Advanced Anti-Deobfuscation

Bjorn De Sutter

ISSISP 2017 – Paris

GHENT UNIVERSITY

About me

GHENT UNIVERSITY

- Research domain: system software
  - compilers, binary rewriting tools, whole program optimization (binary & Java), virtualization, run-time environments
  - improve programmer productivity by means of automation
  - apply tools for different applications
    - obfuscation, diversity, mitigating side channels and fault injection, ...
    - protect against exploitation of vulnerabilities (multi-variant execution)
    - generating code for accelerators
- Also worked/spent time at
- Interrupts enabled

About me

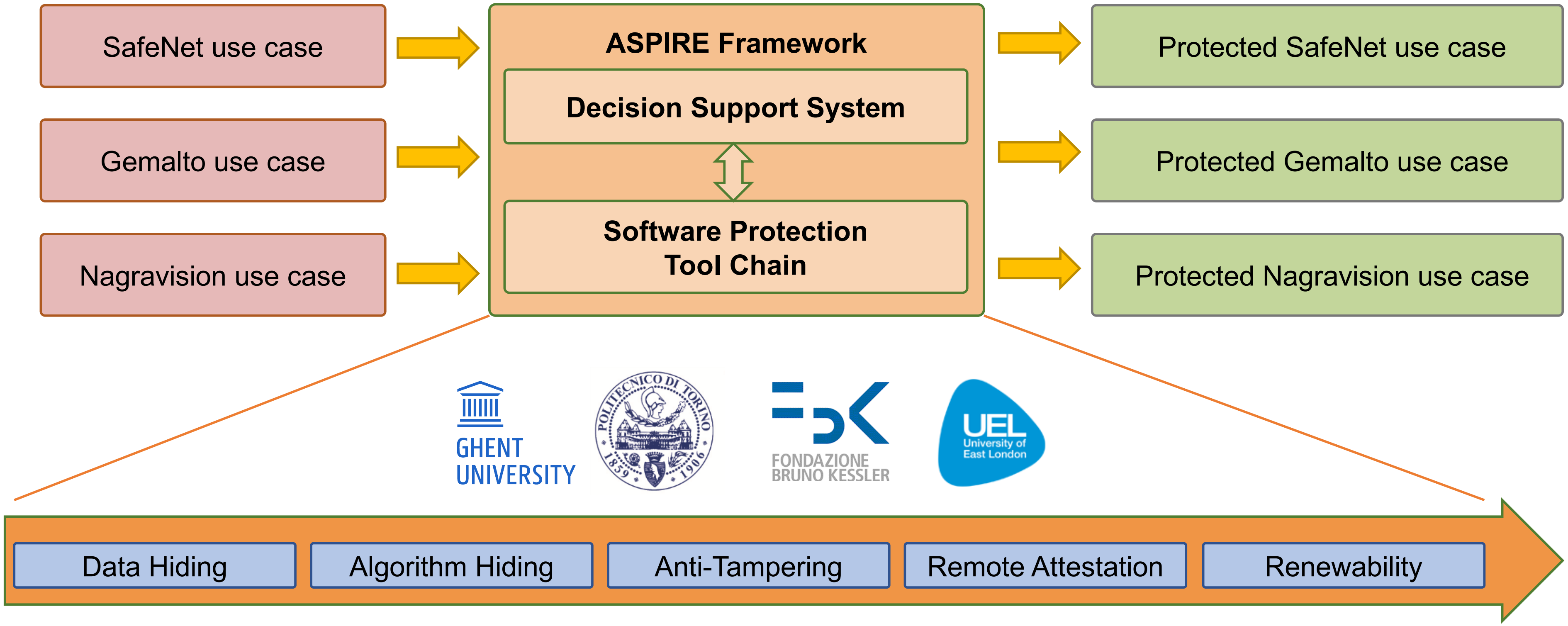
Aspire

GHENT UNIVERSITY

NAGRA

SafeNet

gemalto
security to be free



http://www.aspire-fp7.eu

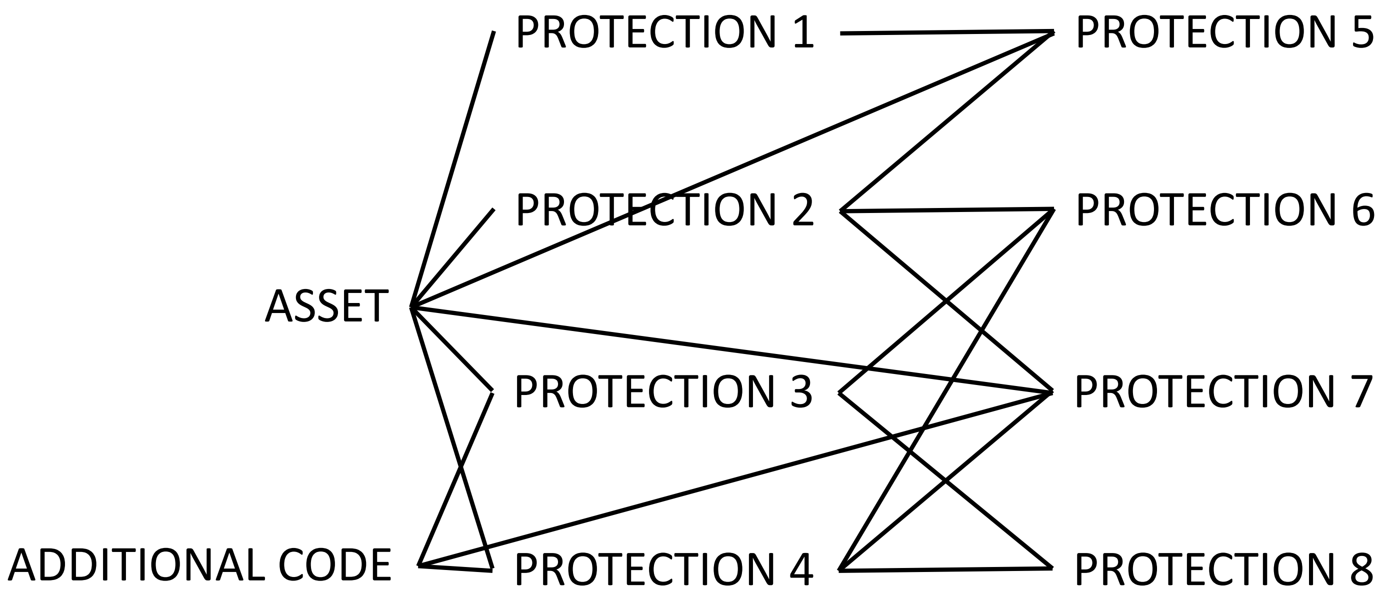
Lecture Overview

1. Basic Attacks
   - attacks on what?
   - basic attack tools & techniques
2. Defenses
   - anti-anything
3. Advanced Automated Attacks
   - generic deobfuscation
   - symbolic execution
4. Defenses
   - anti-even-more

What is being attacked?

Asset category	Security Requirements	Examples of threats
Private data (keys, credentials, tokens, private info)	Confidentiality Privacy Integrity	Impersonation, illegitimate authorization Leaking sensitive data Forging licenses
Public data (keys, service info)	Integrity	Forging licenses
Unique data (tokens, keys, used IDs)	Confidentiality Integrity	Impersonation Service disruption, illegitimate access
Global data (crypto & app bootstrap keys)	Confidentiality Integrity	Build emulators Circumvent authentication verification
Traceable data/code (Watermarks, finger-prints, traceable keys)	Non-repudiation	Make identification impossible
Code (algorithms, protocols, security libs)	Confidentiality	Reverse engineering
Application execution (license checks & limitations, authentication & integrity verification, protocols)	Execution correctness Integrity	Circumvent security features (DRM) Out-of-context use, violating license terms

What is being attacked?



1. Attackers aim for assets, layered protections are only obstacles
2. Attackers need to find assets (by iteratively zooming in)
3. Attackers need tools & techniques to build a program representation, to analyze, and to extract features
4. Attackers iteratively build strategy based on experience and confirmed and revised assumptions, incl. on path of least resistance
5. Attackers can undo, circumvent, or overcome protections with or without tampering with the code

Basic Attack Techniques

- Static attack steps: without executing the code
  - symbolic information
  - graph representations of program
- Dynamic attack steps: observing execution
  - all kinds of hooks
  - start and intervene at interfaces
  - observe features and patterns of program execution (traces)
- Hybrid attack steps: combination of both
  - e.g.: build graphs of (unpacked) code observed during execution

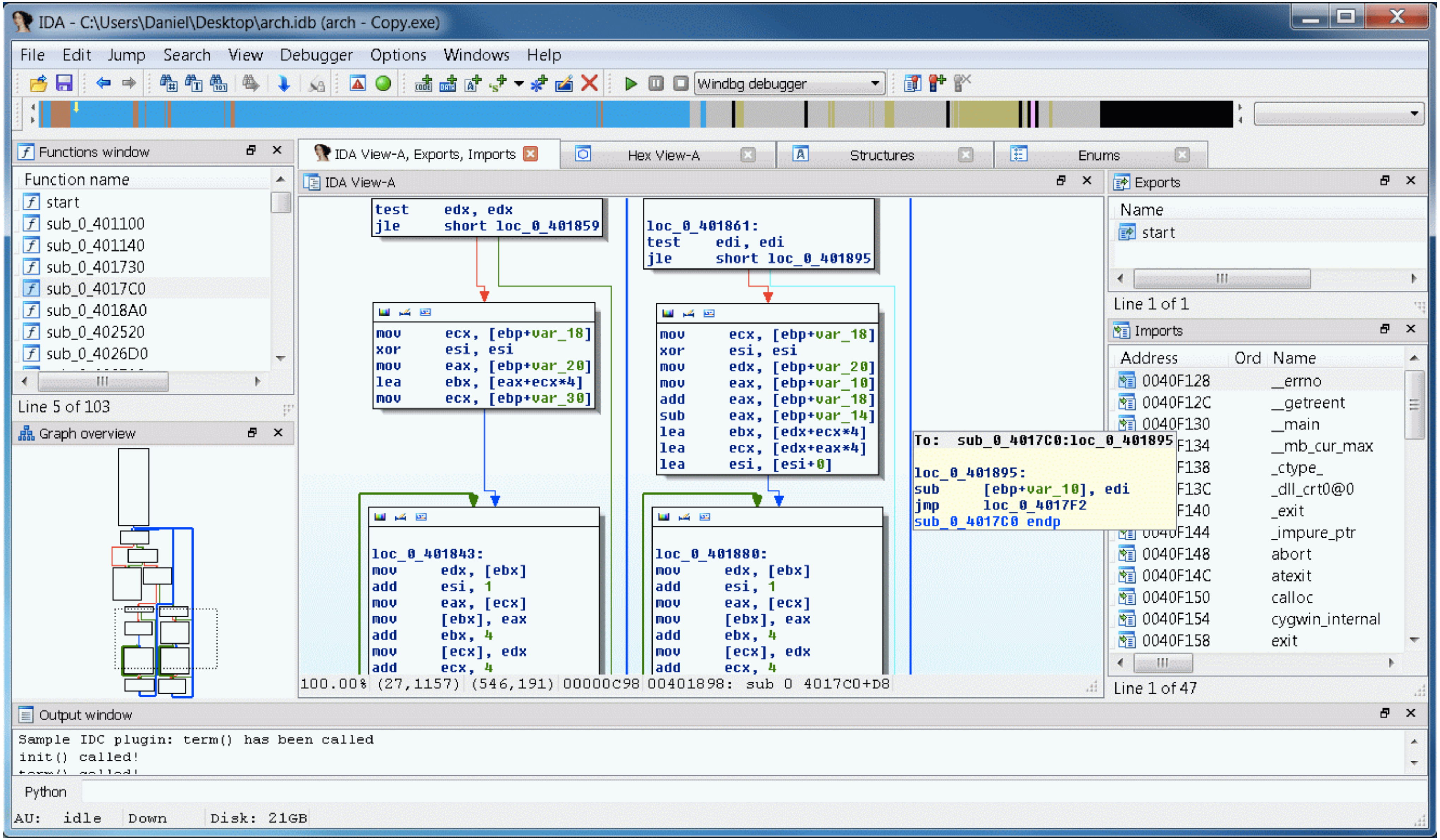
Disassemblers - 1

- IDA Pro
- Binary Ninja
- angr

- Far from perfect
  - incomplete disassembly
  - incorrect graphs (control flow, call graphs)

- Flexible and interactive
  - linear sweep, recursive descent, heuristical and manual disassembly
  - GUI
  - code annotation
  - plug-ins and scripts

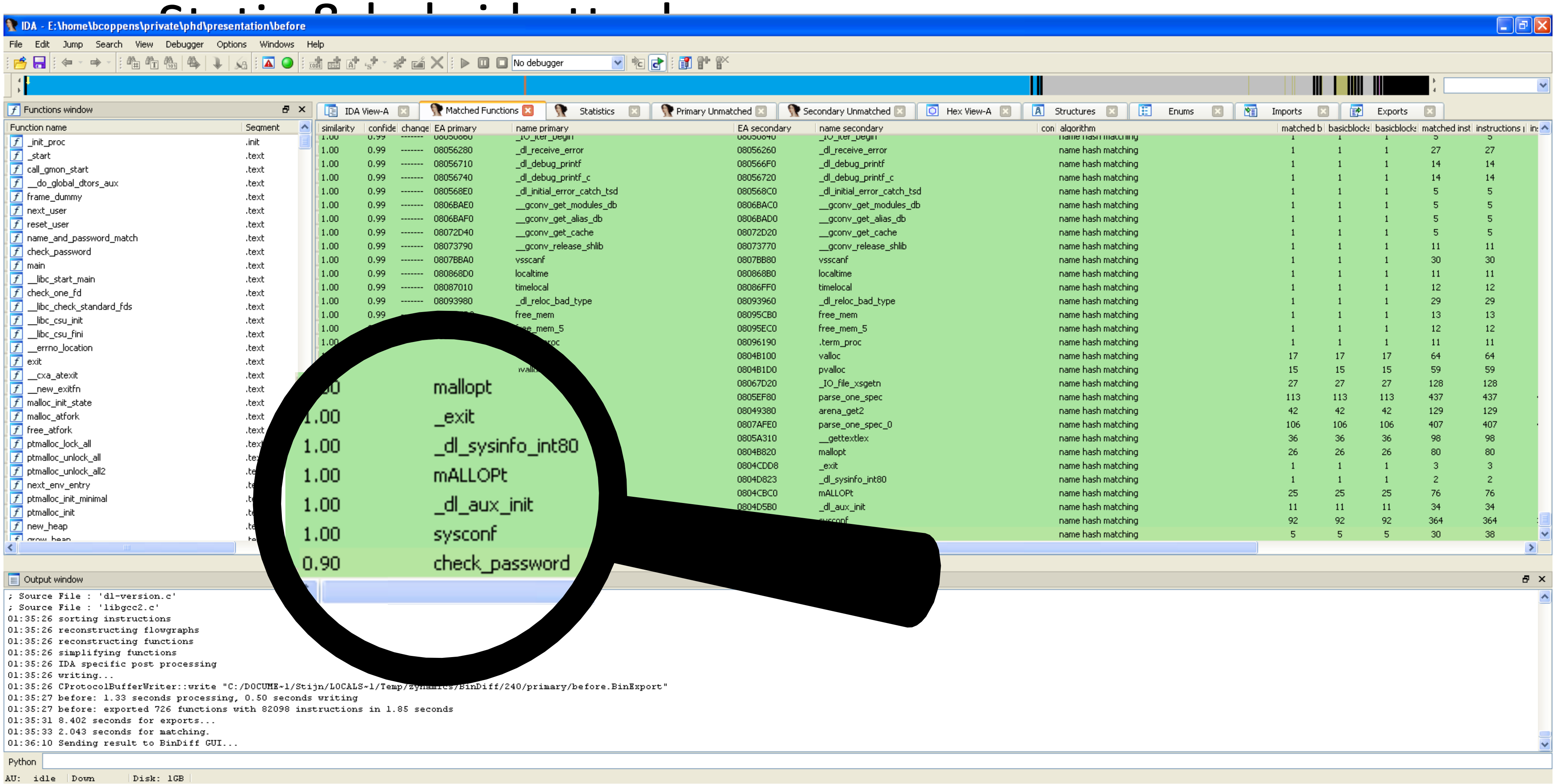
Disassemblers - 1



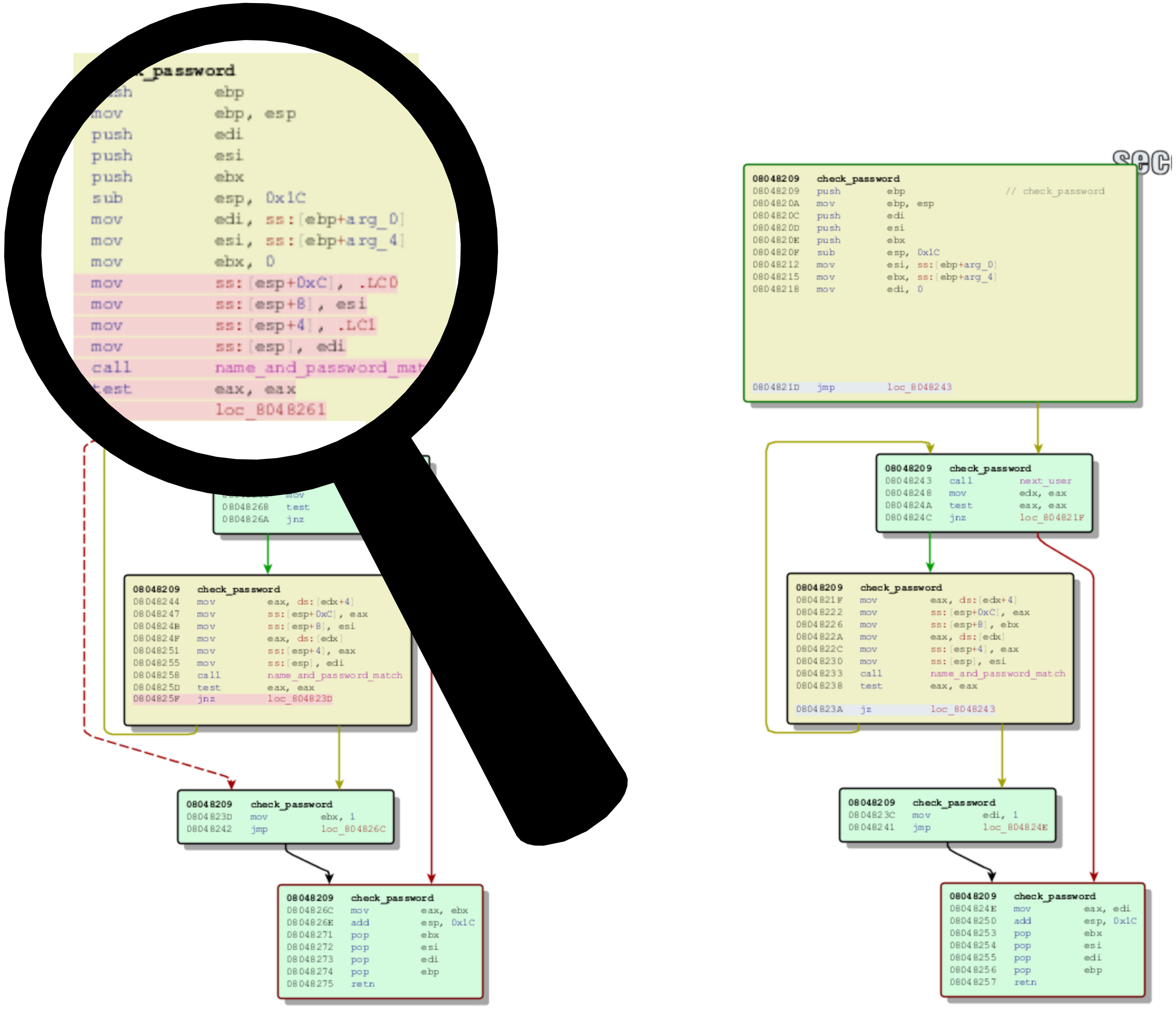
Disassemblers - 2

- Static & hybrid attacks
- Rely on many underlying assumptions
- Library detection
  - F.L.I.R.T
- Diffing tools
  - BinDiff
- Custom tools
  - detect patterns
  - undo obfuscations
  - data flow analysis
- Supports code editing
- Interfaces with (remote) debuggers

Disassemblers - 2

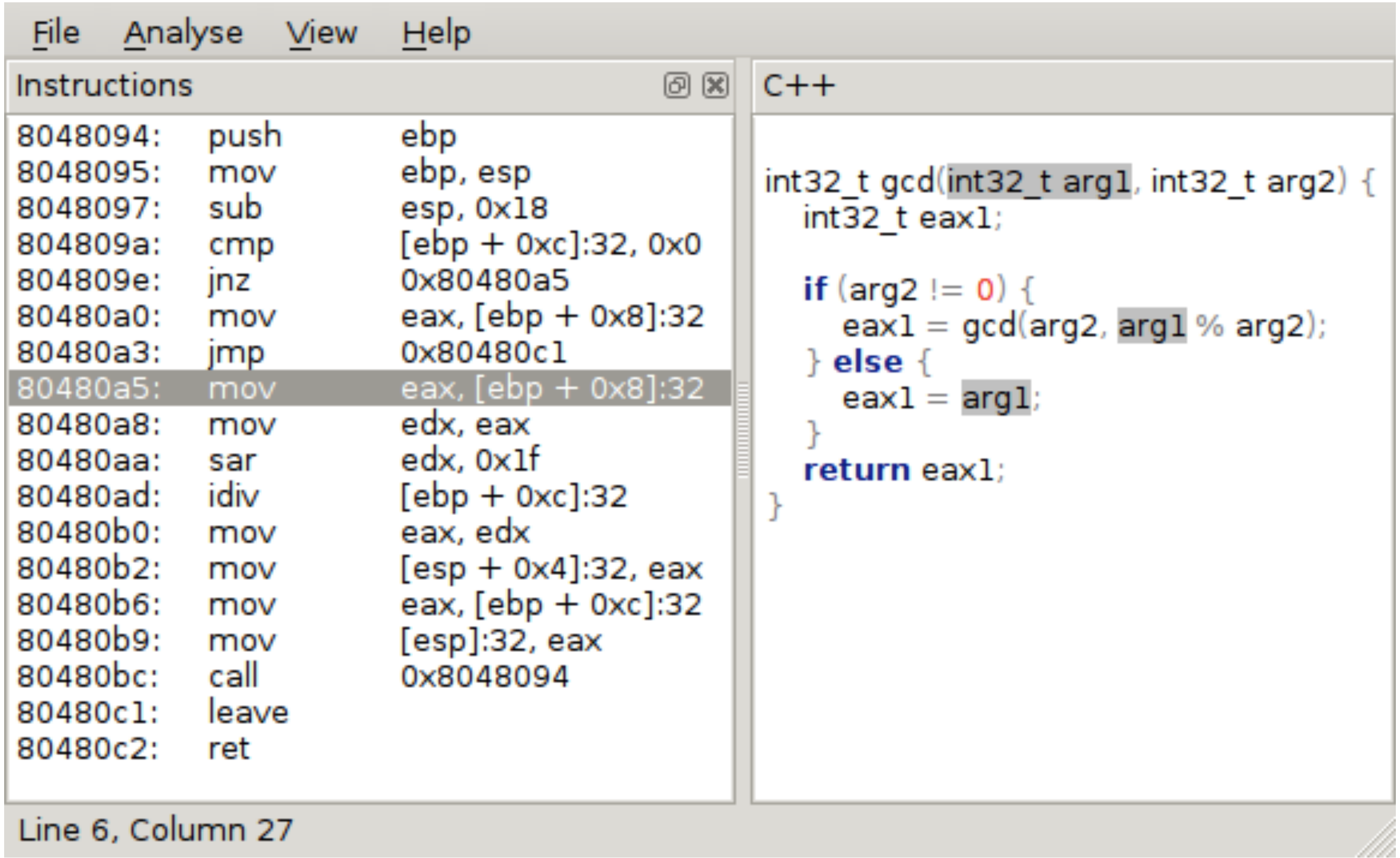


Disassemblers - 2



Disassemblers - 3

- Decompiler



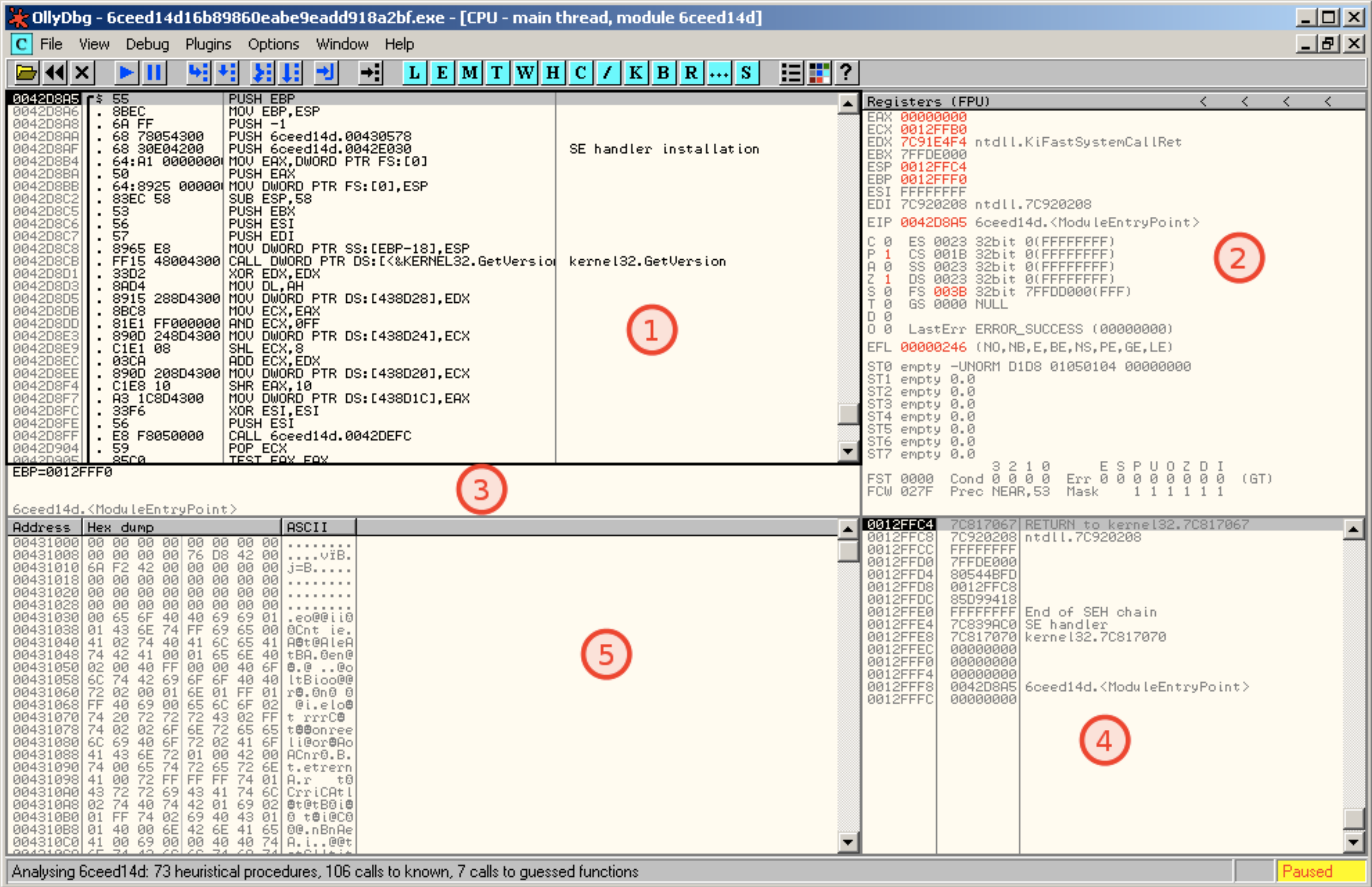
Debuggers - 1

- GDB
- OllyDbg

- Scriptable

- Support tampering
  - alter processor state (incl. program counter)
  - alter memory contents
  - alter code
  - used for out-of-context execution

Debuggers - 1


OllyDbg - 6ceed14d16b89860eabe9eadd918a2bf.exe - [CPU - main thread, module 6ceed14d]
File View Debug Plugins Options Window Help
SE handler installation
kernel32.GetVersion
Registers (FPU)
1
2
3
4
5
EBP=0012FFF0
6ceed14d.<ModuleEntryPoint>
Address Hex dump ASCII
Analysing 6ceed14d: 73 heuristical procedures, 106 calls to known, 7 calls to guessed functions
Paused

Debuggers - 2

- Used for program understanding
- Used for zooming in on relevant code
  - Continuous iterative refinement of scripts

- Low overhead with hardware breakpoints
- High overhead with software breakpoints
  - Requires tampering

Emulation & Instrumentation

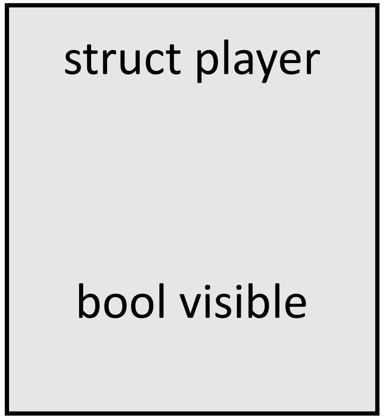
- QEMU
- Pin
- Valgrind
- DynInst
- ltrace

- Used to collect traces
  - To identify patterns and points of interest
- Used like a debugger
  - Iterative refinement of scripts
  - But not interactive

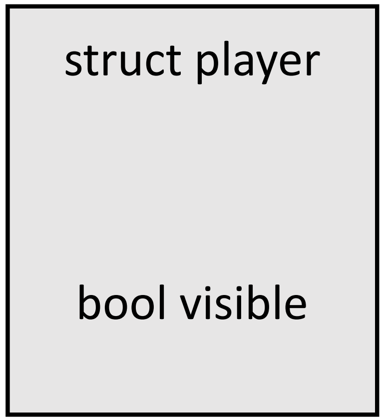
Software Tampering

- Editing the binary
- Alter running process state (CPU, memory)
- Intervene at interfaces
  - system calls
  - library calls
  - network activities
  - ....
- Custom binaries to invoke library APIs
- Aforementioned tools
- Cheat Engine
  - all kinds of reverse engineering aids (pointer chaining)

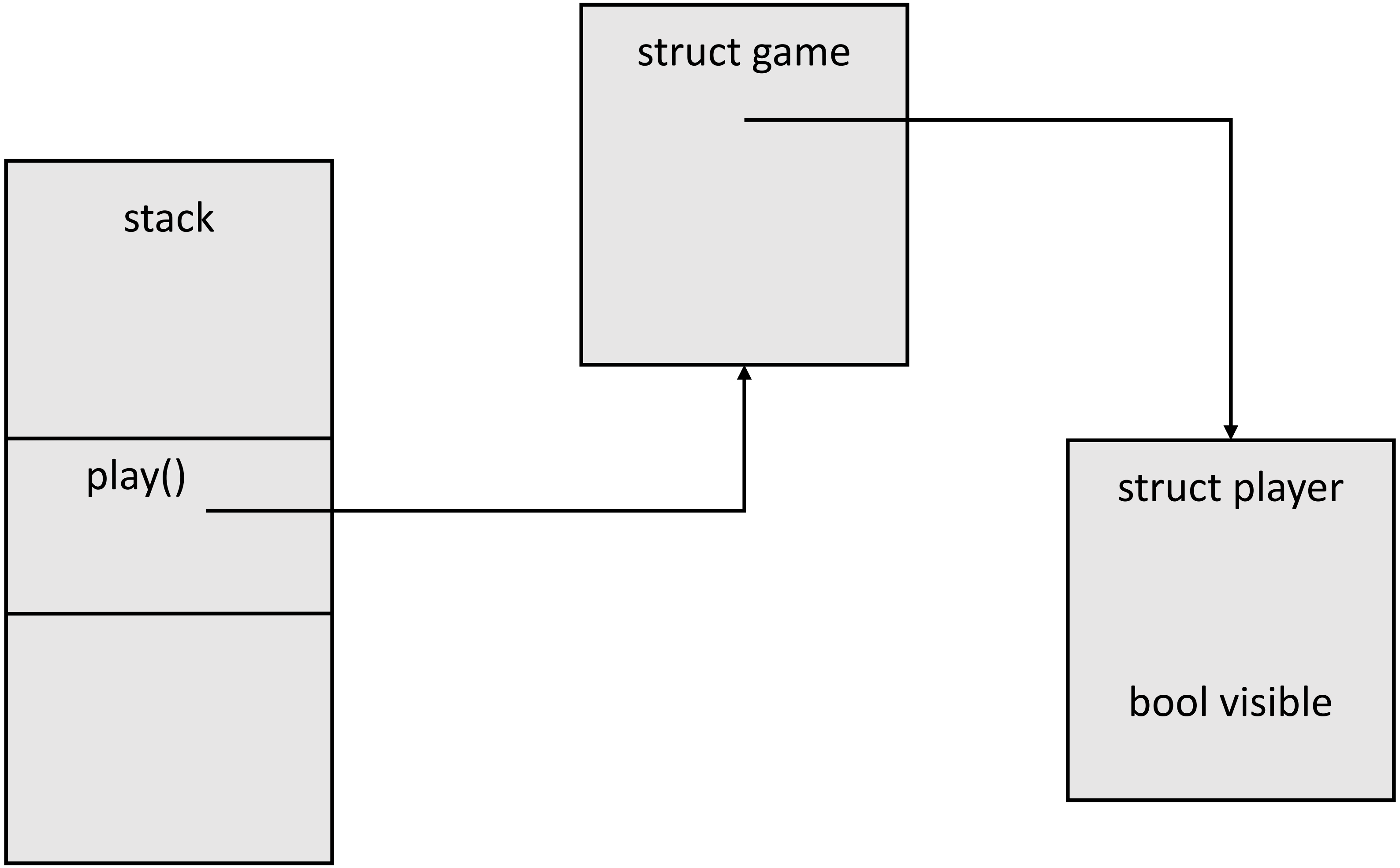
Pointer chaining



Pointer chaining

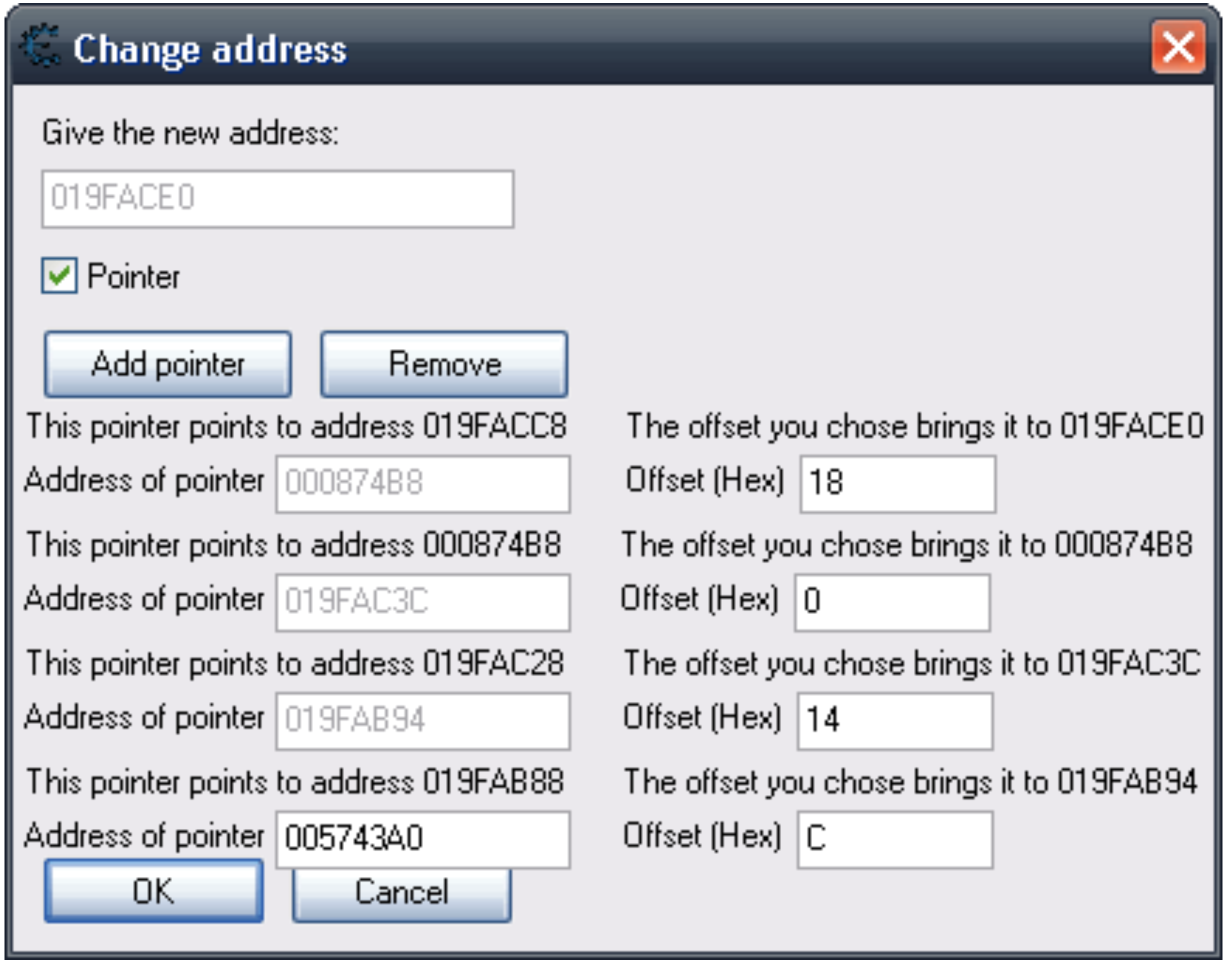


Pointer chaining



((ESP(play())-0x16)+0x4)+0x28

Pointer chaining



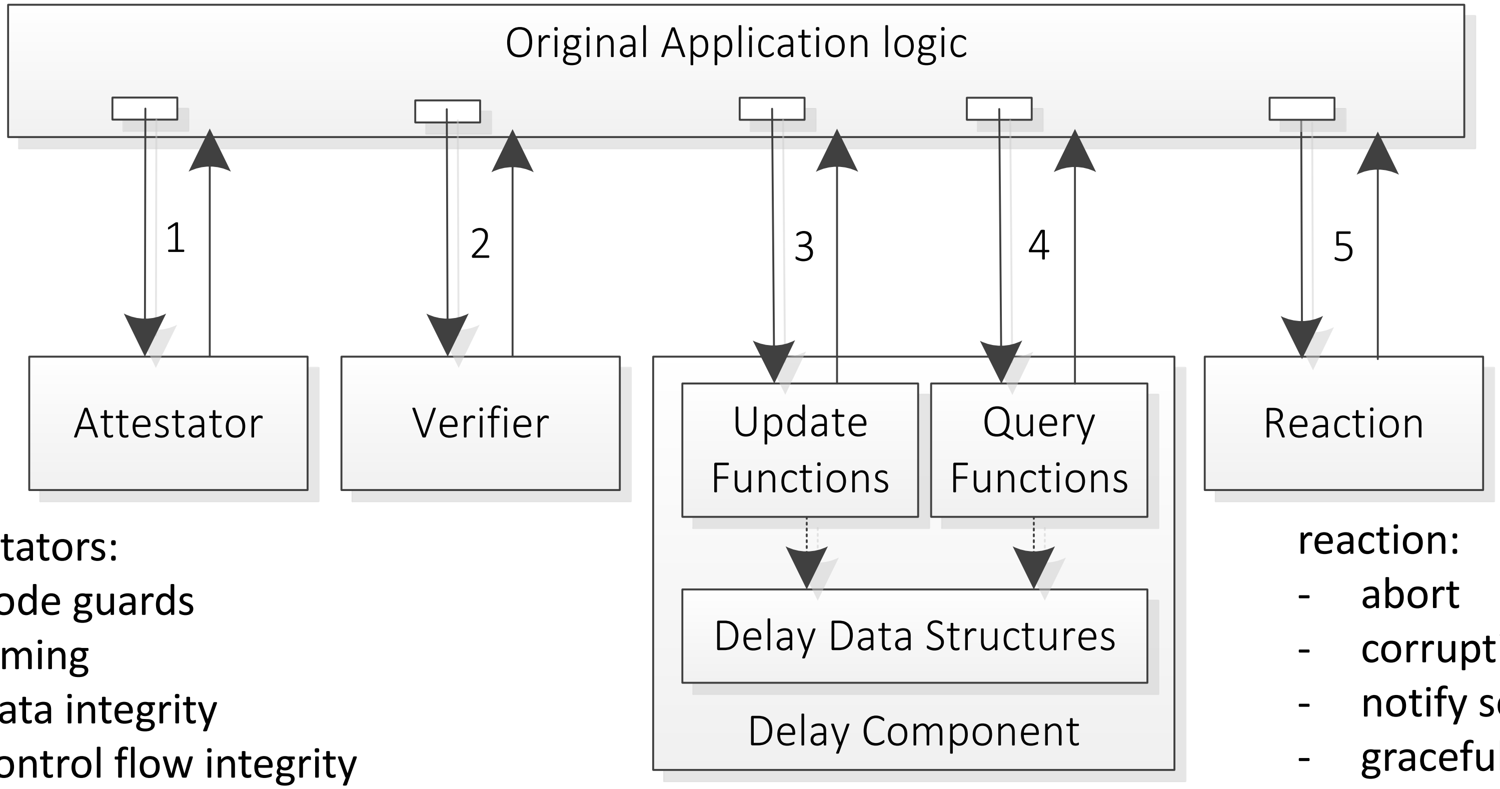
Lecture Overview

1. Basic Attacks
   - attacks on what?
   - basic attack tools & techniques
2. Defenses
   - anti-anything
3. Advanced Automated Attacks
   - generic deobfuscation
   - symbolic execution
4. Defenses
   - anti-even-more

Anti-tampering

- Code guards (code integrity)
  - hashes over code regions
- State inspection
  - check for existing invariants
  - inject additional invariants
  - for data integrity and control flow integrity
- Basic control flow integrity
  - check return addresses
  - check stack frames

Remote attestation



attestators:
- code guards
- timing
- data integrity
- control flow integrity

reaction:
- abort
- corruption
- notify server (block player)
- graceful degradation
- lower quality

verification:
- local vs. remote
- prevent replay attacks

delay reaction:
- attacker sees symptom
- hide relation with cause!

Anti-disassembly

- Hide code
  - packers, virtualization, download code on demand, self-modifying code
- Junk bytes
- Indirect control flow transfers
- Jumps into middle of instructions
- Code layout randomization
- Overlapping instructions
- Exploit known heuristics
  - continuation points
  - patterns for function prologues, epilogues, calls, ...

Often, wrong information is worse than no information.

Anti-disassembly examples

Example 1

```
0x123a:   jmp   0xabca;
          ...
0xabca:   addl  #44,eax
```

obfuscation

```
0x123a:   call       0xabca;
                     ...
0xabca:   pop        ebx;
          addl       #44,eax
```

Example 2

```
0x123a:   call       0xabca;
          ...

0xabca:   ...
          ret
```

obfuscation

```
0x123a:   push       *(0xc000)
          jmp        0xabca
          pop        eax
          ...

0xabca:   ...
          jmp *(esp)

0xc000:   0x12424
```

Anti-decompilation

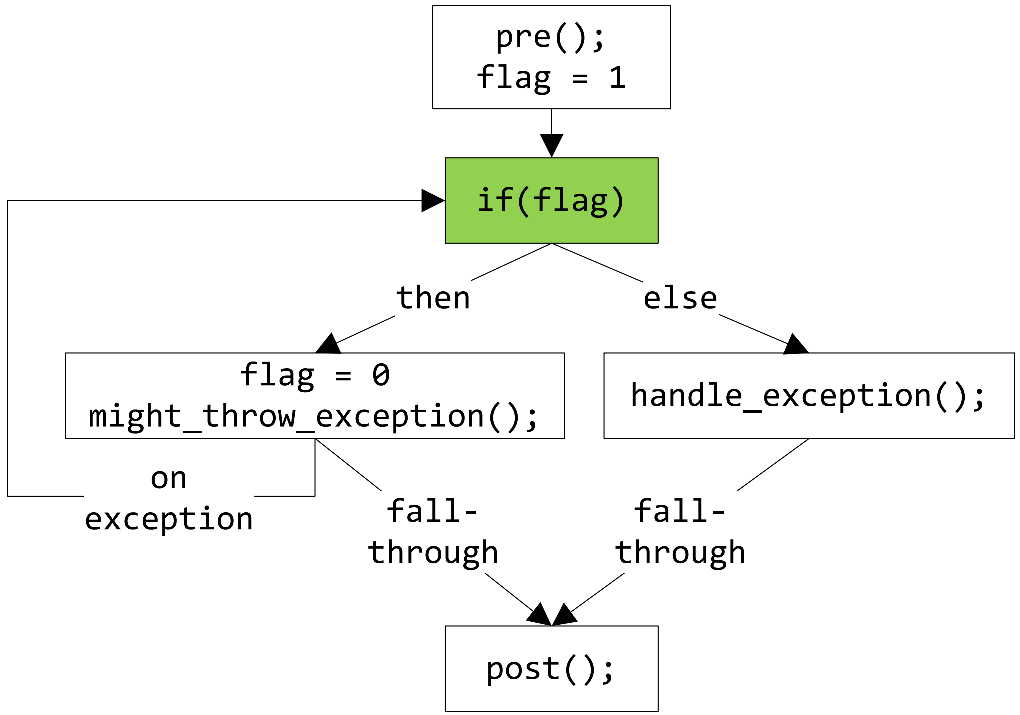
Exploit semantic gap between source code and assembly code or bytecode

- strip unnecessary symbol information

- rename identifiers (I,l,L,1)
- goto spaghetti
- disobey constructor conventions
- disobey exception handling conventions

Anti-decompilation example

```
pre();
try{
  might_throw_exception();
}
catch(Exception e){
  handle_exception();
}
post();
```



Batchelder, Michael, and Laurie Hendren. "Obfuscating Java: the most pain for the least gain." In *Compiler Construction*, pp. 96-110. Springer Berlin Heidelberg, 2007

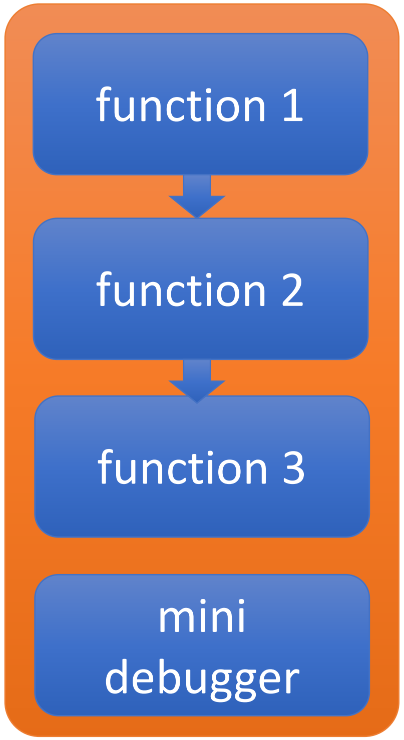
Anti-debugging

- Option 1: check environment for presence debugger

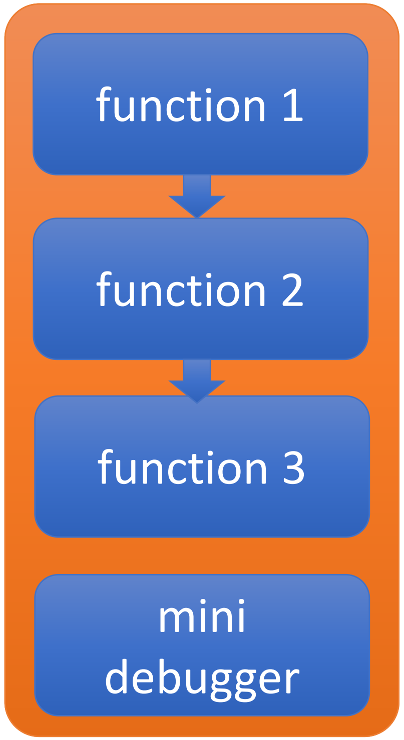
- Option 2: prevent debugger to attach
  - OS & hardware support at most one debugger per process
  - occupy one seat with custom “debugger” process
  - make control & data flow dependent on custom debugger

  - anti-debugging by means of self-debugging

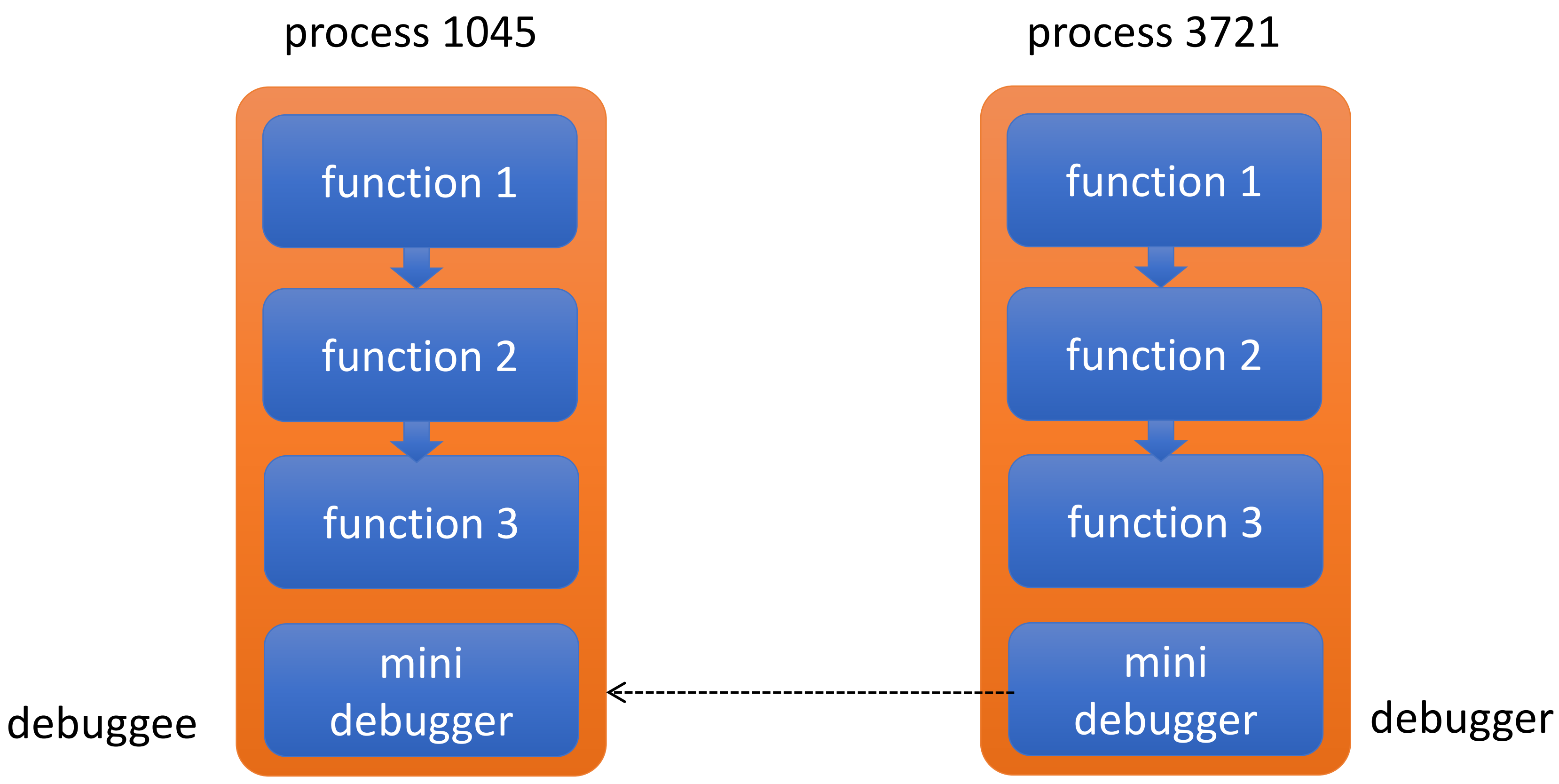
Self-Debugging



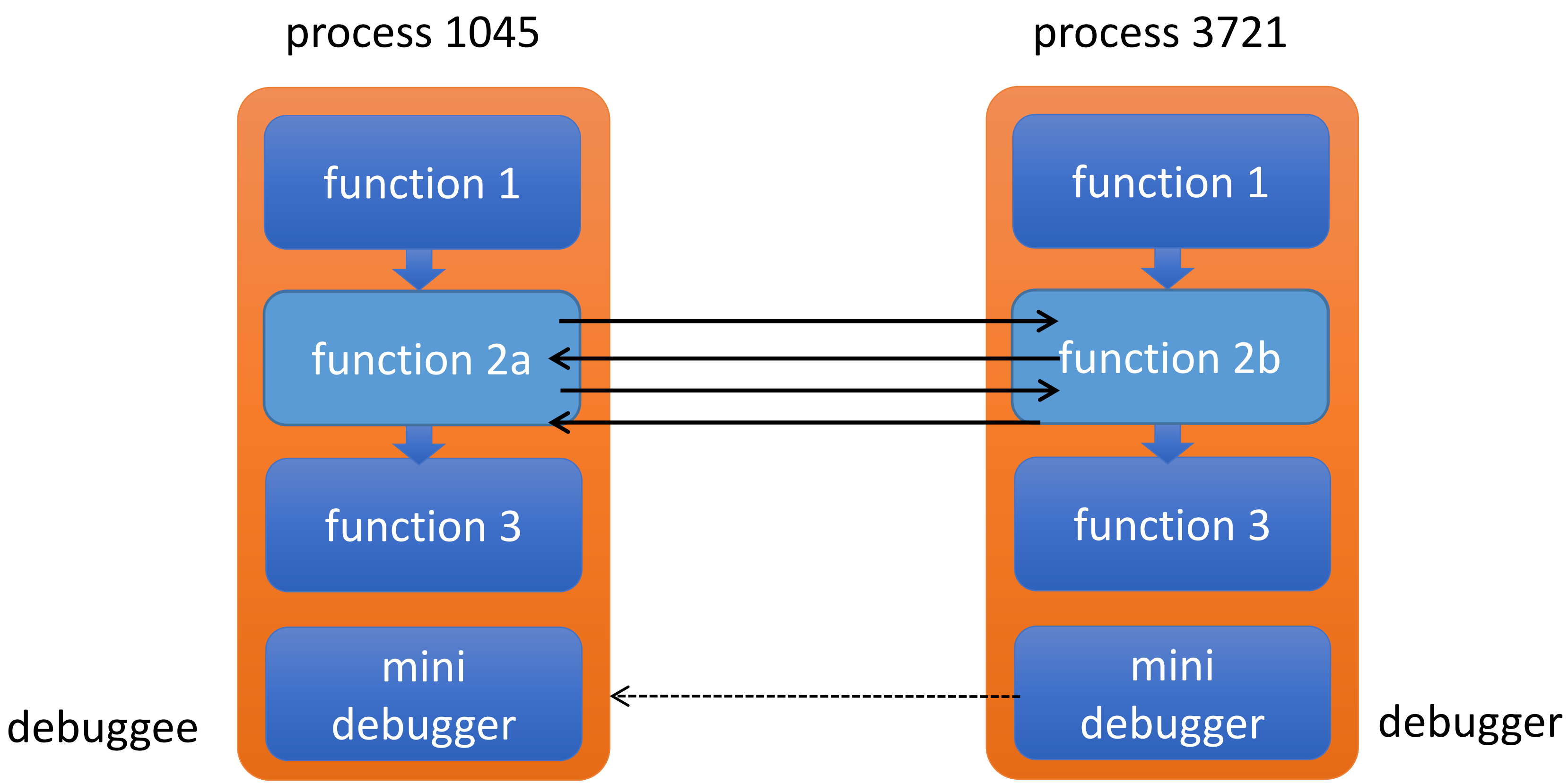
Self-Debugging



Self-Debugging



Self-Debugging



Anti-emulation

- Emulators are buggy – incomplete
- Virtual environments are not real

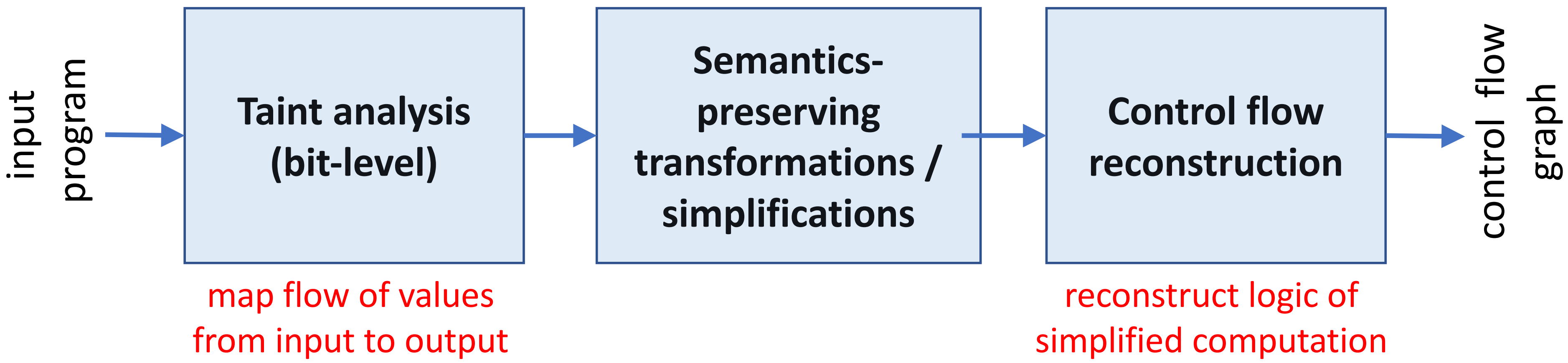
- Johanna Rutkowska
  - Blue pill
  - Red pill

Lecture Overview

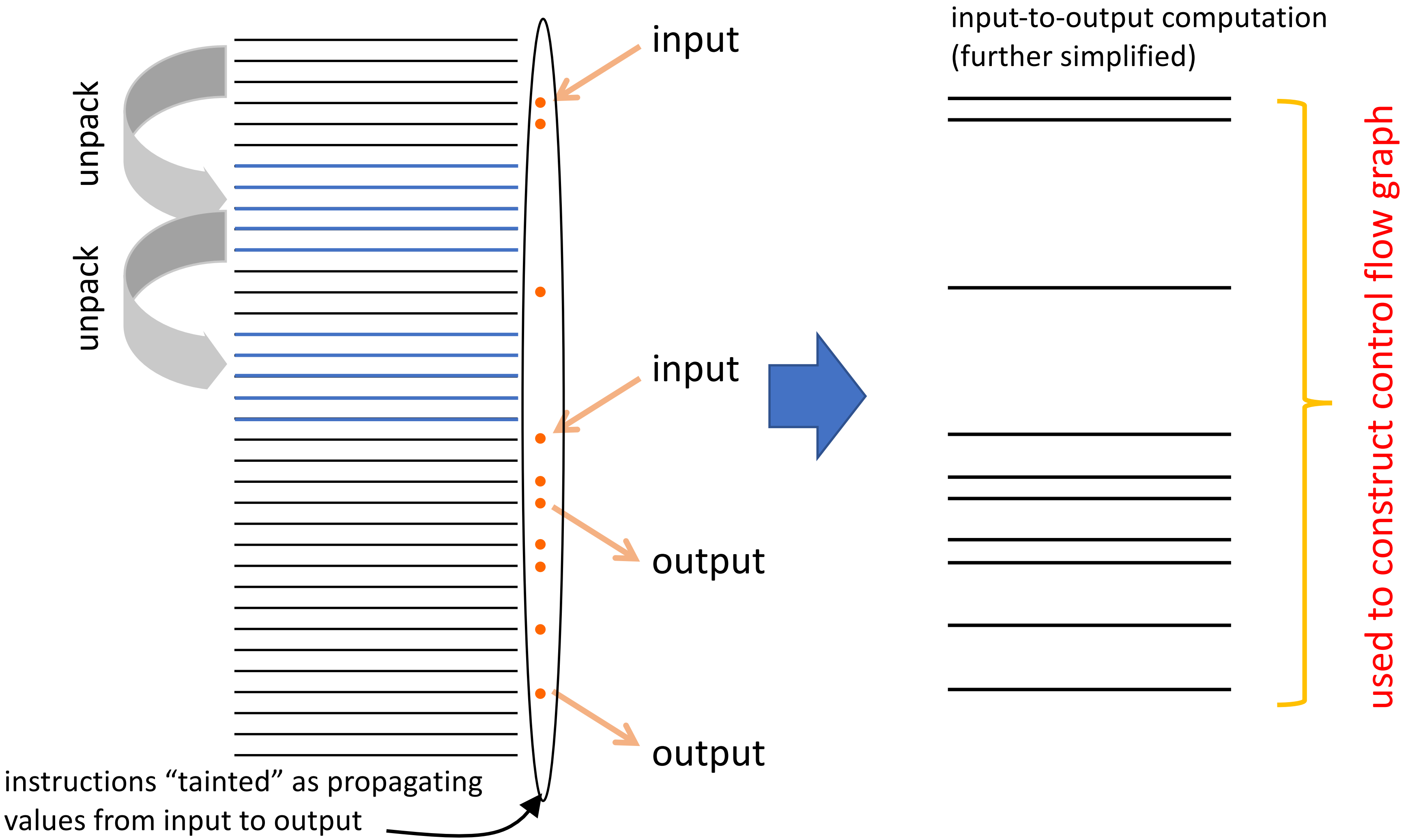
1. Basic Attacks
   - attacks on what?
   - basic attack tools & techniques
2. Defenses
   - anti-anything
3. Advanced Automated Attacks
   - generic deobfuscation
   - symbolic execution
4. Defenses
   - anti-even-more

Generic Deobfuscation (Yadegari et al IEEE S&P 2015)

- no obfuscation-specific assumptions
  - treat programs as input-to-output transformations
  - use semantics-preserving transformations to simplify execution traces
- dynamic analysis to handle runtime unpacking



Generic Deobfuscation (Yadegari et al IEEE S&P 2015)

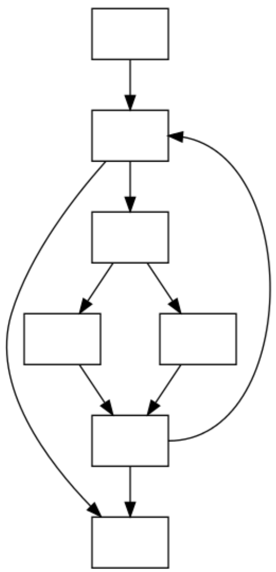


Generic Deobfuscation (Yadegari et al IEEE S&P 2015)

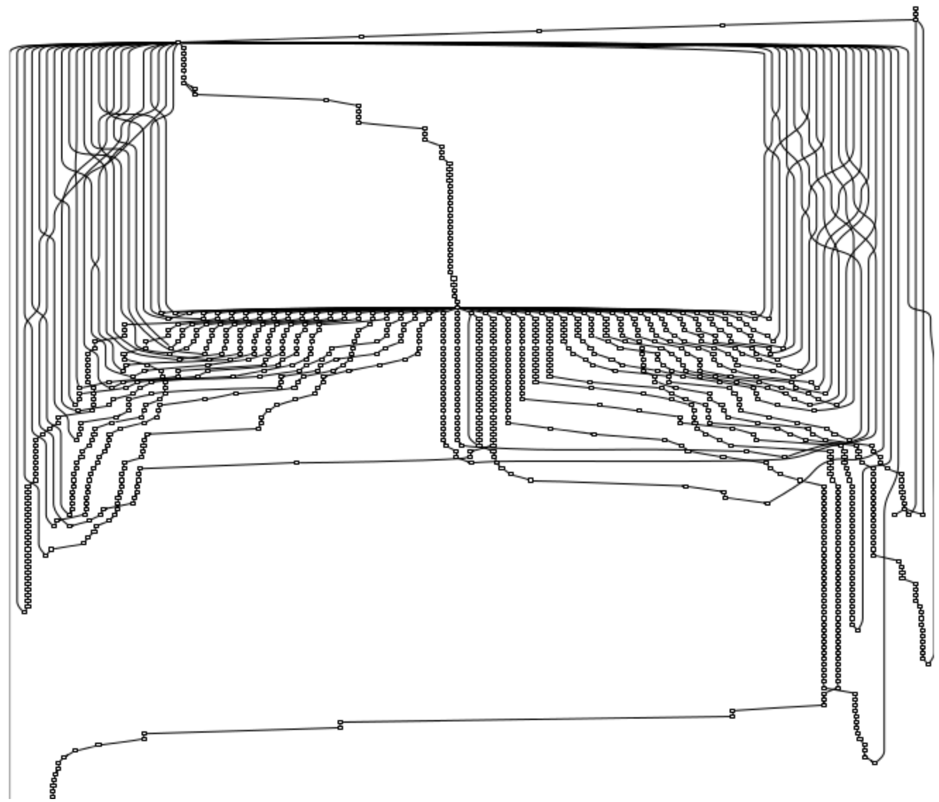
- *Quasi-invariant locations*: locations that have the same value at each use.
- Their transformations:
  - Arithmetic simplification
    - adaptation of *constant folding* to execution traces
    - consider quasi-invariant locations as constants
    - controlled to avoid over-simplification
  - Control simplification
    - E.g., convert indirect jump through a quasi-invariant location into a direct jump
  - Data movement simplification
    - use pattern-driven rules to identify and simplify data movement.
  - Dead code elimination
    - need to consider implicit destinations, e.g., condition code flags.

Generic Deobfuscation (Yadegari et al IEEE S&P 2015)

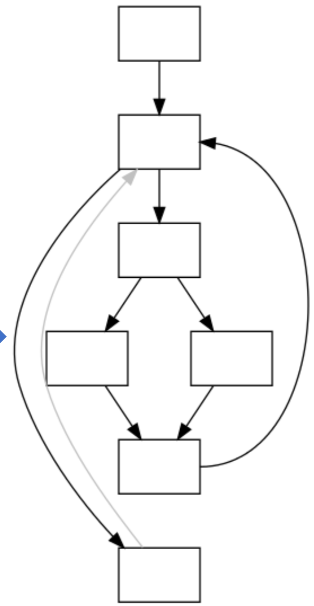
original



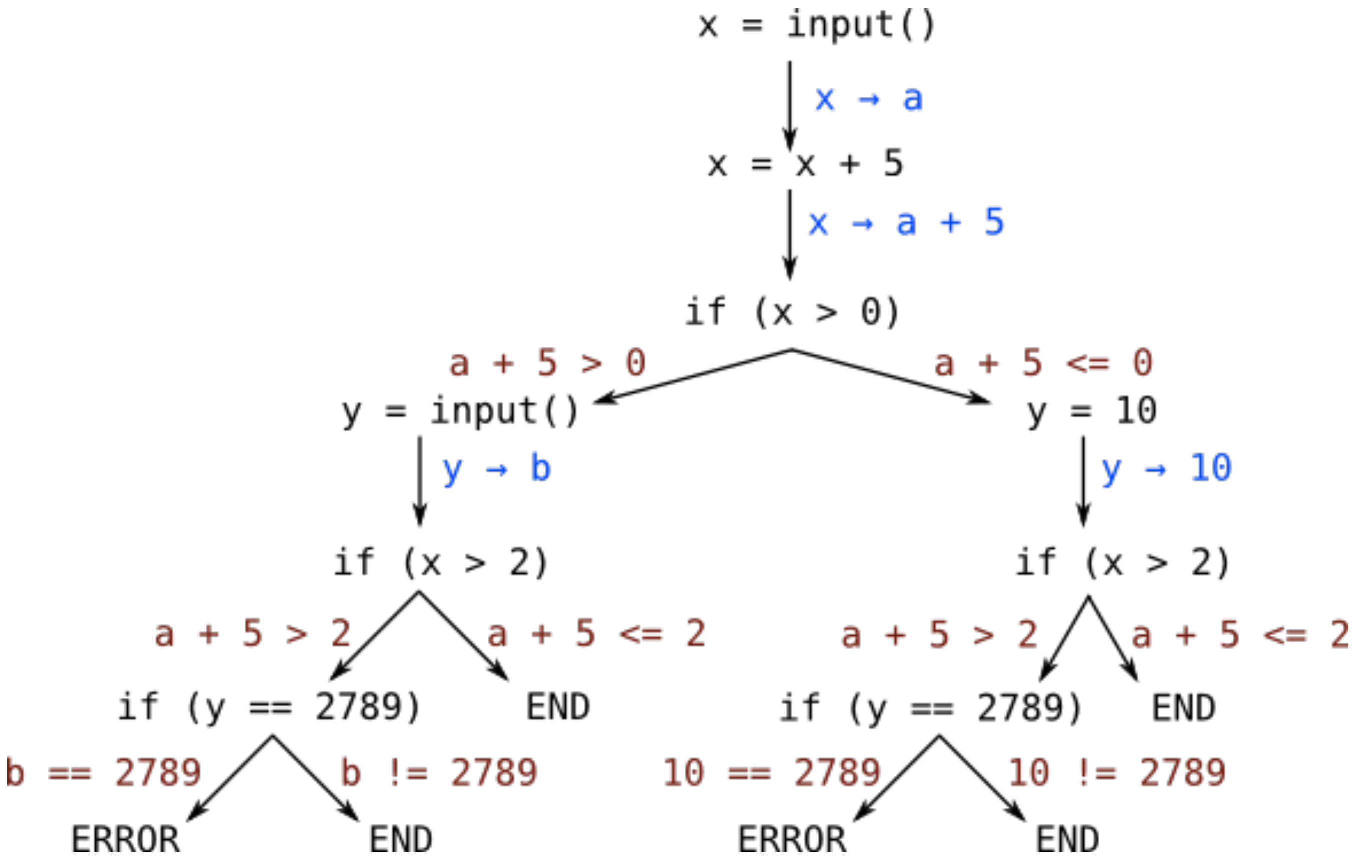
obfuscated with Themida (cropped)



deobfuscated



Symbolic Execution



effective because most obfuscations implement semantics that do not involve input

Lecture Overview

1. Basic Attacks
    - attacks on what?
    - basic attack tools & techniques
2. Defenses
    - anti-anything
3. Advanced Automated Attacks
    - generic deobfuscation
    - symbolic execution
4. Defenses
    - anti-even-more

Anti-taint analysis

- tainting all data with artificial computations

- hiding data dependencies through covert channels
  - time
  - system state
  - anything not normally checked by analysis

Obfuscations with varying, input-dependent behavior (Banescu et al, ACSAC 2016)

```c
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i = 0; i < strlen(str); str++, i++) {
  hash = (hash << 7) ^ (*str);
}
if (hash == 809267) printf("win\n");
```

1. RANGE DIVIDER

```c
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i = 0; i < strlen(str); str++, i++) {
  char chr = *str;
  if (chr > 42) {
    hash = (hash << 7) ^ chr;
  } else {
    hash = (hash * 128) ^ chr;
  }
}
if (hash == 809267) printf("win\n");
```

Obfuscations with varying, input-dependent behavior (Banescu et al, ACSAC 2016)

2. INPUT INVARIANTS

1. inject extra inputs into programs
2. let correct execution depend on invariant properties of those inputs

for example: feed program extra key to decrypt bytecode

(how to get these to the user ???)