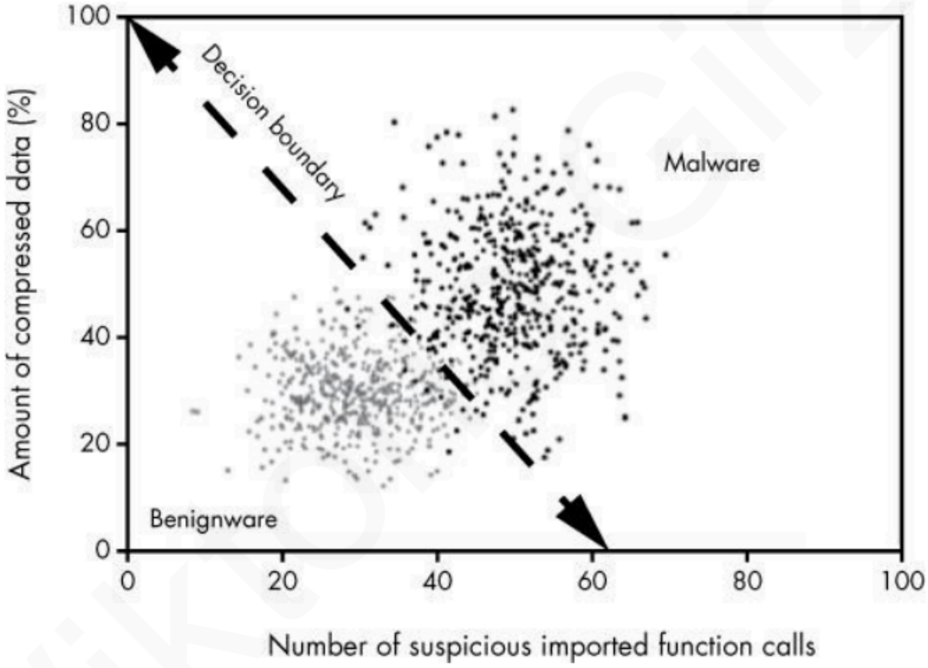

- Features create a geometrical space
- Decision boundaries separate benignware from malware
- Different algorithms create different types of boundaries

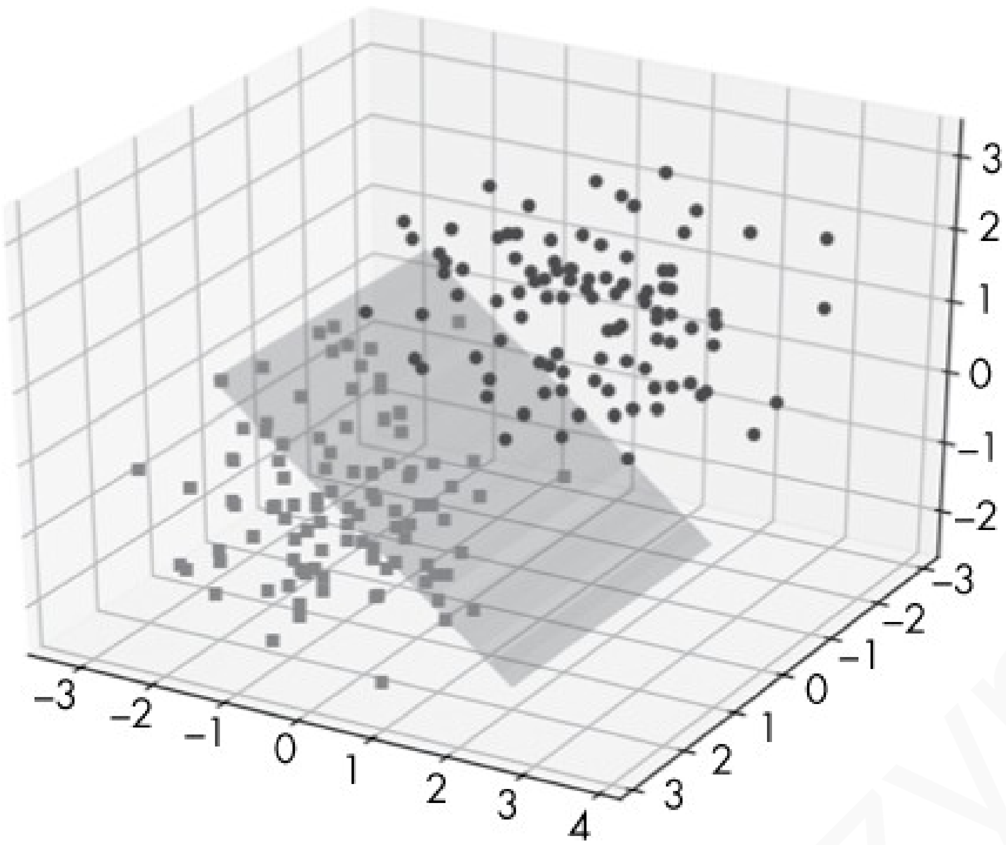
## Logistic Regression

Logistic Regression




Defining a Malware Detection Decision Boundary

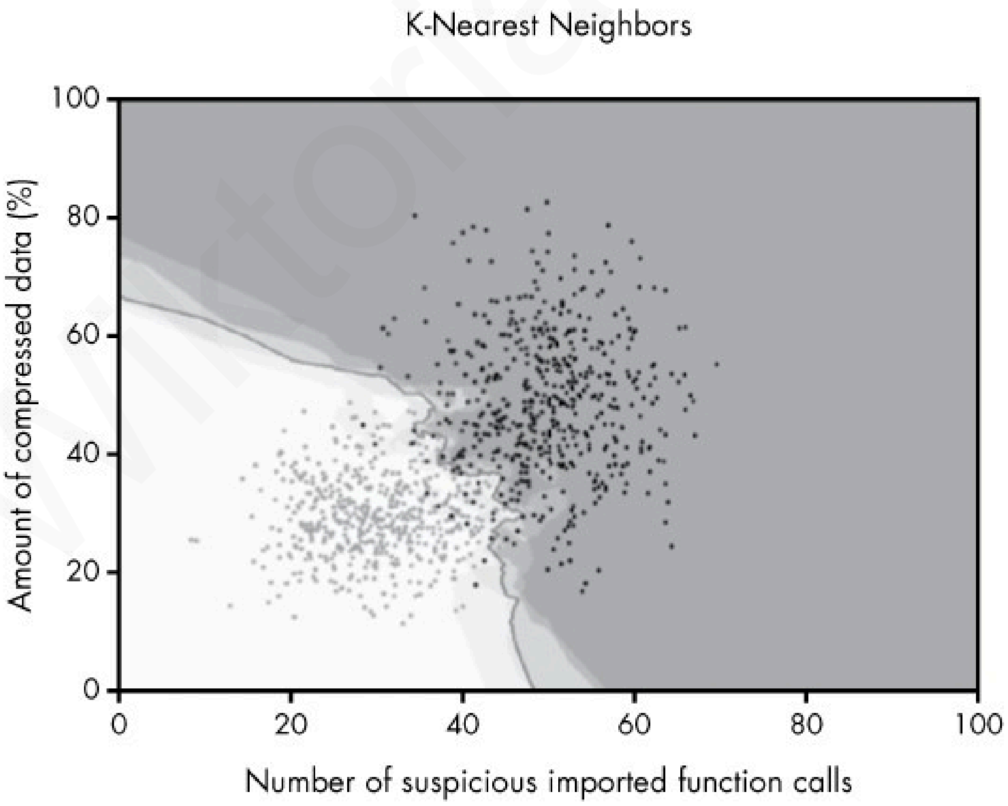


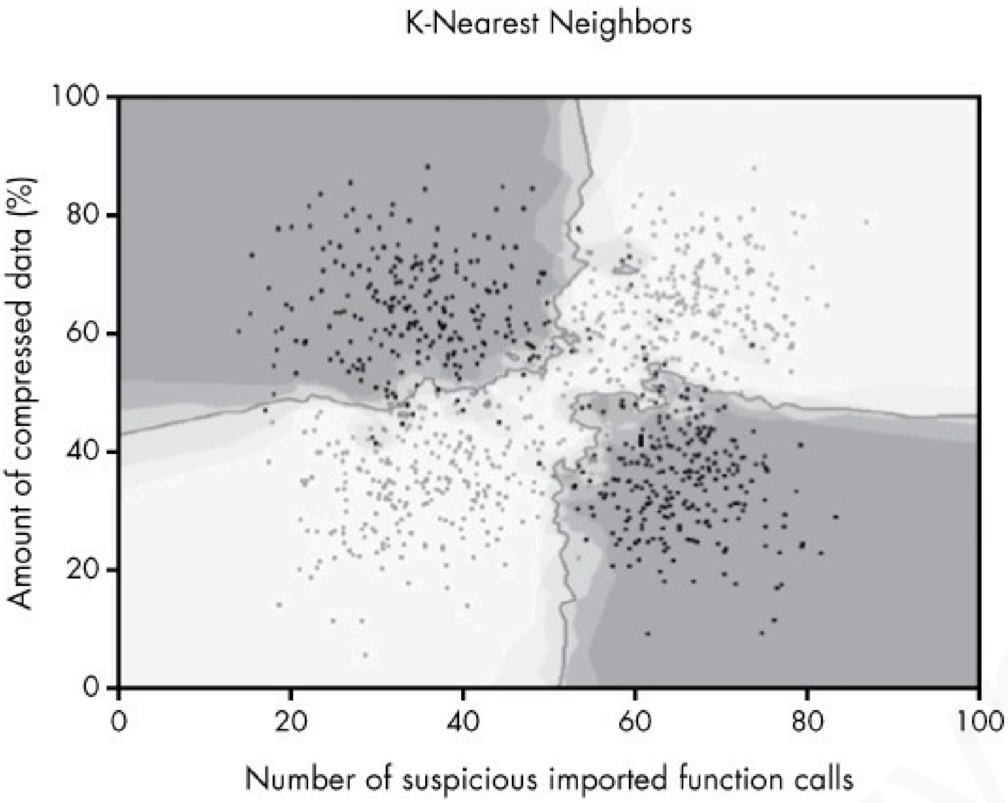



Example in 3d space

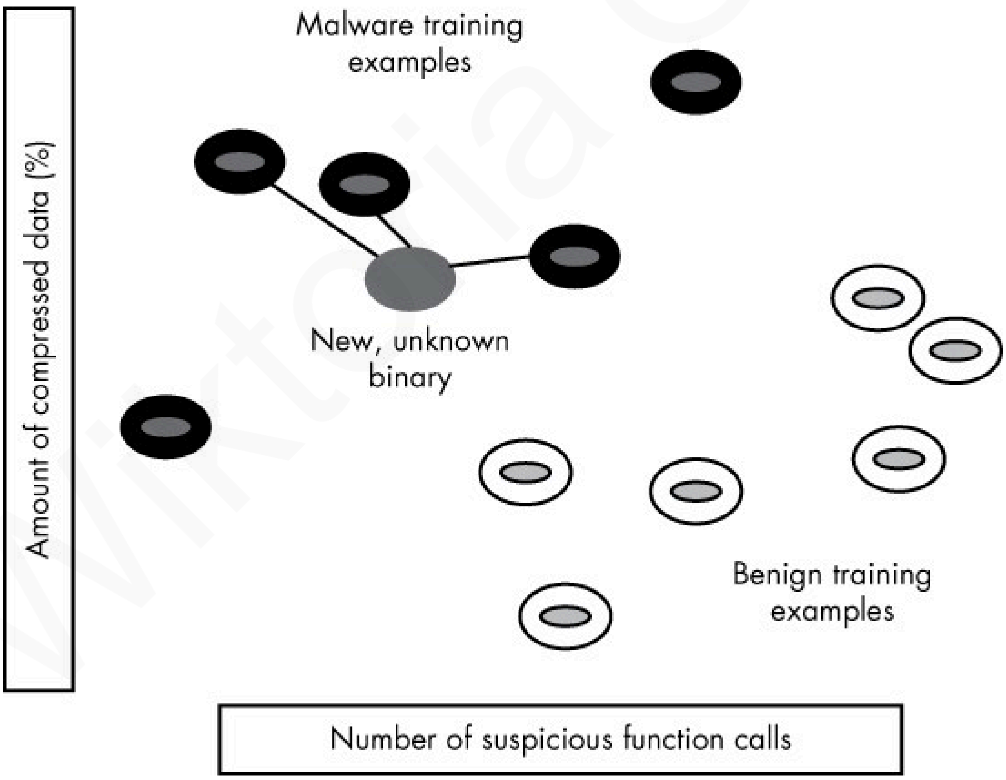
- Creates linear decision boundary (line, plane, or hyperplane)
- Good when individual features are strong indicators
- Limited with complex relationships between features

## K-Nearest Neighbours

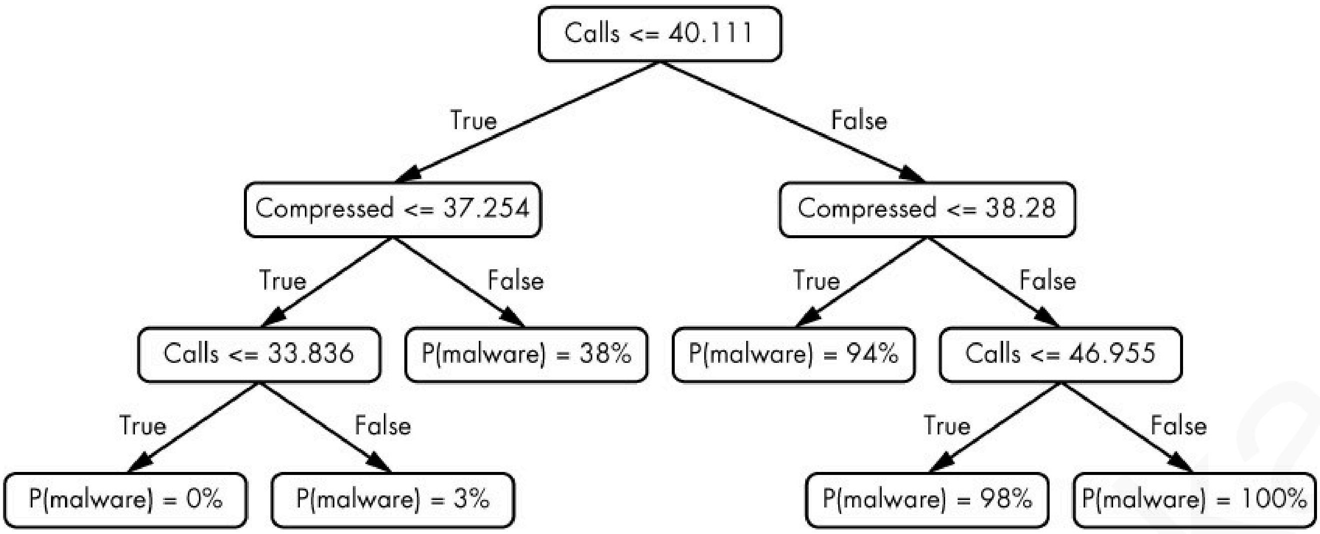


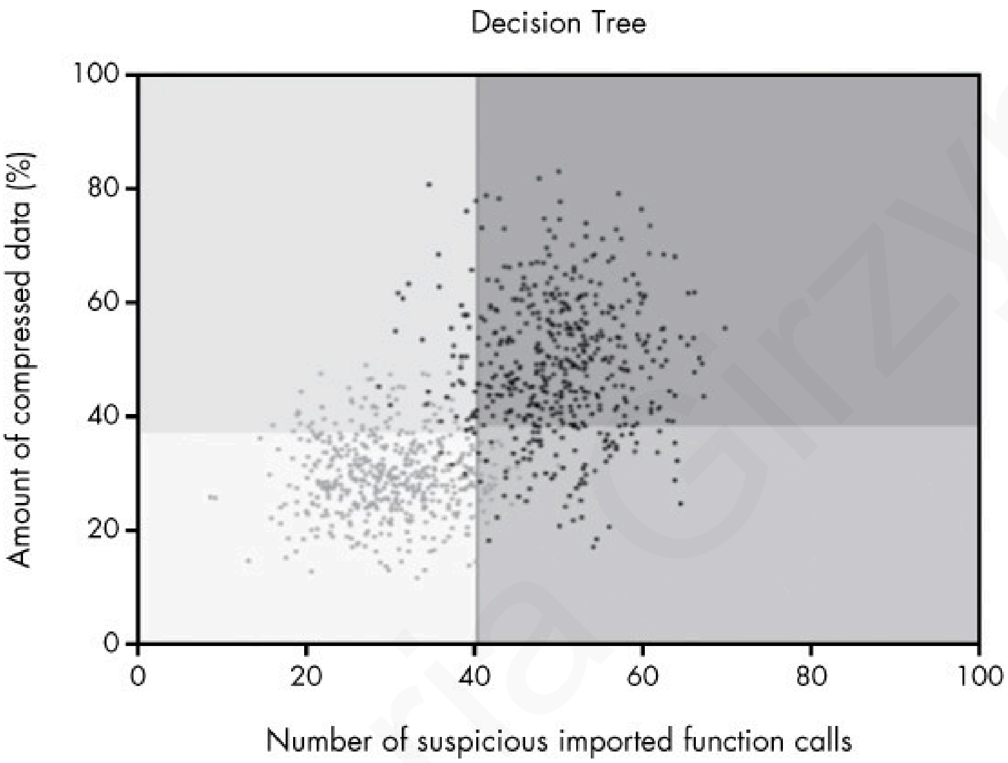



- Based on proximity to known samples
- If majority of k closest binaries are malicious, classify as malicious
- Works well when "closeness" to known samples is meaningful
- Good for malware family classification

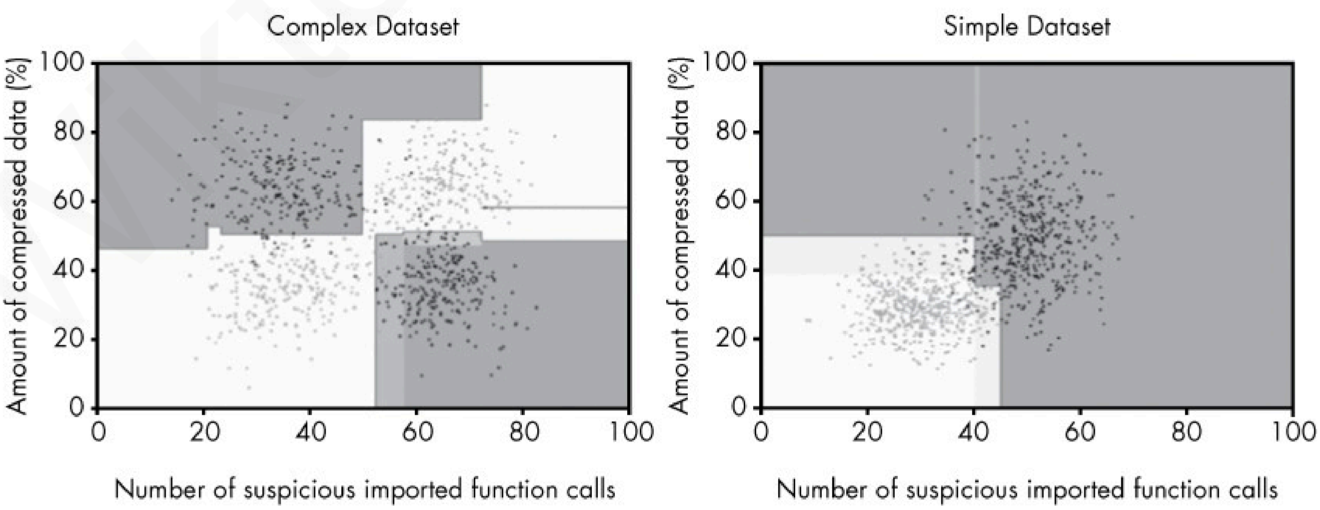



## Decision Trees



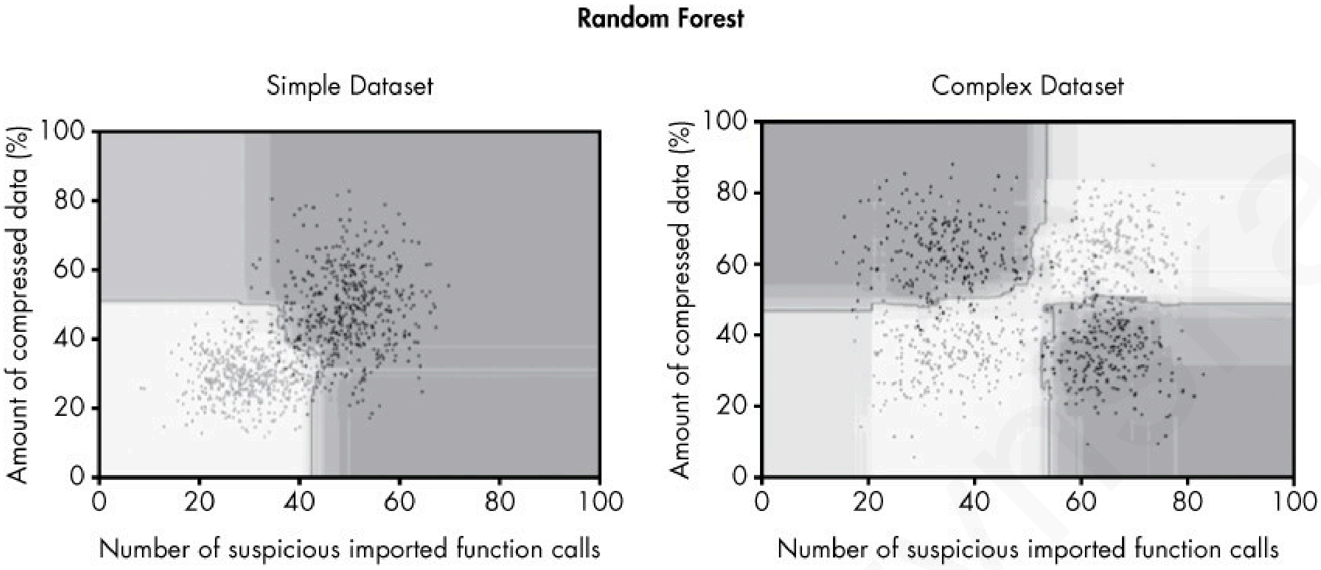

**Decision Tree (Limited Depth)**




- Generate series of questions through training
- Can learn irregular boundaries

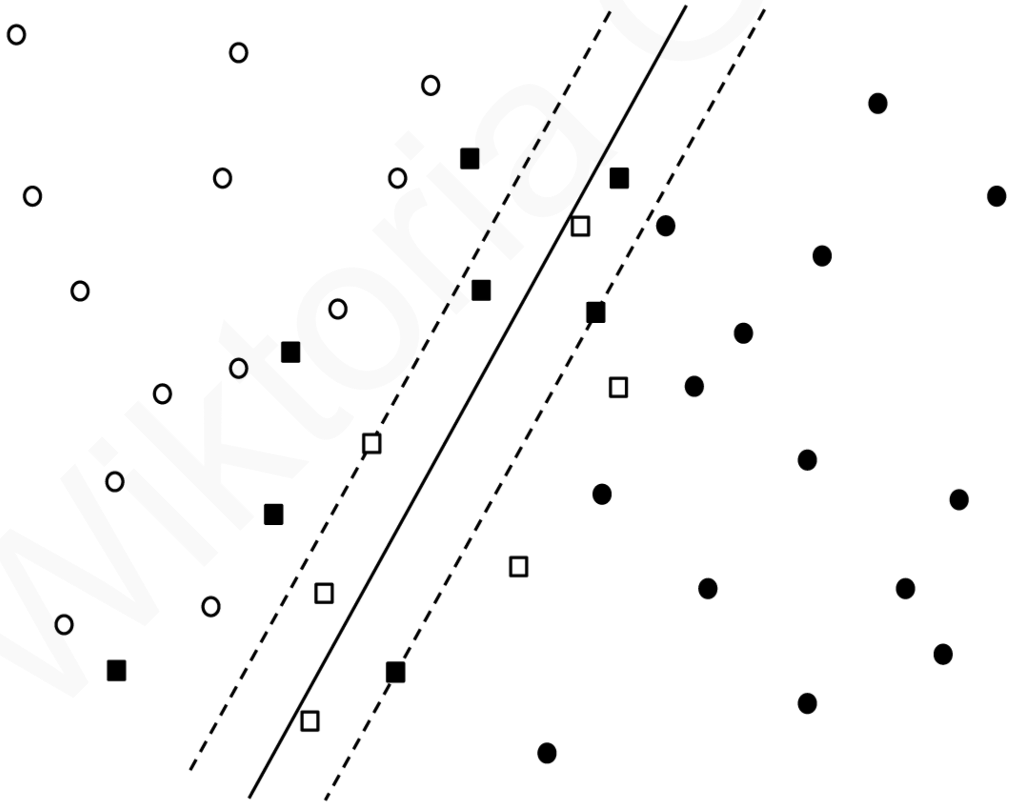
- May not generalise well to new examples
- Decision boundaries can be jagged

## Random Forest

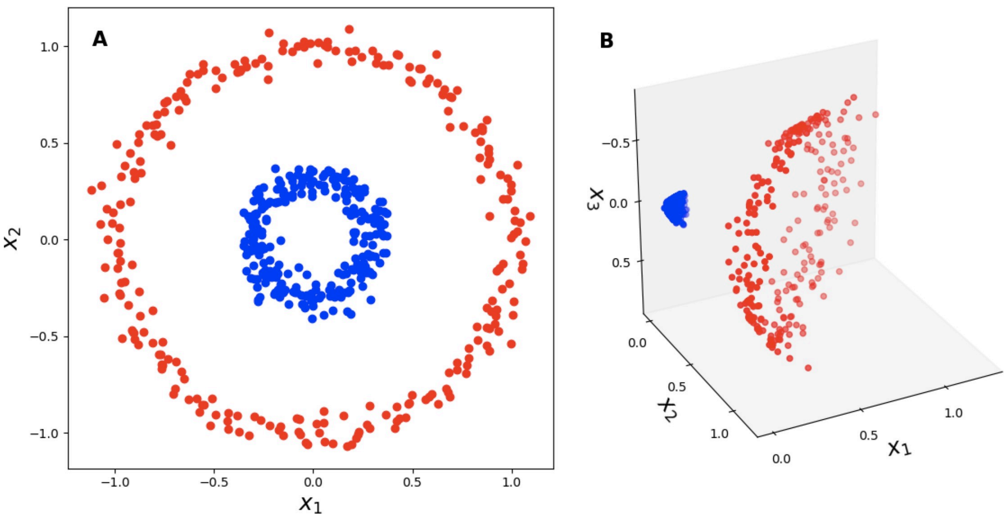



- Ensemble of decision trees
- Each tree trained differently for diverse perspectives

## Support Vector Machines (SVMs)



Classification boundary (dark line) and margins (dashed lines) for linear SVM separating two classes (black and white points); squares represent support vectors




- Finds maximum-margin hyperplane separating classes
- Kernel trick allows non-linear boundaries
- Performs well in high-dimensional spaces
- Training complexity increases with dataset size

# Evaluating Malware Detection Systems

## Performance Metrics

- **True Positive Rate** (Sensitivity/Recall) : `TPR = TP/(TP+FN)`
- **False Positive Rate**: `FPR = FP/(FP+TN)`
- **Precision** (Positive Predictive Value): `PPV = TP/(TP+FP)`
- **F1 Score**: `2·(PPV·TPR)/(PPV+TPR)`
- **Accuracy**: `ACC = (TP+TN)/(TP+TN+FP+FN)`
- **ROC Curve**: Plots TPR against FPR at various thresholds
- **AUC**: Area under ROC Curve (higher is better)

### $F_1$-Score

$F_1$-Score is defined as the harmonic mean of Precision and Recall:

$$F_1 = 2 \cdot \frac{\mathrm{PPV} \cdot \mathrm{TPR}}{\mathrm{PPV} + \mathrm{TPR}}$$

### Accuracy

Accuracy is the proportion of true results (both true positives and true negatives) among the total number of cases examined:

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

## Base Rate Considerations

- Base rate: percentage of binaries that are actually malware

- **Precision Base on Base Rate**: `PPV = (TPR·BR)/(TPR·BR + FPR(1-BR))`
- Affects precision but not TPR/FPR
- Base rate fallacy: ignoring prevalence when interpreting test results

## No-Free-Lunch Theorem

- No single ML algorithm works best across all scenarios
- Each algorithm has strengths and weaknesses
- Model selection requires understanding the problem domain