COMPX508 – Malware Analysis

Week 7

Lecture 1: Introduction to dynamic analysis

Vimal Kumar

What does this program do?

Malware Analysis

- When analysing malware
  - We want to understand malware behaviour
    - What, why, where, when, who, how ?

- We don’t have access to the source code
  - At best we have access to an executable/binary
    - We can analyse it statically without executing it
    - We can observe its behaviour when it is executed

Dynamic Analysis

- Dynamic analysis is any examination performed after executing malware
- Unlike static analysis, in dynamic analysis you observe the malware’s true functionality
  - We analyze
    - the observable effects of the program
    - the unobservable effects that the program has on the system.

- Advantages
  - Dynamic analysis is immune to obfuscation attempts
- Limitations
  - Not all code paths may execute when a piece of malware is run resulting in incomplete analysis
  - Risk of Damage

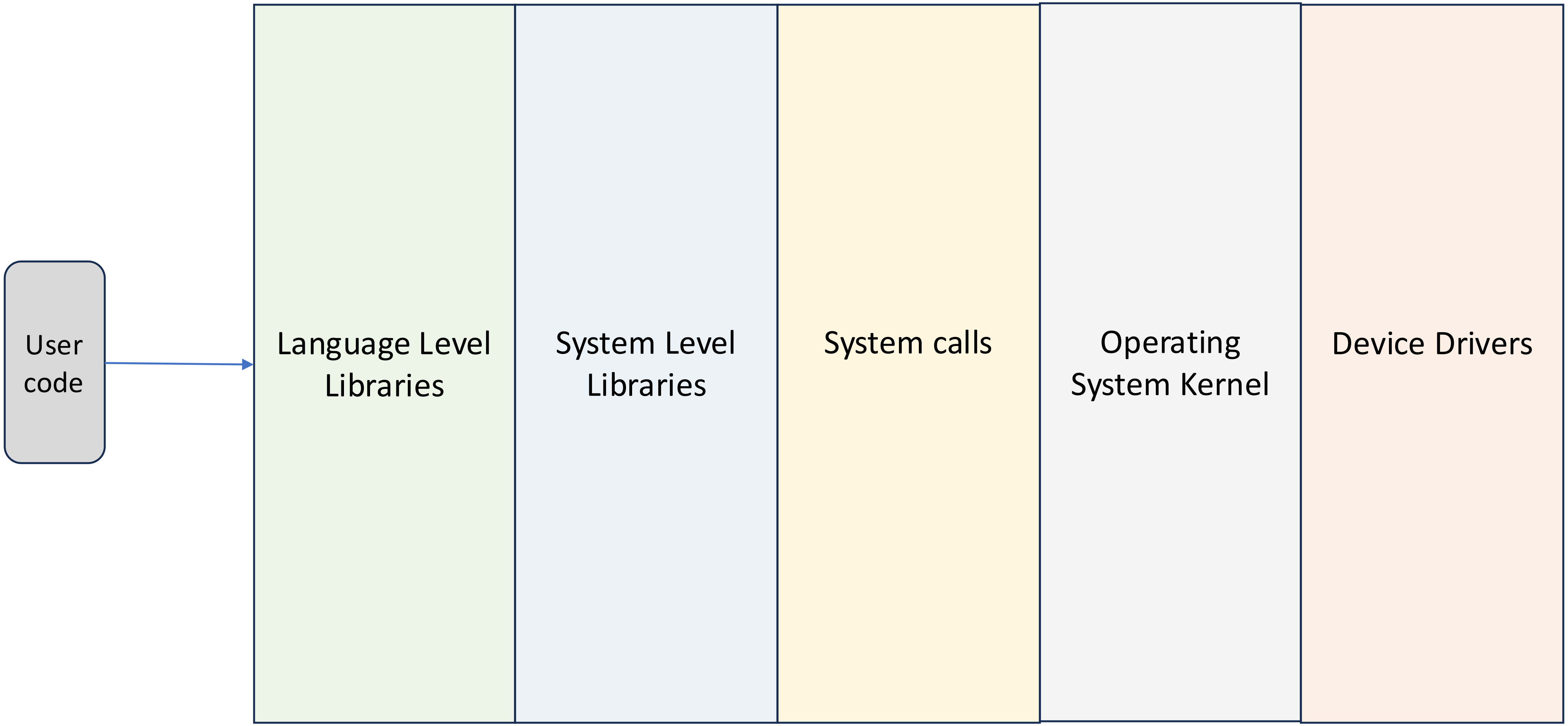
Footprints

- As the program executes, it interacts with the rest of the computer system.
- Monitoring this interaction can also help us understand the behaviour of the program.
  - Monitor interaction with the operating system
  - Monitor network activity
  - Monitor interaction with registry
  - Monitor file system interaction

Dynamic Analysis

- Monitor various parts of the systems that malware tend to affect. Includes the following.
- Process monitoring
  - Involves monitoring the process activity and examining the properties of the result process during malware execution.
- File system monitoring
  - Includes monitoring the real-time file system activity during malware execution.
- Registry monitoring
  - Involves monitoring the registry keys accessed/modified and registry data that is being read/written by the malicious binary.
- Network monitoring
  - Involves monitoring the live traffic to and from the system during malware execution.
- System Call monitoring
  - Monitor system calls generated by the actions of the malware
- API Call monitoring
  - Monitor the API calls being made by the malware

Code at various levels of program execution



User code

Language Level Libraries

System Level Libraries

System calls

Operating System Kernel

Device Drivers

Code at various levels of program execution

- Language-level libraries
  - Functions provided by the language
    - I/O, Memory Management, Data structures etc.
  - Third party libraries
  - User-defined functions
- System-level libraries
  - Provided by the operating system
    - Often dynamically linked
      - .dll, .so, .dylib
    - libc.so, UCRTbase.dll, kernel32.dll, user32.dll, etc.
  - Functions in these libraries are often called by other language-level library functions
    - printf(), malloc(), getc(), putc(), etc.

- System calls
  - System calls are necessary to transition from user-mode to kernel-mode
  - Read, write, etc.
  - System calls are generally used through calls to pre-defined functions in system-level libraries
  - They can however, with some difficulty be called directly by user.
    - Some malware do this to avoid being tracked.
- Kernel
  - Core part of the operating system
  - Handles memory, CPU, devices etc.
  - Runs in privileged mode
- Device Drivers
  - Code specific to each device, that translates higher level commands like write, read, etc. into actual physical actions for that specific device

Hello World

```
#include <stdio.h>

int main(void)
{
   printf("Hello World!\n");
   return 0;
}
```

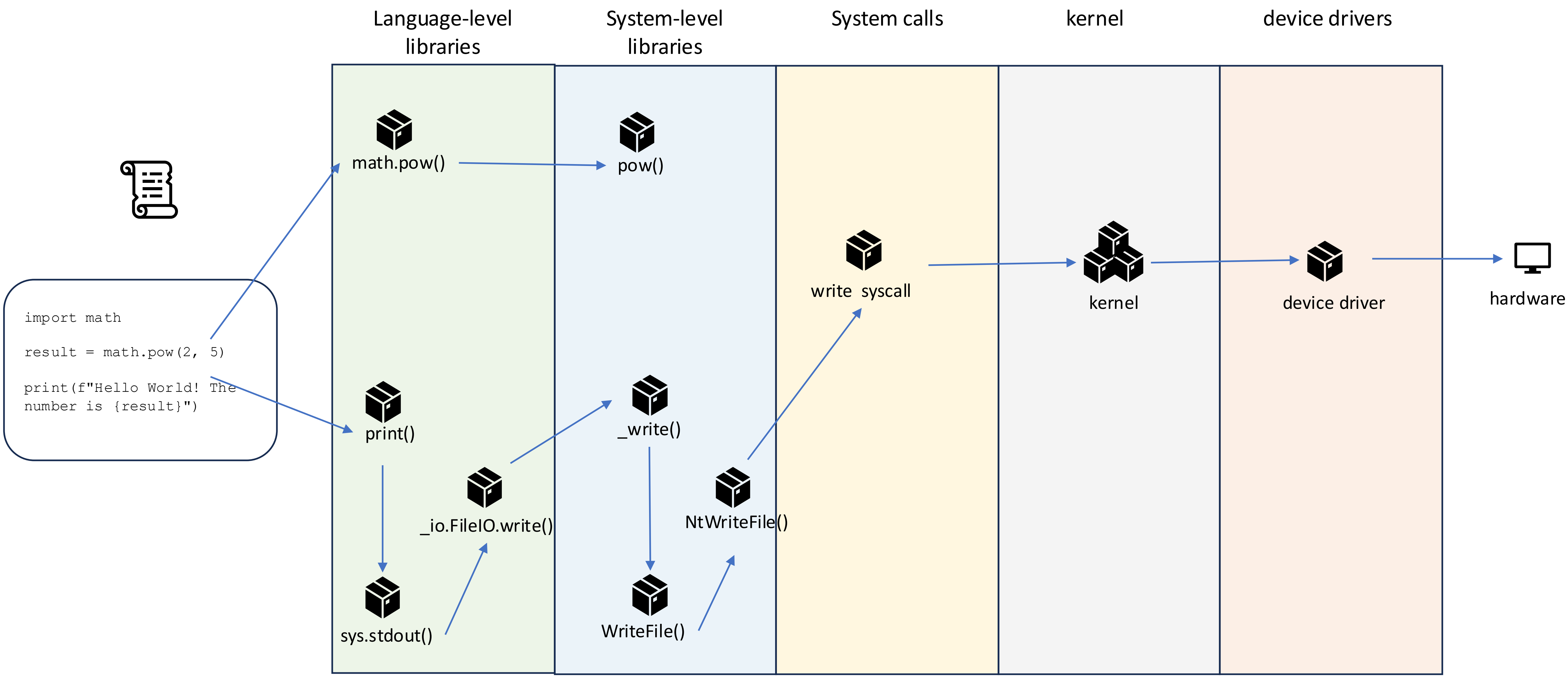
```
#include <iostream>

int main()
{
  std::cout<<"Hello World!";
  return 0;
}
```

```
public class Hello
{
   public static void
main(String[] args)
   {
          System.out.println(
"Hello World!");
    }
}
```

```
print("Hello World!")
```

Example: Python



```
import math

result = math.pow(2, 5)

print(f"Hello World! The
number is {result}")
```

Language-level libraries

System-level libraries

System calls

kernel

device drivers

math.pow()

pow()

print()

sys.stdout()

_io.FileIO.write()

_write()

WriteFile()

NtWriteFile()

write syscall

kernel

device driver

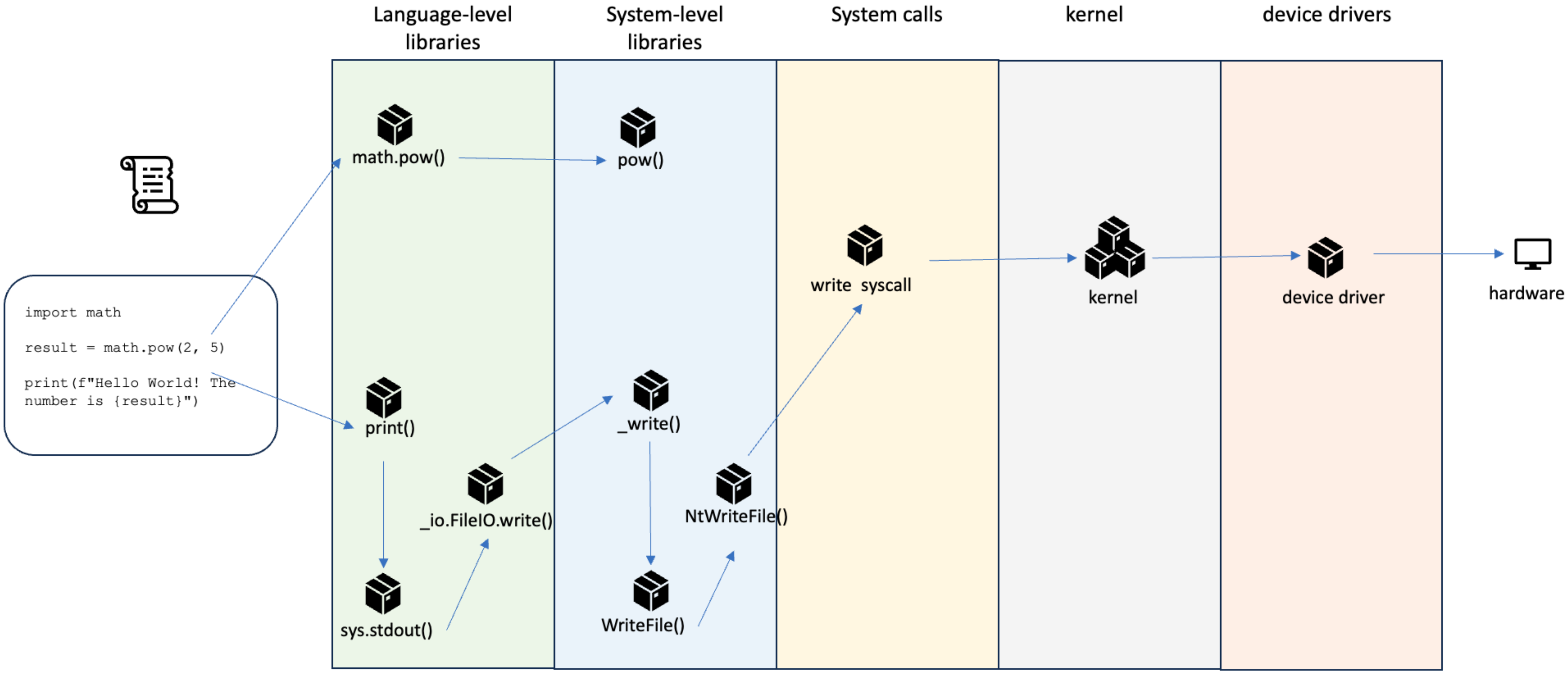
hardware

Behaviour of an executable

- Some understanding of an executable's behaviour can be gained from looking at the function calls being made

- Function names are often informative and provide a reasonable idea of what they do
  - e.g. –`print()` , `TextOutA()`, etc.

- However, malware executables often have that information "**stripped**"
  - This is in fact the the default behaviour of IDEs such as VS Code when any code is built for release.
  - The information often *stripped* is function names and variable names in the code as well as of any language-level library compiled with the code



```
import math

result = math.pow(2, 5)

print(f"Hello World! The
number is {result}")
```

Behaviour of an executable

- The information that is *stripped* is information that is useful for debugging but not necessary for program execution.

- The information that is necessary for program information cannot be *stripped*
  - Literal strings in the code
  - Names of shared libraries/dlls
  - Call to functions in the shared libraries/dlls