SAPIENZA
UNIVERSITÀ DI ROMA

Symbolic Execution of Malicious Software: Countering Sandbox Evasion Techniques

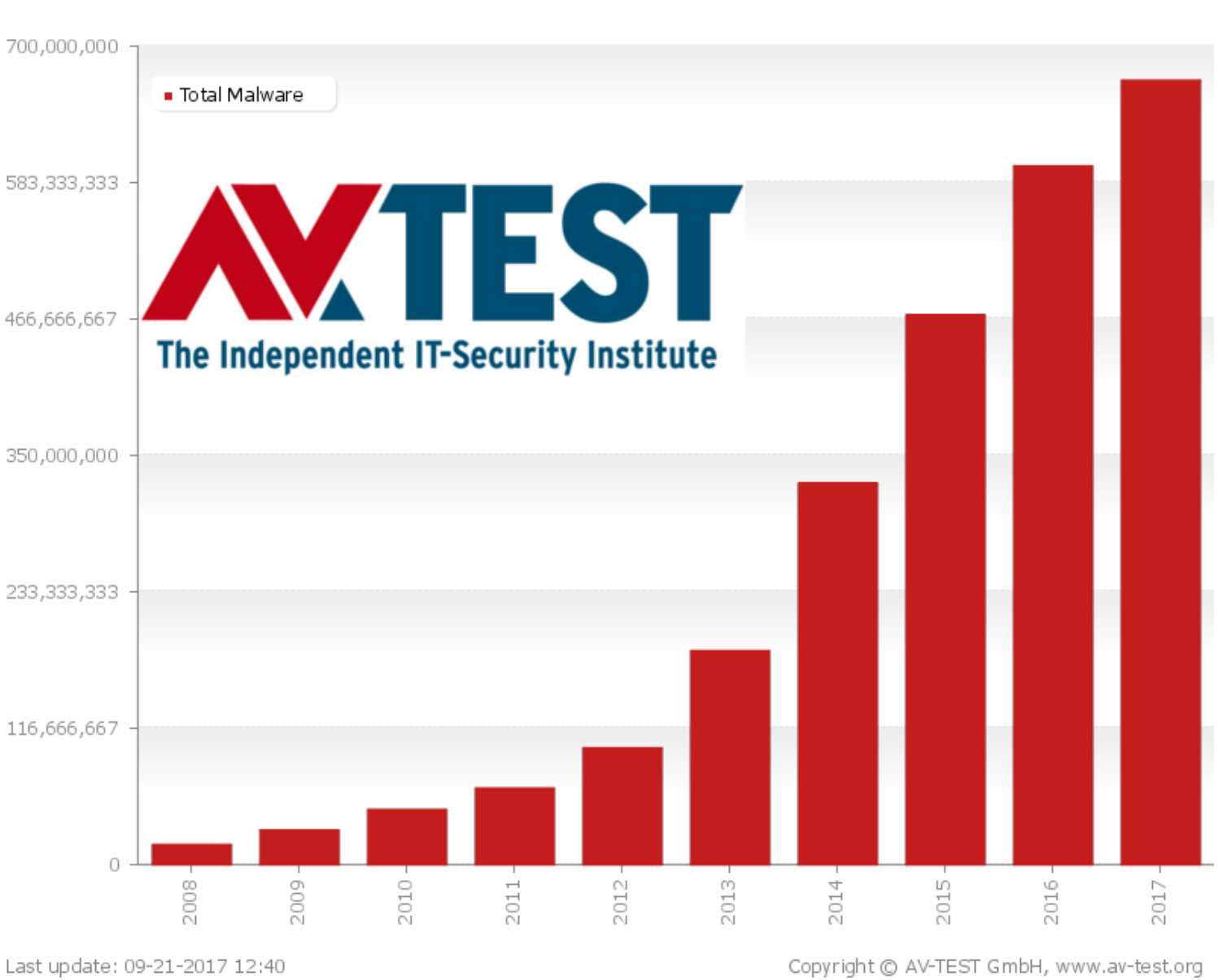
Fabio Rosato

Advisors: Camil Demetrescu, Daniele Cono D'Elia

October 24, 2017

Master of Science in Engineering in Computer Science

Malicious software - An ever growing threat



Malware analysis

Static analysis

Dynamic analysis

Concrete execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();
    int z = y * 2 + x
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Concrete execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();   <⇦  3
    int z = y * 2 + x
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Concrete execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();    <⇦  3
    int z = y * 2 + x      <⇦  7
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Concrete execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();     <= 3
    int z = y * 2 + x       <= 7
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;     ✓
    }
}
```

Symbolic execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();
    int z = y * 2 + x
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Symbolic execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();   ⇦  λ
    int z = y * 2 + x
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Symbolic execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();     ⇦  λ
    int z = y * 2 + x       ⇦  λ * 2 + 1
    if (z == 13) {
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Symbolic execution - Example

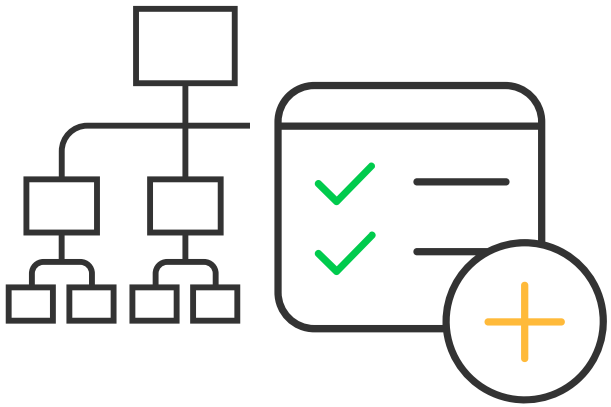
```
int foo() {
    int x = 1;
    int y = read_int();        ⇦  λ
    int z = y * 2 + x          ⇦  λ * 2 + 1
    if (z == 13) {             ⚠  Forking!
        return ERROR;
    } else {
        return SUCCESS;
    }
}
```

Symbolic execution - Example

```
int foo() {
    int x = 1;
    int y = read_int();          ⇦  λ
    int z = y * 2 + x            ⇦  λ * 2 + 1
    if (z == 13) {               ⚠  Forking!
        return ERROR;            ⇨  if λ * 2 + 1 == 13
    } else {
        return SUCCESS;          ⇨  if λ * 2 + 1 != 13
    }
}
```

Symbolic execution of malware

Symbolic execution applications have been practically exclusively confined to general software testing, with excellent results.



The ability to potentially cover all possible execution paths and subsequently identify the corresponding concrete input values that would elicit them, make it an ideal instrument for the study of the trigger-based behaviors extremely common in malware.

Malware evasion defined

Malware evasion

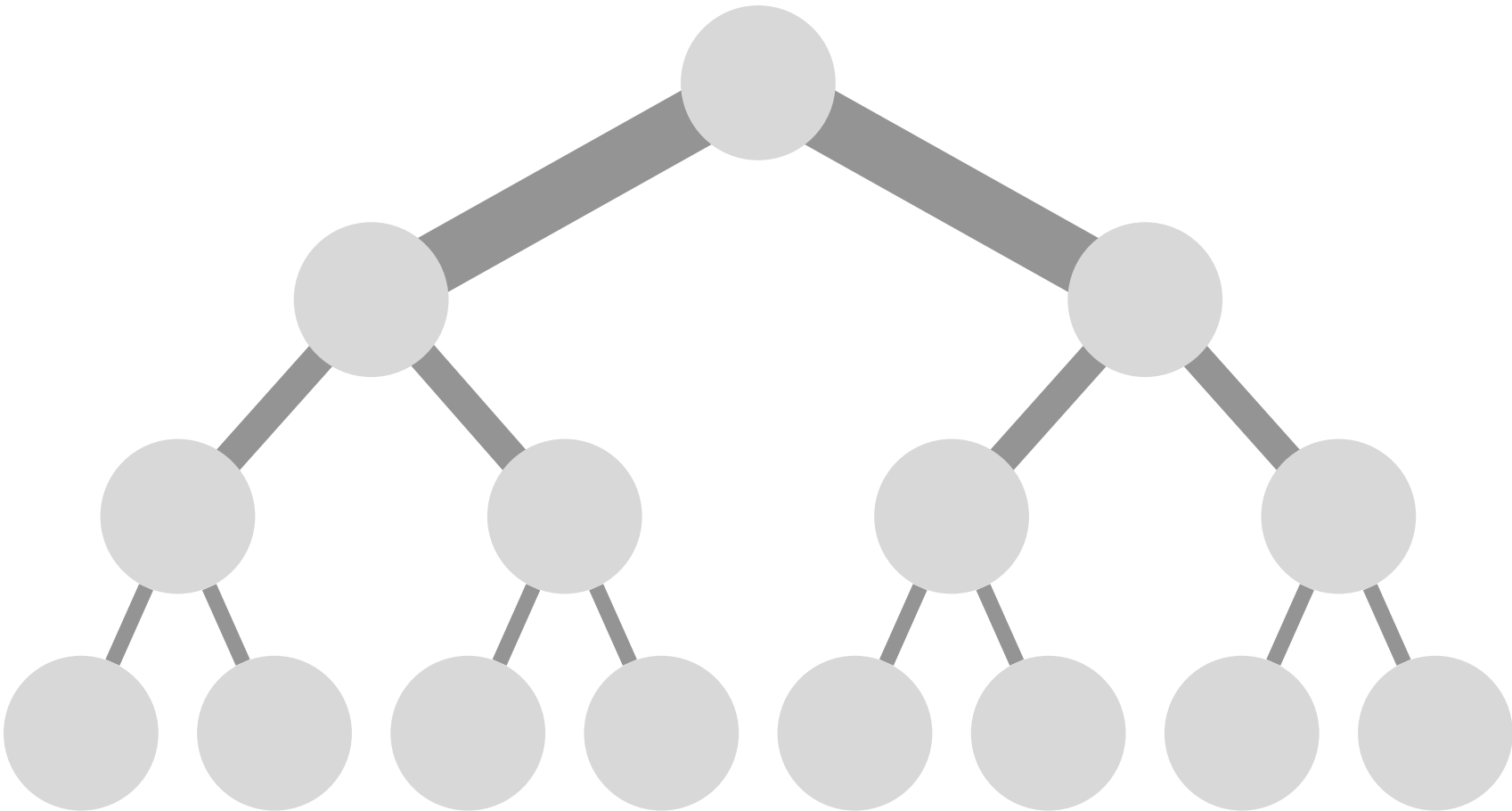
The set of techniques employed by malware to avoid being detected by an automated dynamic analysis product.

An evasive malware is a malware that exhibits no malicious behavior in a *sandbox*, but that infects the intended target.

```
if observed:
    act_innocent()
else:
    do_bad_things()
```

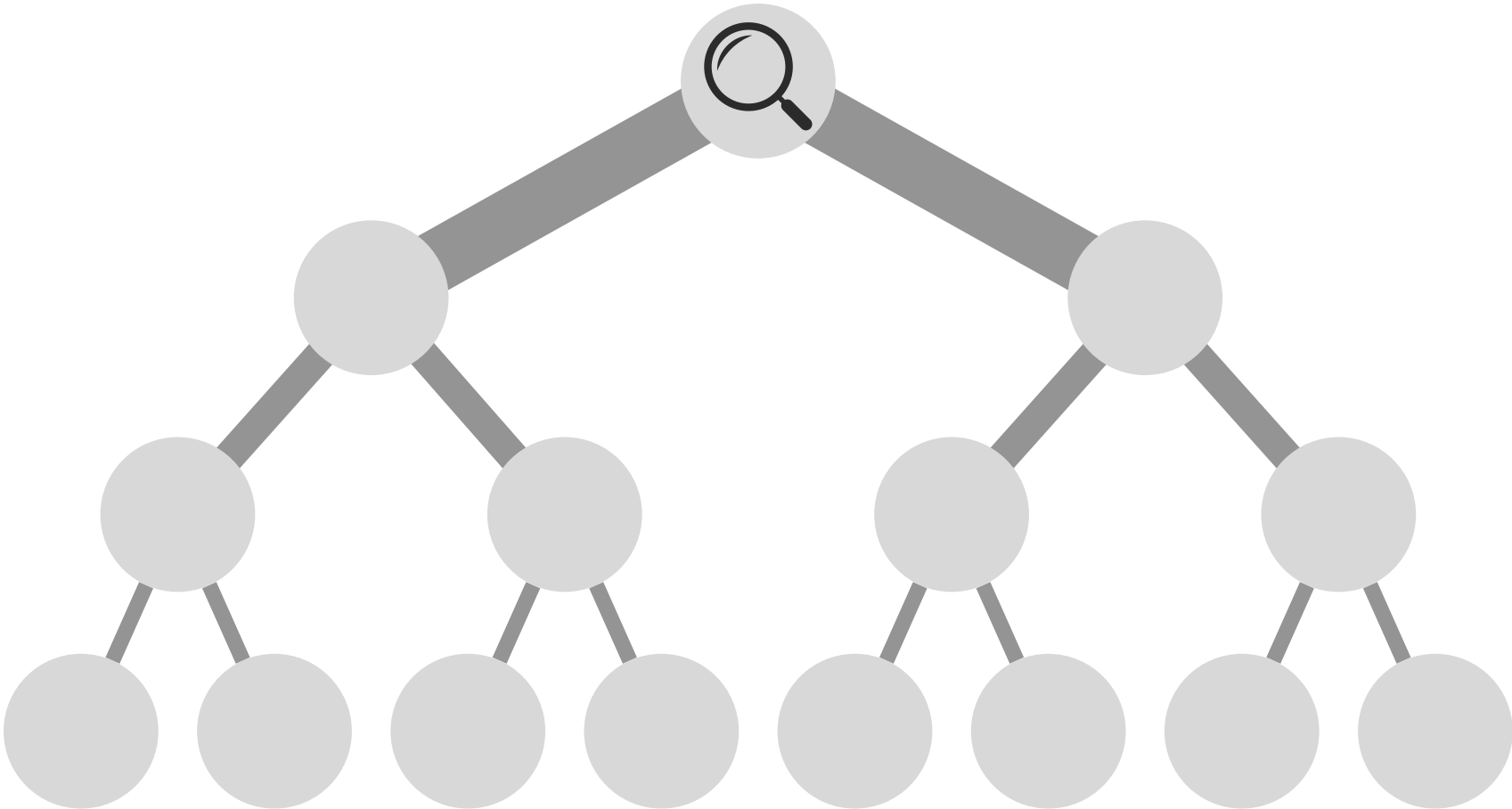
The problem with malware evasion

Evasive checks represent critical **branching** points



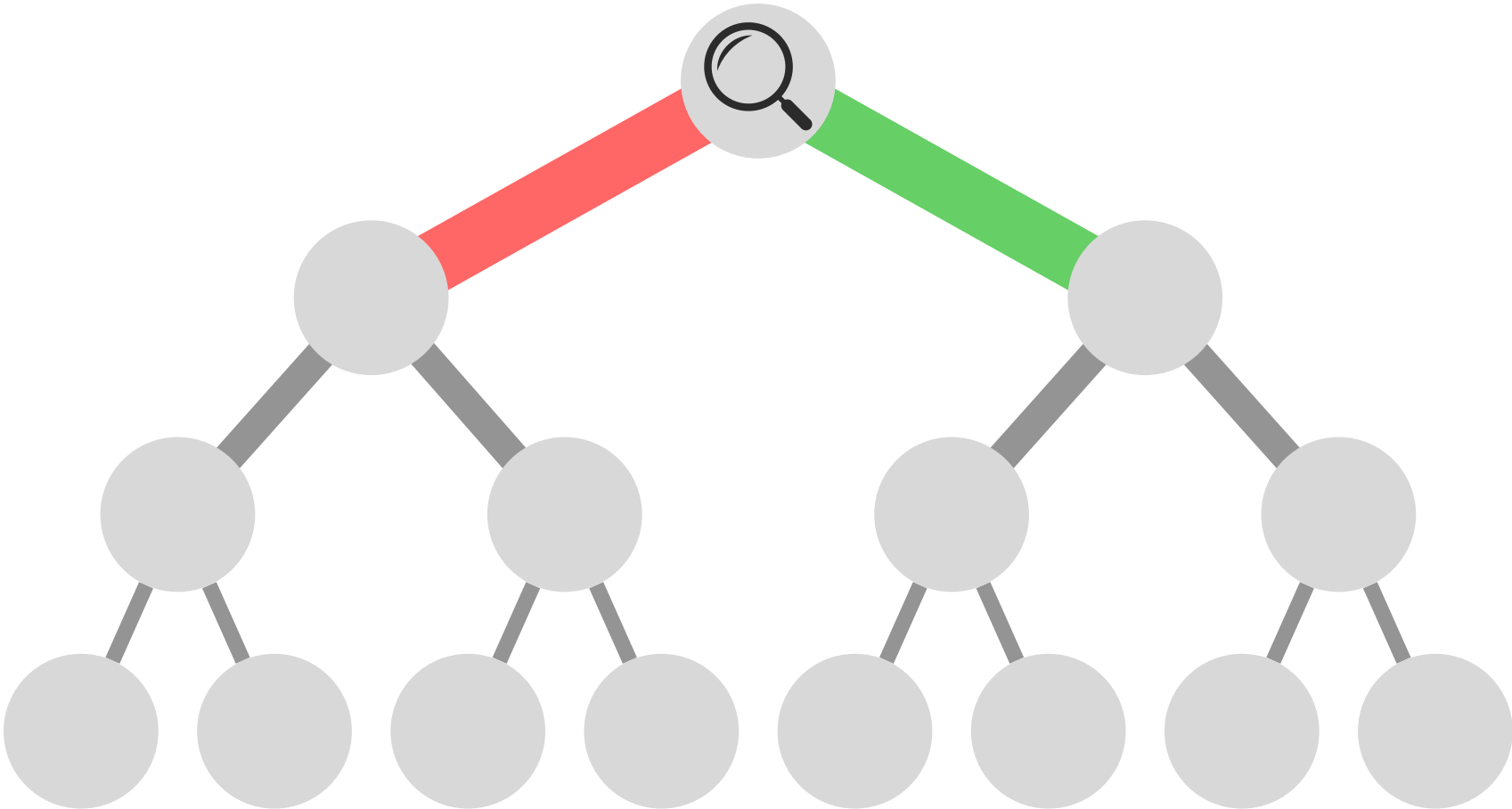
The problem with malware evasion

Evasive checks represent critical **branching** points



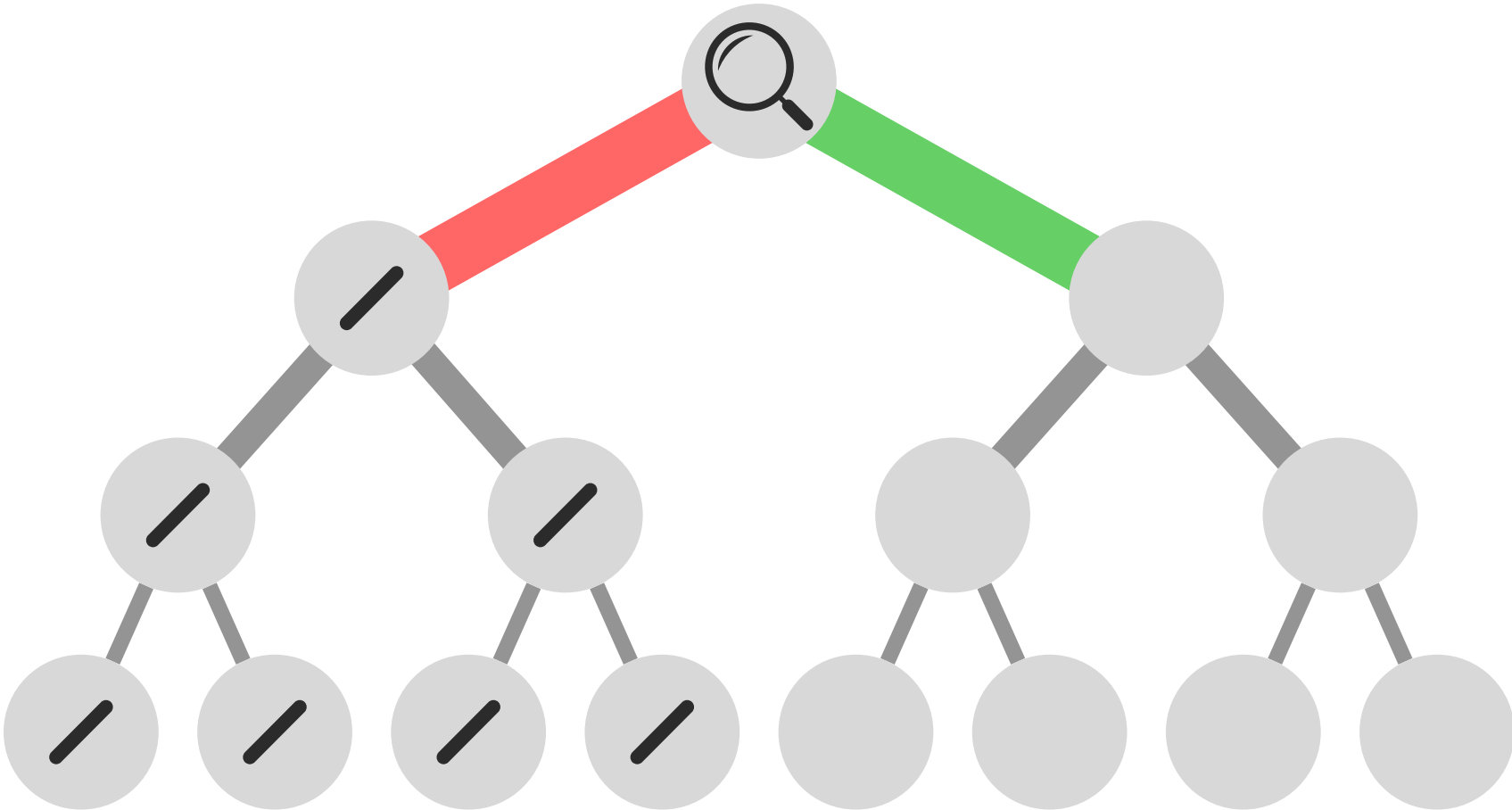
The problem with malware evasion

Evasive checks represent critical **branching** points



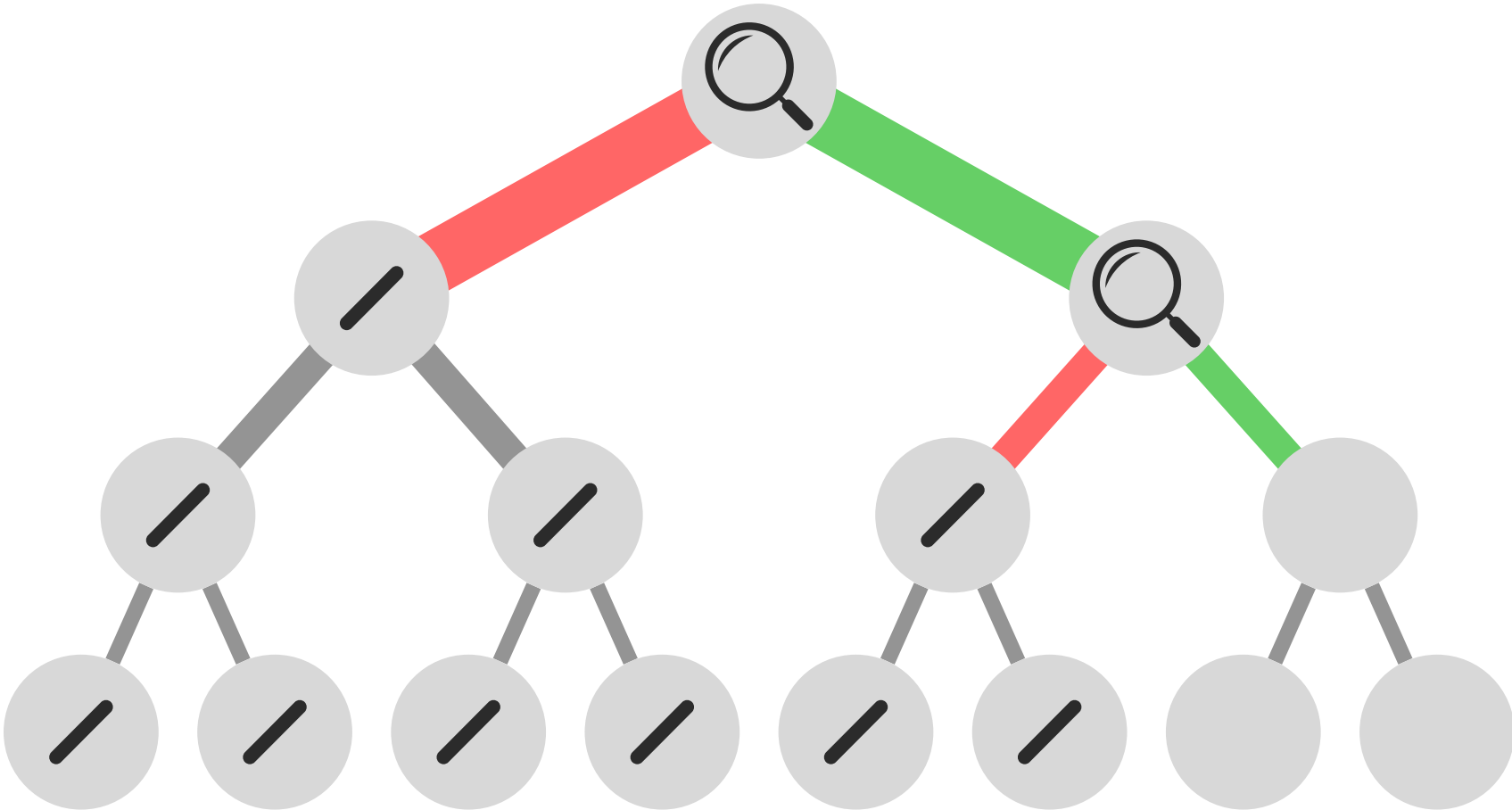
The problem with malware evasion

Evasive checks represent critical **branching** points



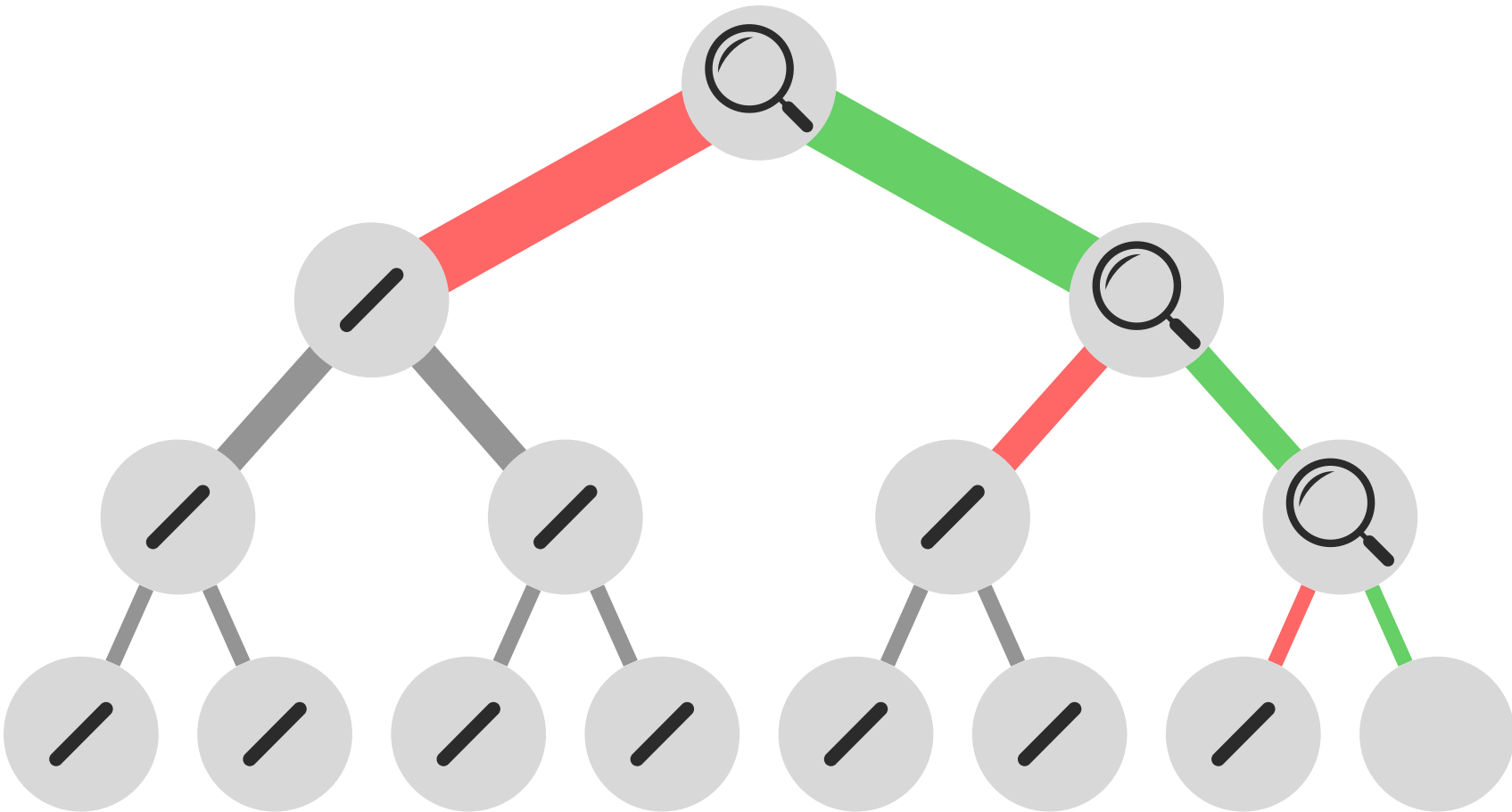
The problem with malware evasion

Evasive checks represent critical **branching** points



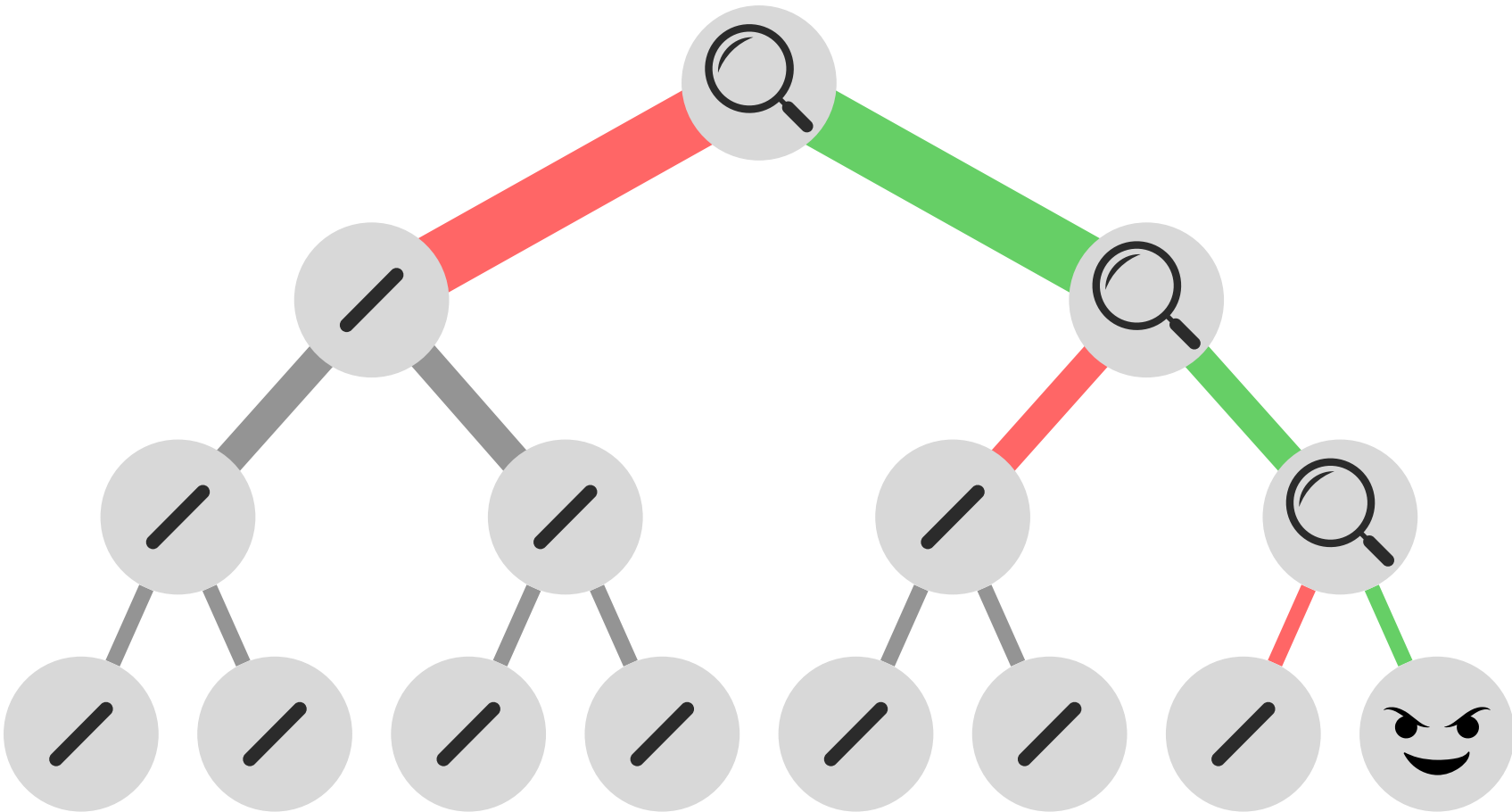
The problem with malware evasion

Evasive checks represent critical **branching** points

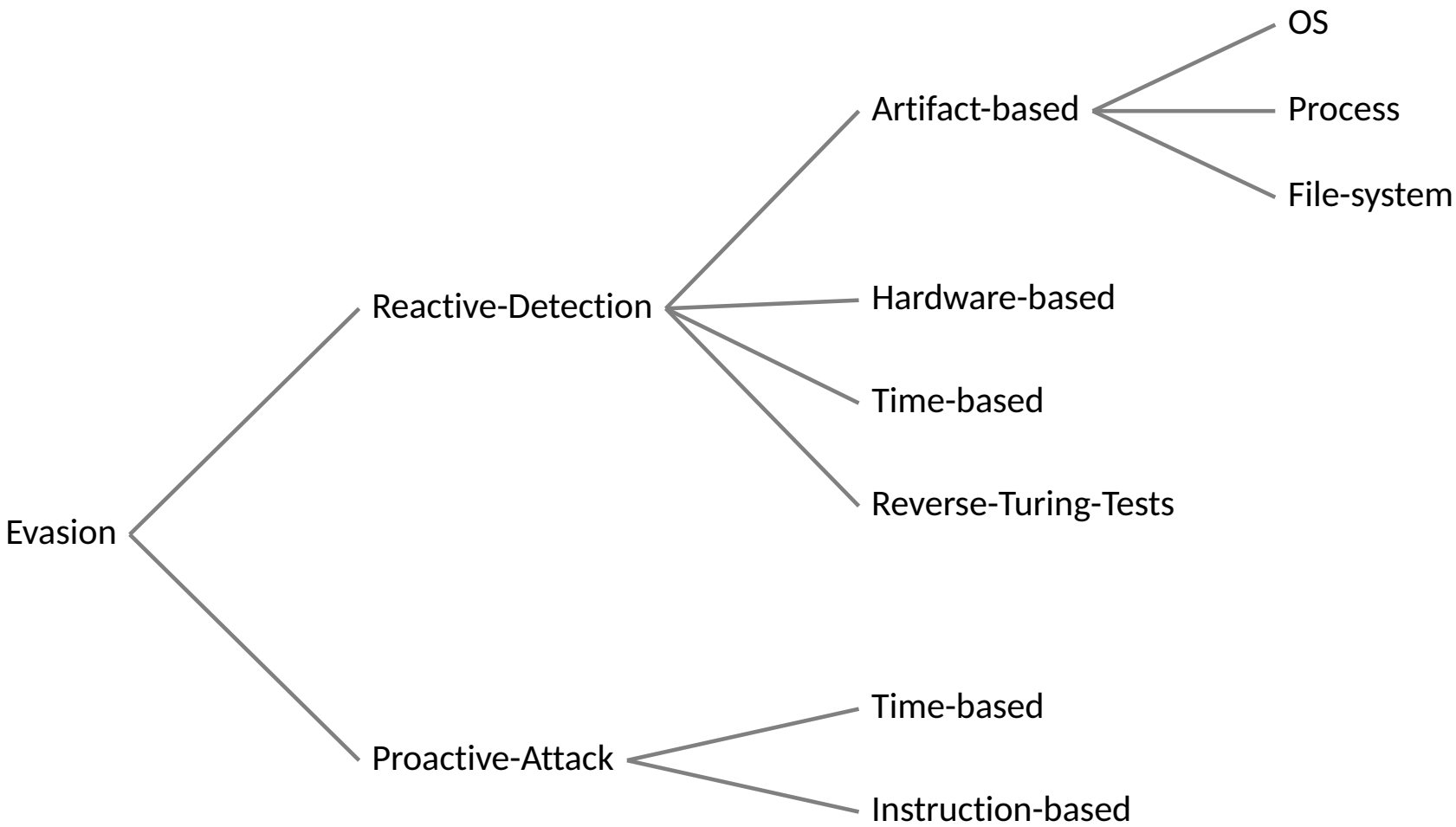


The problem with malware evasion

Evasive checks represent critical **branching** points



Contribution: malware evasion categorization



Contribution: common Windows API interaction patterns

Category / Check	Involved APIs
OS artifacts detection	
Registry key presence	`RegOpenKeyEx`
Registry key value	`RegOpenKeyEx` `RegQueryValueEx`
User name	`GetUserName`
Process enumeration	`CreateToolhelp32Snapshot`
Windows	`FindWindow`

Contribution: common Windows API interaction patterns

Category Check	Involved APIs
Process artifacts detection	
Hooks	–
Injected DLLs	`GetModuleHandle`

Debugging	`IsDebuggerPresent`
	`OutputDebugString`
	`CheckRemoteDebuggerPresent`

Contribution: common Windows API interaction patterns

Category Check	Involved APIs
File system artifacts detection	
File system artifact presence	`GetFileAttributes` `CreateFile`
Execution path	`GetModuleFileName`
Common file names	`GetLogicalDriveStrings` `GetFileAttributes`

Contribution: common Windows API interaction patterns

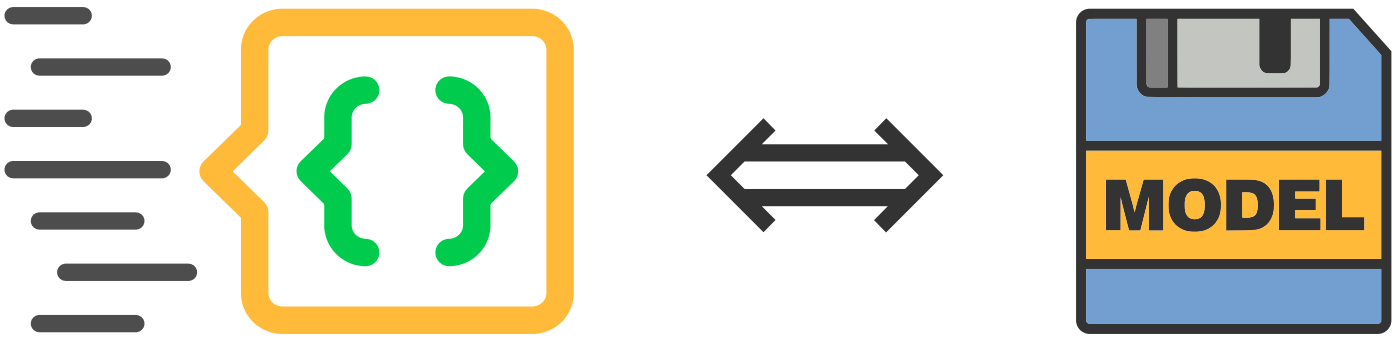
Category / Check	Involved APIs
Hardware-based detection	
Single-CPU	`GetSystemInfo`
Small amount of RAM	`GlobalMemoryStatusEx`
Small drive size	`DeviceIoControl` `GetDiskFreeSpaceEx`
CPUID fingerprinting	`CPUID`
Network adapter details	`GetAdaptersAddresses`

Contribution: common Windows API interaction patterns

Category / Check	Involved APIs
Time-based detection	
Sleep patched	`GetTickCount` `Sleep`
Uptime	`GetTickCount`
RDTSC timing	`RDTSC`
Network adapter details	`GetAdaptersAddresses`
Reverse Turing tests	
Mouse movement	`GetCursorPos`

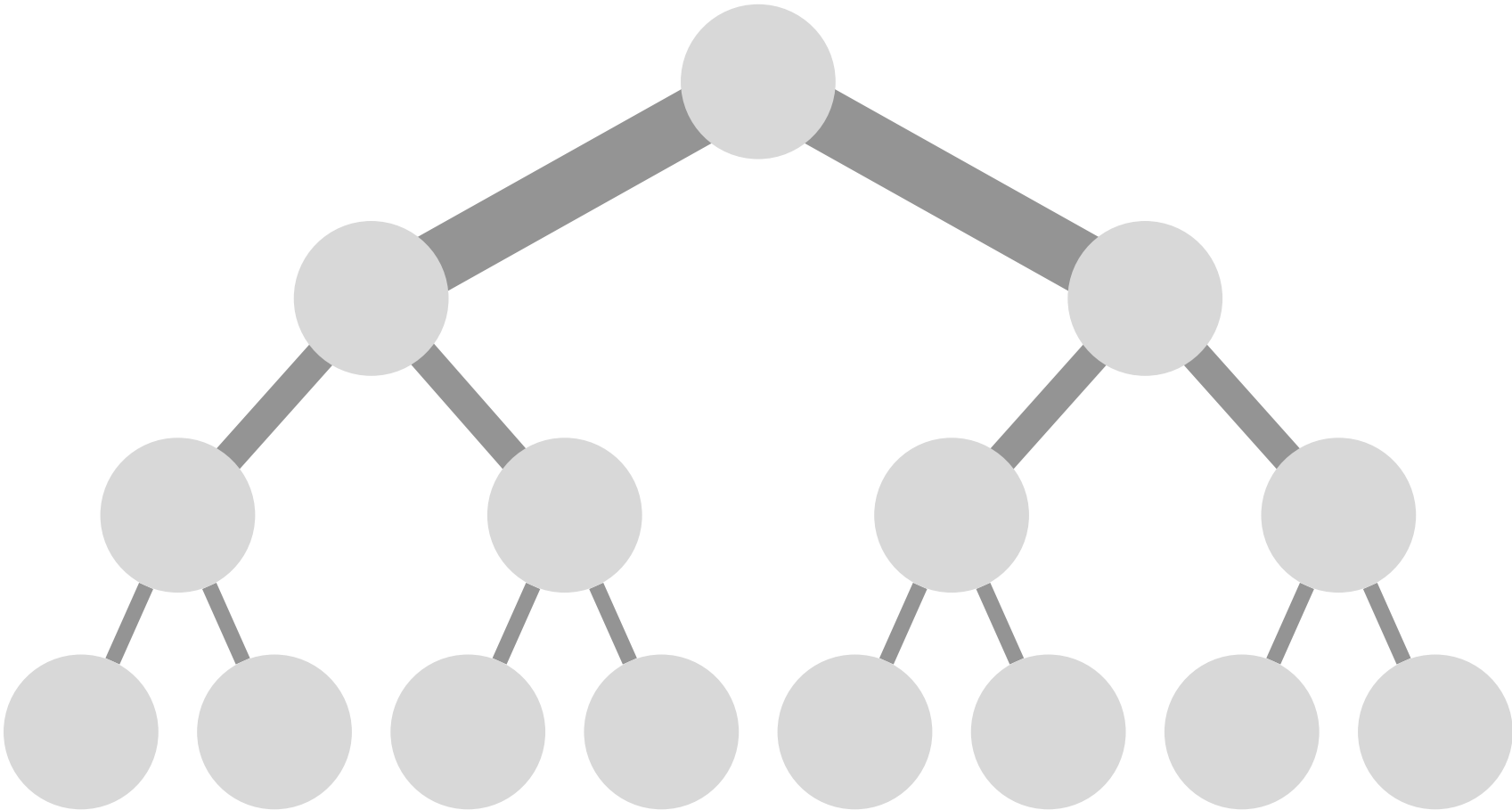
Contribution: Angr anti-evasion extension

Angr: an open source python framework for analyzing binaries that combines both static and dynamic symbolic analysis (`http://angr.io/`).



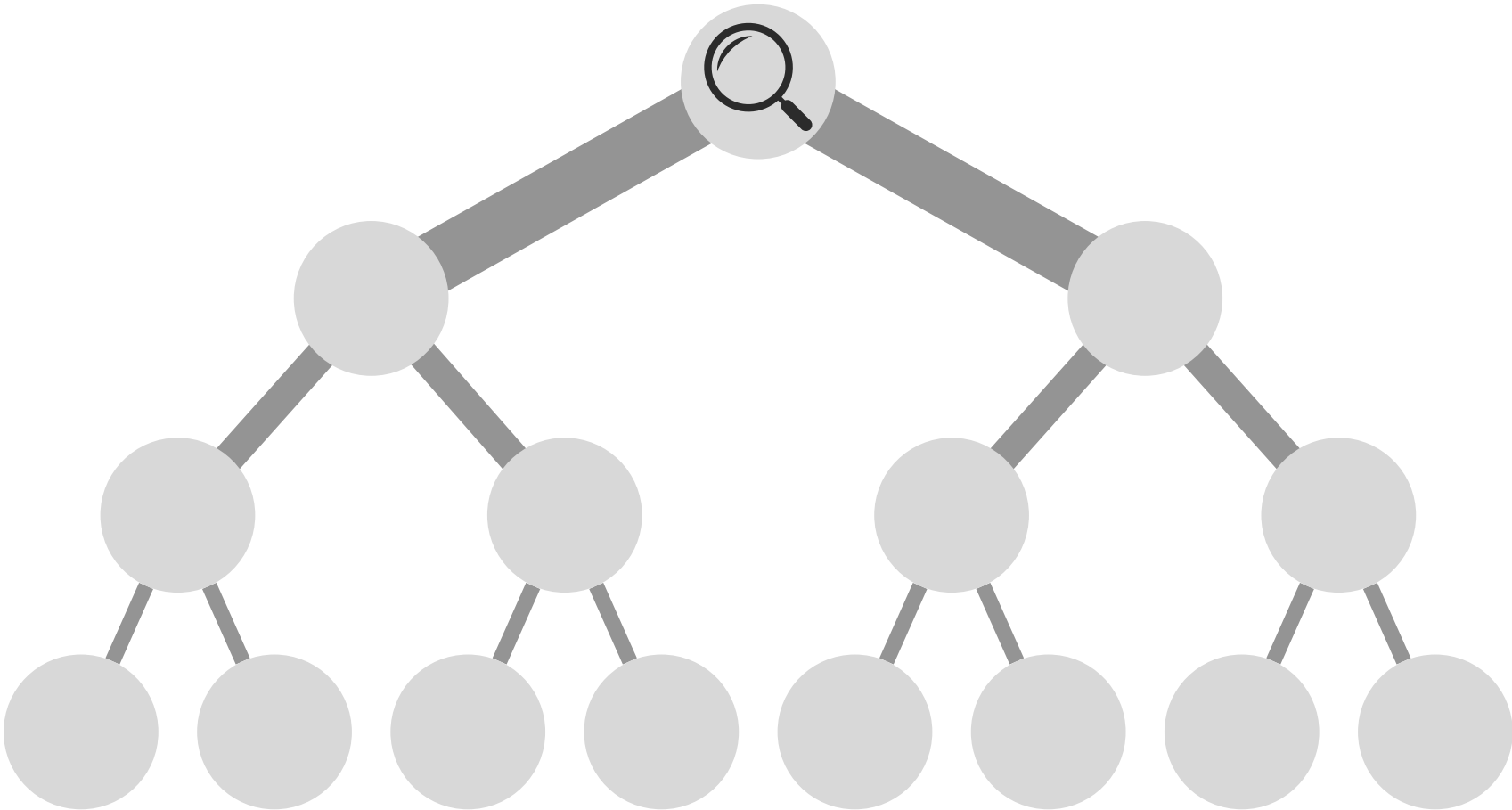
Contribution: Angr anti-evasion extension

Constrain the symbolic exploration to pass the evasive detection checks



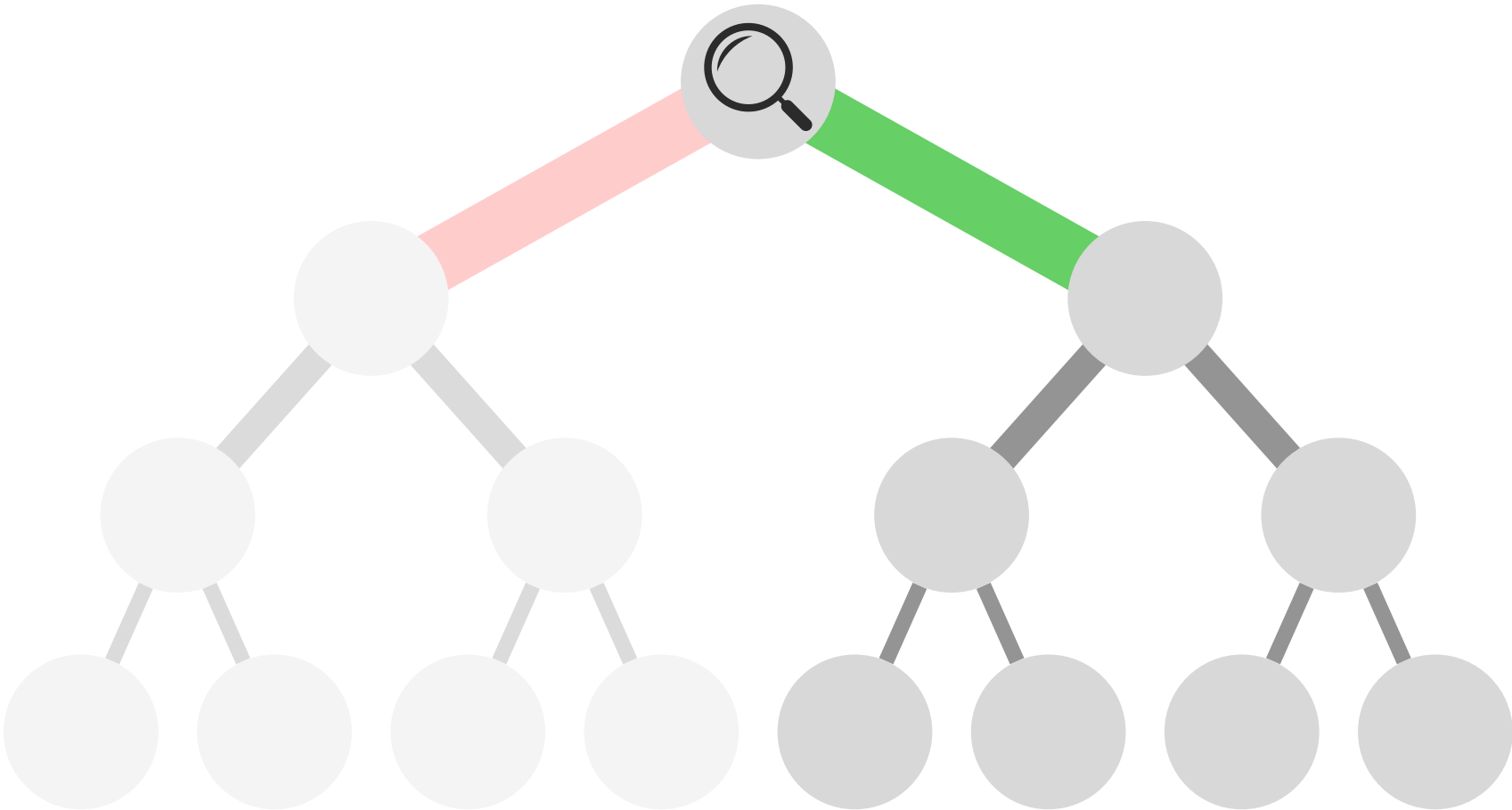
Contribution: Angr anti-evasion extension

Constrain the symbolic exploration to pass the evasive detection checks



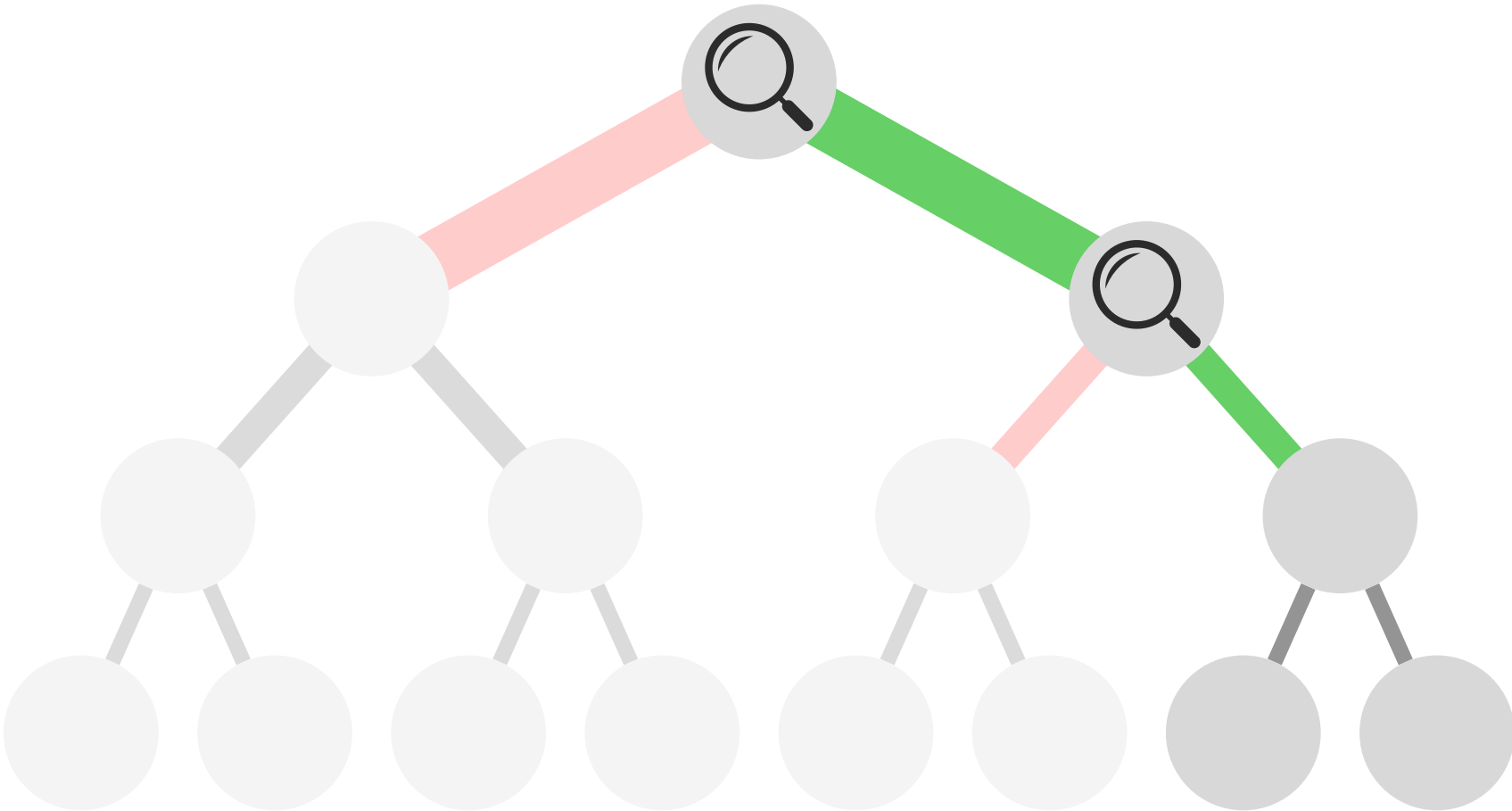
Contribution: Angr anti-evasion extension

Constrain the symbolic exploration to pass the evasive detection checks



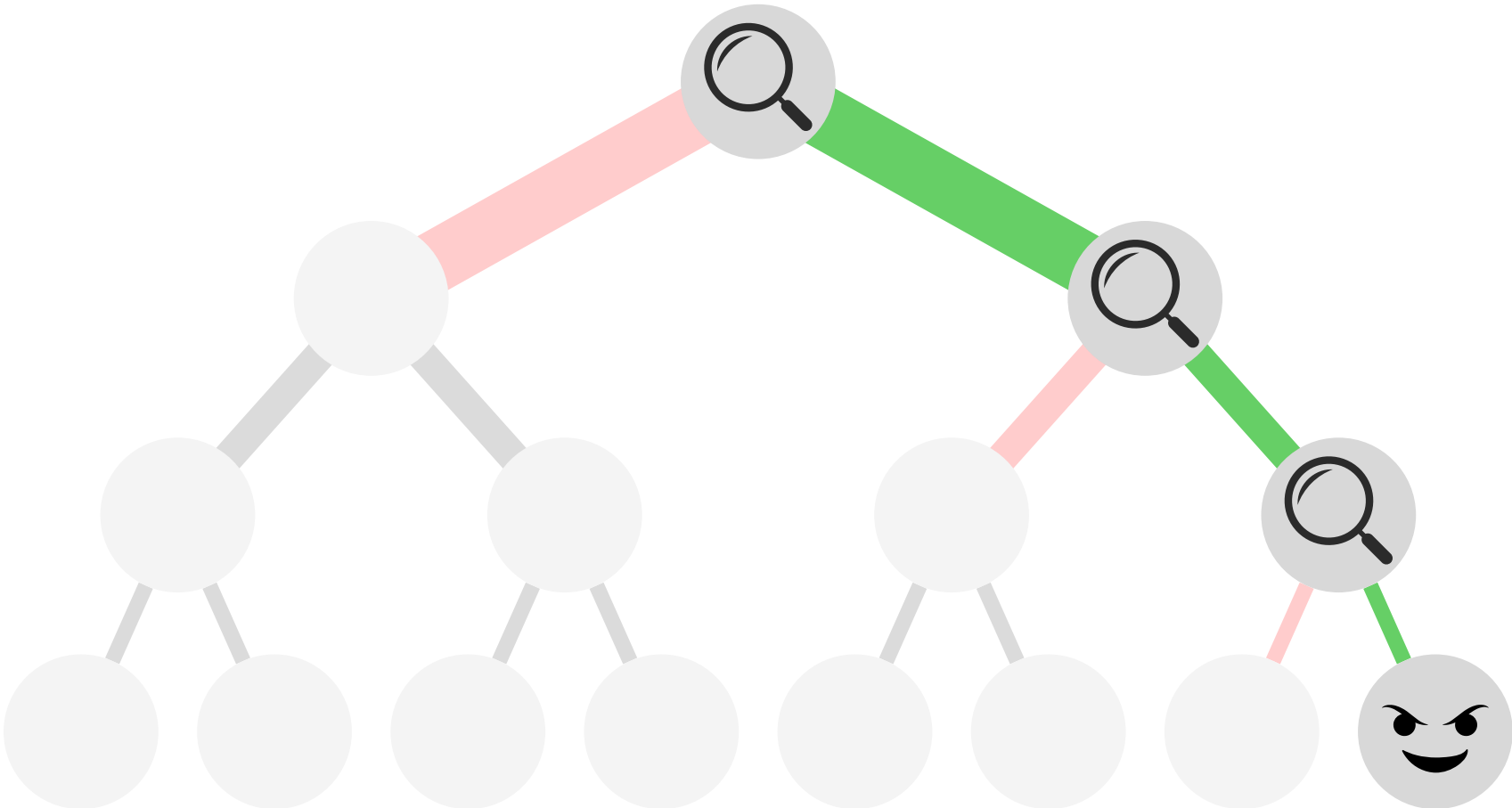
Contribution: Angr anti-evasion extension

Constrain the symbolic exploration to pass the evasive detection checks



Contribution: Angr anti-evasion extension

Constrain the symbolic exploration to pass the evasive detection checks

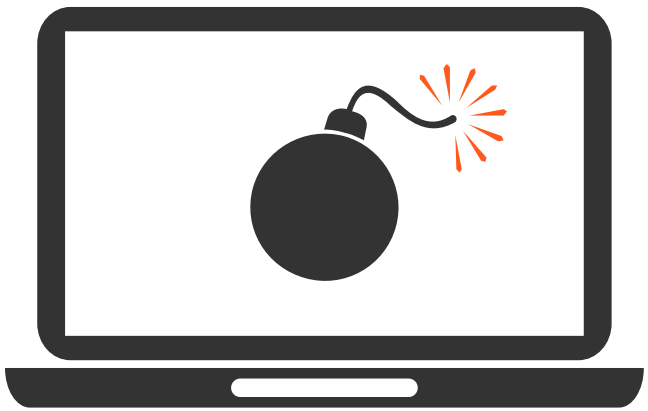


Extension evaluation - Paranoid Fish

Paranoid Fish: open source tool that demonstrates several techniques employed by malware families to detect whether they are being executed in an analysis environment, be it a debugger, a VM, or a sandbox.

57 different checking functions

Unaided



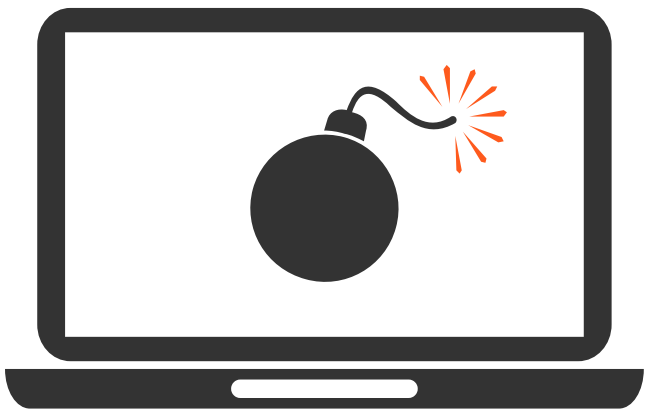
Aided by extension

Extension evaluation - Kasidet

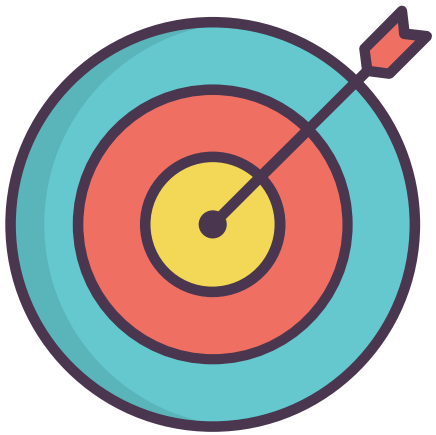
Kasidet malware: backdoor, gathers information and communicates with a C&C server, allowing a remote attacker to take over the infected machine by exchanging files with the server and executing shell commands. Sample `de1af0e97e94859d372be7fcf3a5daa5`.

17 different detective evasion functions

Unaided



Aided by extension



Conclusions - Summary

Thesis focus:

improving the applicability of symbolic execution to the malware domain by patching evasion techniques.

Contributions:

- high level categorization of evasive techniques
- Windows API interaction patterns study for the most widespread detective evasion techniques
- Angr anti-evasion extension

Thesis, slides, and code are all available at

`https://github.com/fabros/angr-antievasion`

Thank you!

Fabio Rosato

`rosato.1565173@studenti.uniroma1.it`