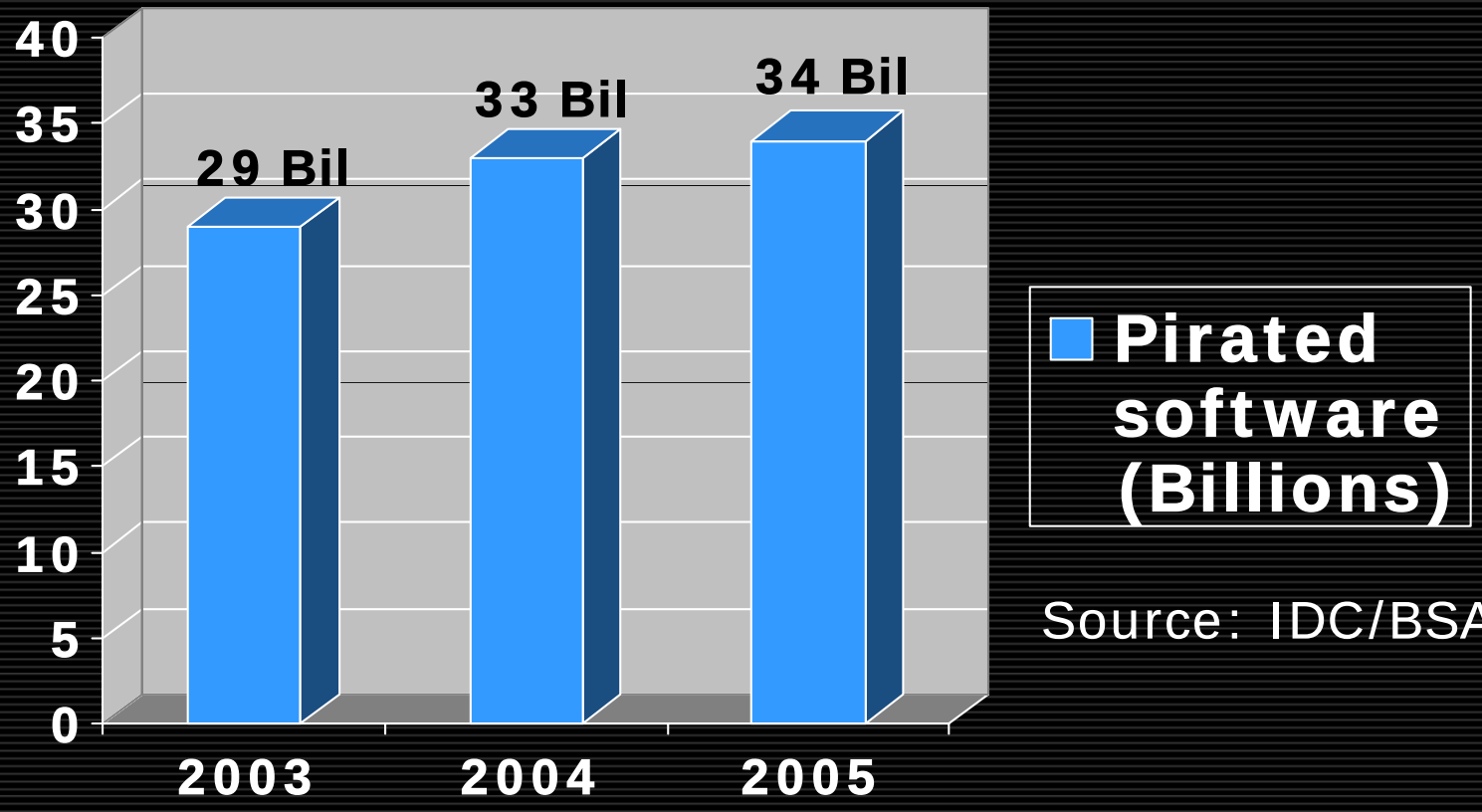
Delayed and Controlled Failures in Tamper-Resistant Software

Gang Tan[*], Yuqun Chen[#], and Mariusz H. Jakubowski[#]

[*]Boston College

[#]Microsoft Research

Software Piracy Worldwide



Source: IDC/BSA

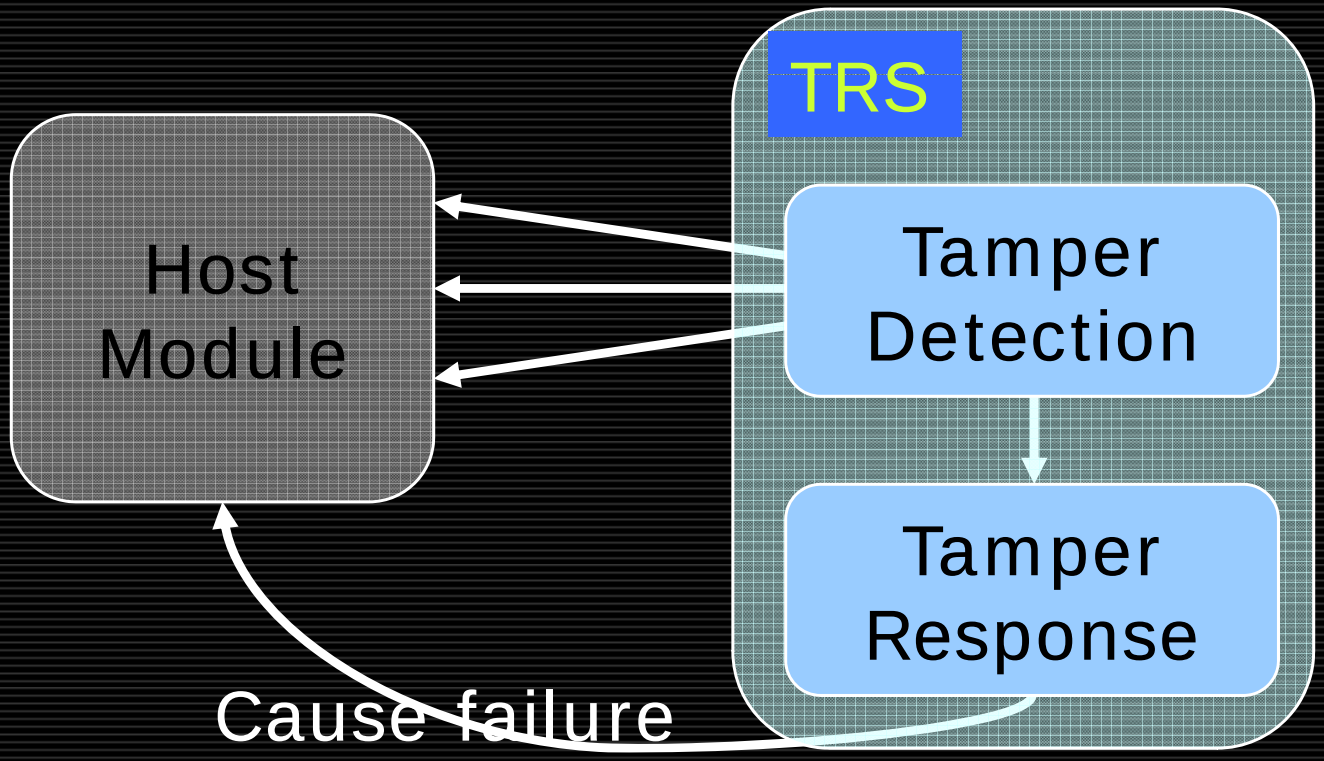
One out of every three copies of PC software is obtained illegally.

Software Tampering Attacks

```
...
if (noLicense()) {
    notifyAuthority();
    terminateProgram();
}
...
```

Bypass copyright-protection code

Tamper-Resistant Software (TRS)



How to hide the TRS module in the host module?

Tamper Detection

- Tamper Detection
  - Detect tampering of the host module
- A fair amount of R&D exists
  - Static checksum
    - [Chang & Atallah 01], [Horne et al. 01]
  - Oblivious hashing (dynamic trace)
    - [Chen et al. 03]
  - ...

What about Tamper Response?

- What kind of response
  - Make the tampered module unusable/fail
  - Notify the authority
- An inadequate tamper-response module can become the weakest link
- Little research on the tamper-response module
  - FADE protection, which degrades game play over time
  - Informal and ad hoc

A Naïve Response Module

```
if tampered_with() then 1/0
```

- Fail at the place where response happens
- Divide-by-zero is unusual
- Fail immediately

Possible Attacks

- Trace back to the TRS code and remove it
- Scan for the divide-by-zero
- Effect observed immediately

The Guideline of Good Response Systems

- A good response system should not reveal information of the TRS module

An Ideal Response System

- Spatial separation between response and failure
  - Respond in one place, but its effect appears in other places.
- Temporal separation between response and failure
  - Insert delays to thwart the process of tampering
  - Psychologically frustrate adversaries

An Ideal Response System, Cont'd

- Stealthy response code
  - Blend into the host module
  - Can't use automatic scanning tools to locate response code
- Predictable/controllable failure
  - The tampered module should eventually fail, with high probability

Related Work

- Separating tamper detection from response
  - Some commercial use
  - Ad hoc and not systematic
- Tamper resistance, and software obfuscation
  - Pointer-based opaque predicates
    - Collberg, Thomborson, et al.
  - Complication of control flow
    - Wang et al.

Outline

- Motivation
- Principles of ideal tamper response
- Our tamper-response mechanism
  - Evades hacker removal by introducing delayed, probabilistic failure
- System implementation and experiments
- Future work

Starting Insight

- Corrupt the program's internal state
  - Program is likely to fail
  - Stealthy: similar to a "programming error"
  - Spatial and temporal separation
    - If we carefully choose where and when to corrupt

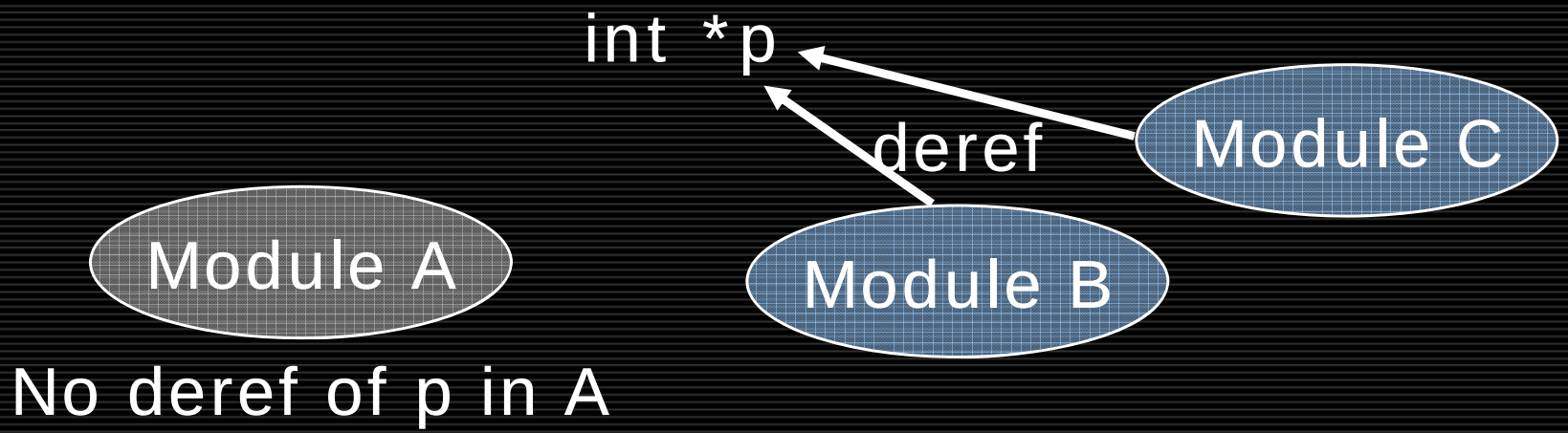
Corrupt Pointer Variables

- Corrupt the program's own variables
- Non-pointer variables
  - Hard to predict the behavior after corruption
- Corruption of pointer variables will cause its dereference to fail

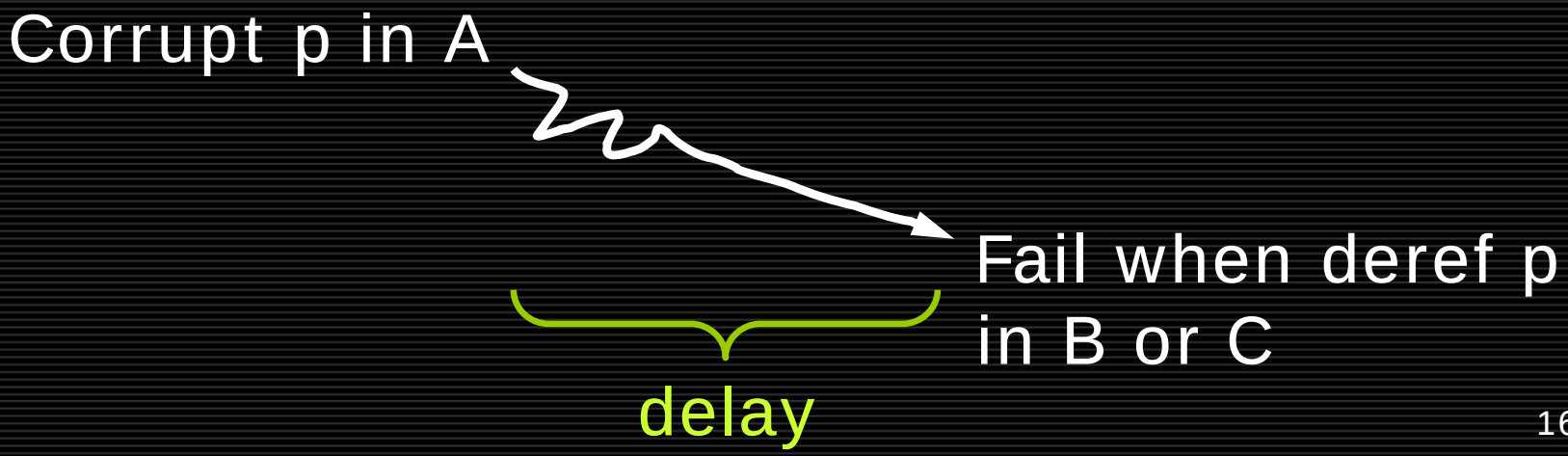
But Not Every Pointer

- Local pointers: not much delay
  - Have to be corrupted locally
  - Use in the local scope
- Corruption of global pointers can achieve wide delay

Corruption of Global Pointers



No deref of p in A



Creating Global Pointers

```
int a;
int *p = &a;
void f() {  a = 3;}
             *p
void main() {
        f();
        printf("a = %i\n",  a);
}                            *p
```

- Can create arbitrary # of global pointers
- Can control the failure site

Outline

- Motivation
- Principles of ideal tamper response
- Our tamper-response mechanism
  - What to corrupt: global pointers
  - Creating global pointers
  - Where to corrupt?
- System implementation and experiments
- Future work

Where to Corrupt?

- Goal: Given a global pointer, identify corruption sites, to achieve desired spatial and temporal separation between corruption and failure
- Failure sites: where the pointer is derefed
- Corruption sites: anywhere in the program
  - It's a global variable

Identify Corruption Sites

- Searching unit: functions
- Suppose
  - The set of all functions is G
  - The set of functions that deref p is

$$F = \{f_1, f_2, \ldots, f_n\}$$

- Goal: find functions in G such that there is a desired delay before any function in F is invoked

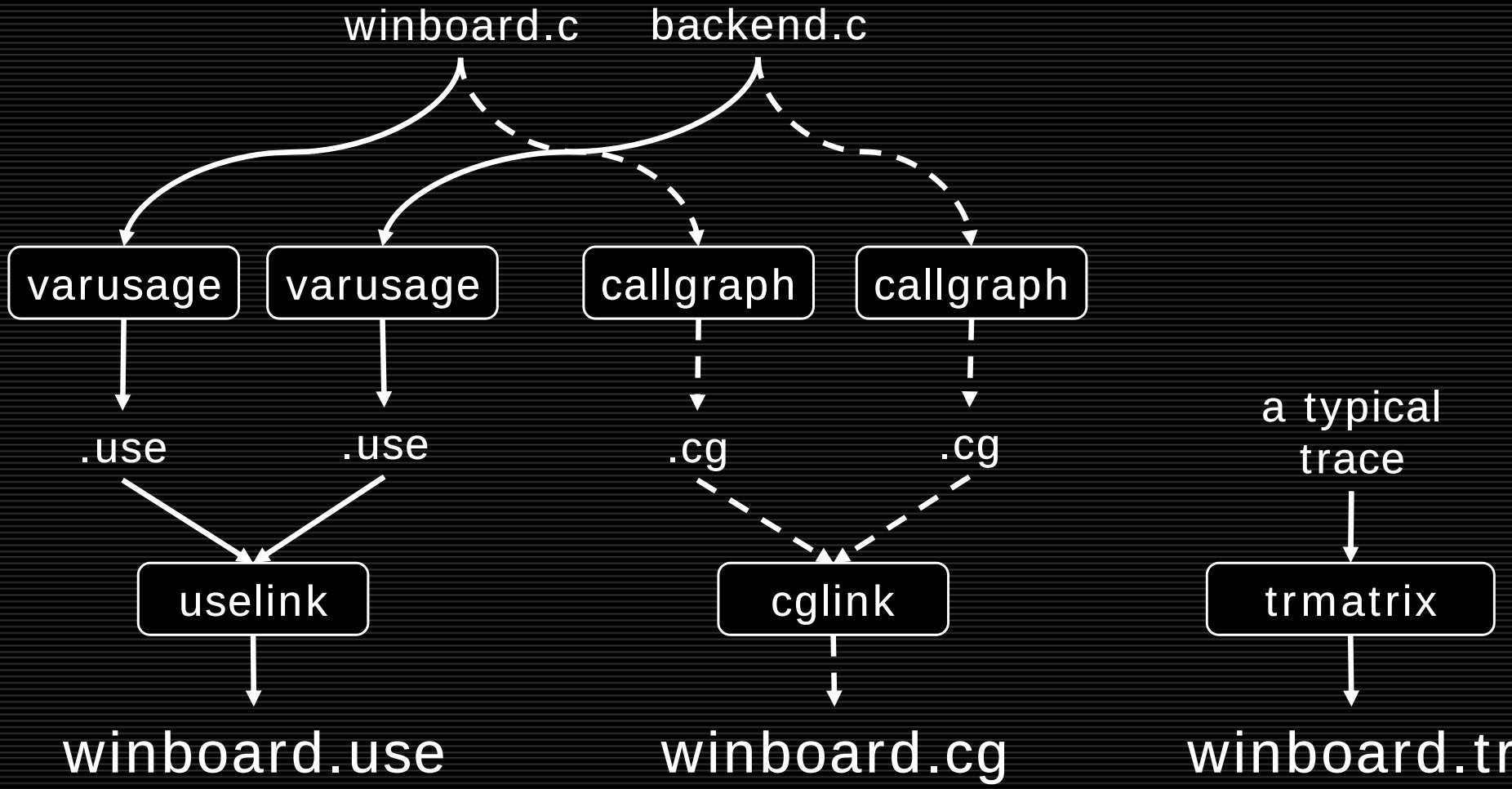
Searching Algorithm

- Make sure when failure happens, the corruption site (function) is not in the call stack
  - Use call graphs to rule out all ancestors of failure sites
- Rule out those functions that are too close to the failure sites
  - Measure the average distance between two functions in a typical dynamic trace

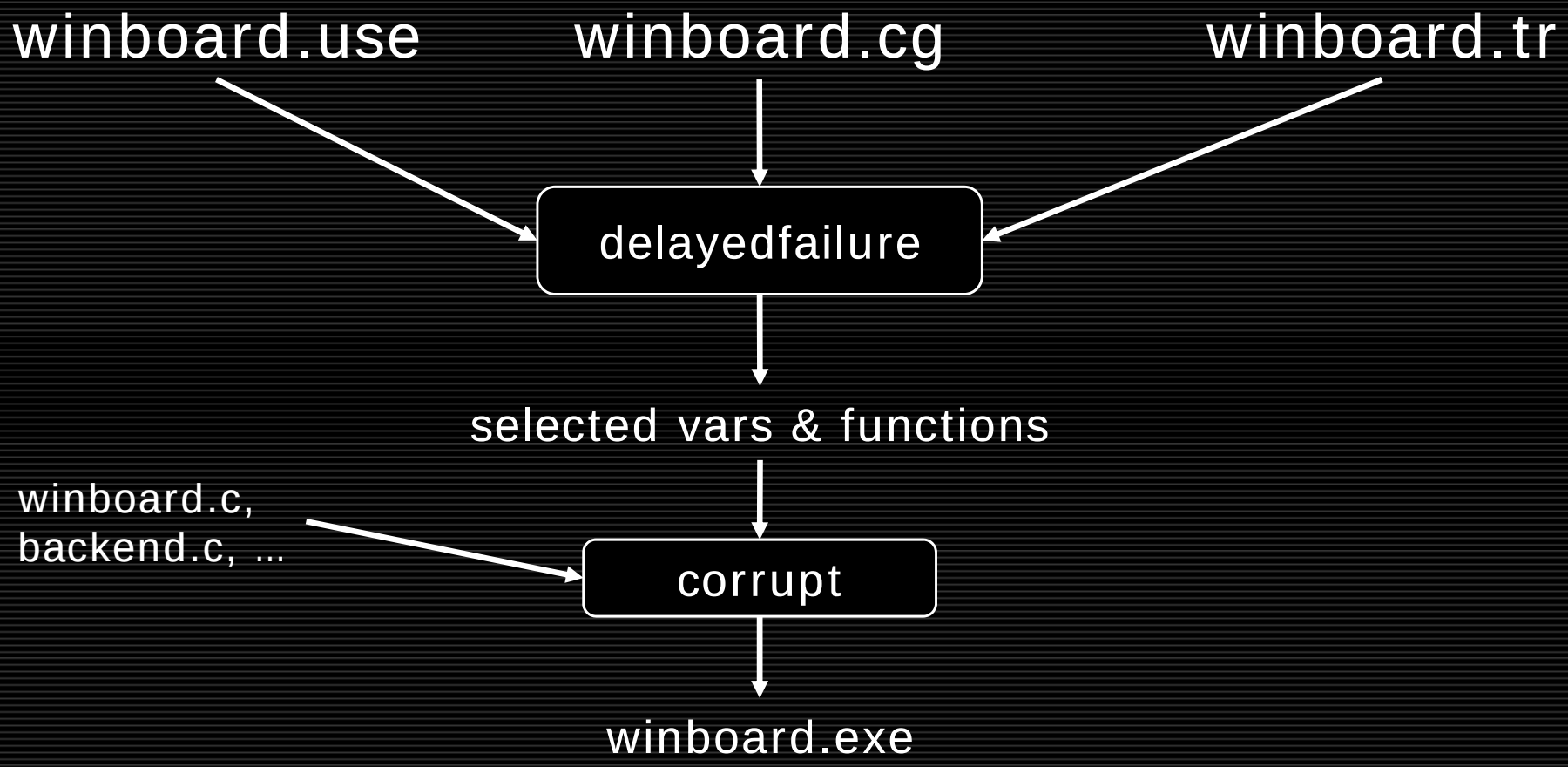
Outline

- Motivation
- Principles of ideal tamper response
- Our tamper response mechanism
- System implementation and experiments
- Future work

System Implementation



System Implementation, Cont'd



Preliminary Experiments

- Winboard
  - 27,000 lines of C code
  - 297 global variables
- After the instrumentation of our system
  - ~ 10^4 number of function calls between corruption and failure

Possible Attacks

- Track pointers in the program
- Counter measures
  - Should be combined with other code-obfuscation techniques
  - Embed multiple pointer corruptions
  - Use pointers that get frequently updated
  - Corruption using one set of pointers, but failure through another set

Ongoing (Future) Work

- Graceful degradation
  - More stealthy
  - The cumulative effect makes the program unusable
- Safe languages such as C# and Java
  - The idea of corrupting the program's state is still applicable

Conclusions

- Inadequate tamper response can become the weakest link of TRS
  - It should not reveal any information of the TRS
- We have proposed a scheme that evades hacker removal by introducing delayed and probabilistic failures

The End