Evaluating the strength of software protections

Bjorn De Sutter
Ghent University

ISSISP
30 July 2014

Verona, Italy

About me

- Research domain: system software
  - compilers, binary rewriting tools, whole program optimization (binary & Java), virtualization
  - improve programmer productivity
  - apply tools for different applications
  - obfuscation, diversity and mitigating side channels

UNIVERSITEIT GENT

- Also worked at

- Interrupts enabled, but not all handlers might be installed

First: What do we want to achieve with the protection and the evaluation?

Evaluating the strength of software protection

- Four criteria (Collberg et al)
  - **Potency**: confusion, complexity, manual effort
    - of what? what task? how computed? by who?
  - **Resilience**: resistance against (automated) tools
    - existing and non-existing? operated by who? to achieve what?
  - **Cost**: performance, code size
  - **Stealth**: identification of (components of) protections

Overview

- ASPIRE in a nutshell
- Modelling attacks
- Evaluation Criteria
  - Metrics of complexity
  - Resilience
- Theory versus practice: involving the humans

Aspire in a nutshell

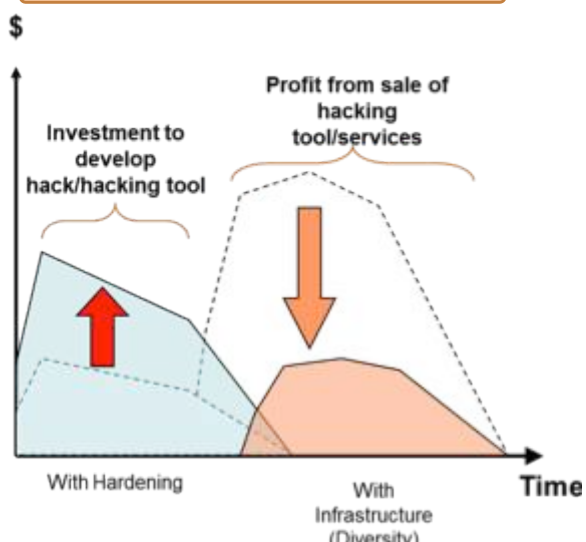
NAGRA · SafeNet · gemalto

Data Hiding → Algorithm Hiding → Anti-Tampering → Remote Attestation → Renewability

- SafeNet use case → **ASPIRE Framework** → Protected SafeNet use case
- Gemalto use case → **ASPIRE Framework** → Protected Gemalto use case
- Nagravision use case → **ASPIRE Framework** → Protected Nagravision use case

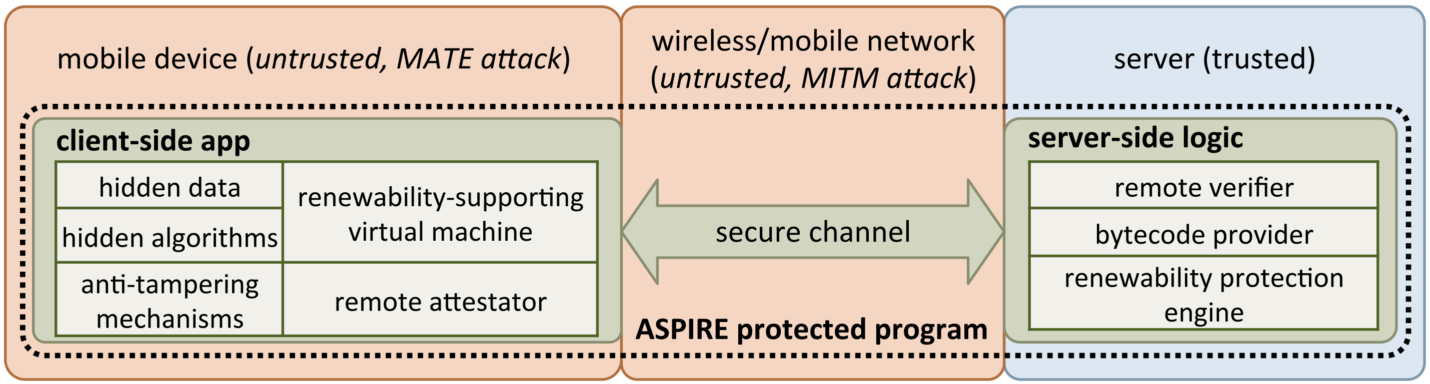
ASPIRE Framework

- **Decision Support System**
- ⇕
- **Software Protection Tool Chain**

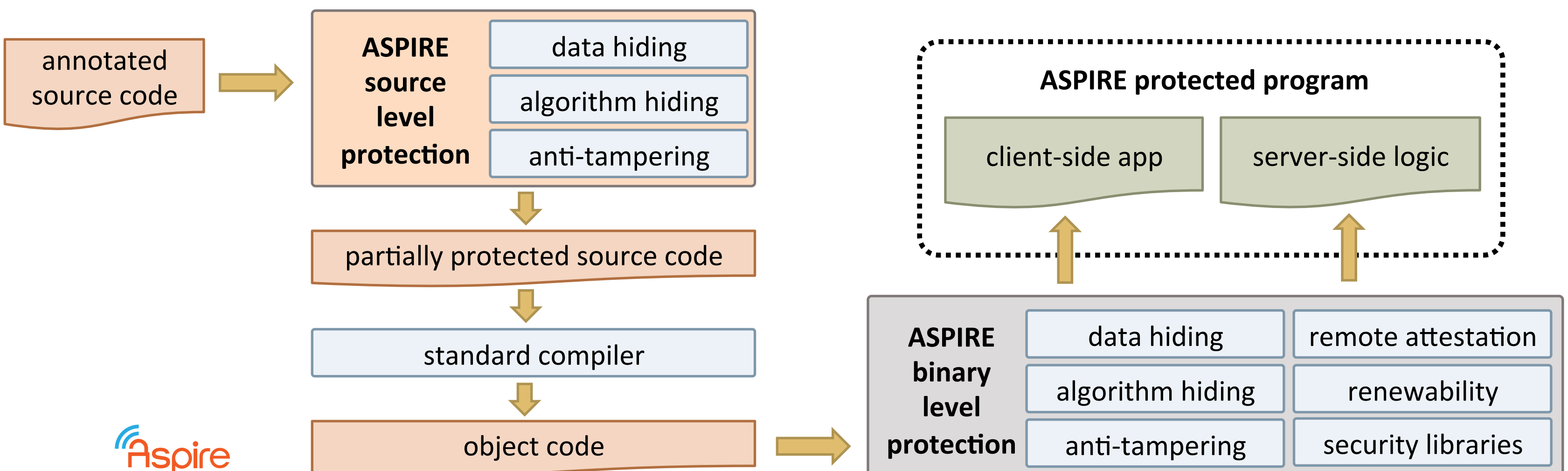


Aspire Goals

1. Protected mobile services

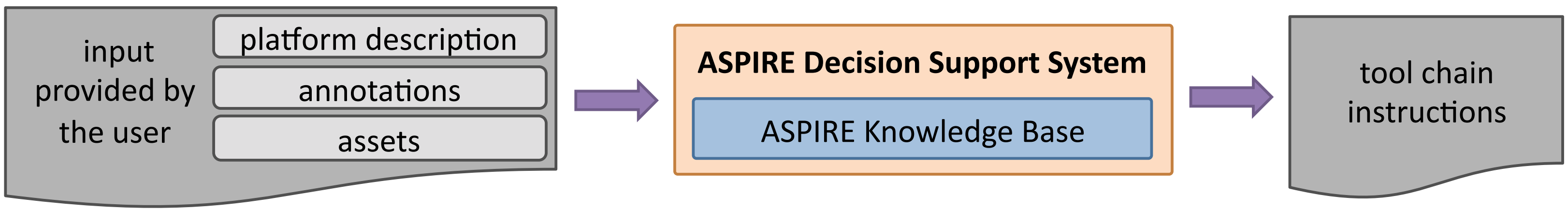


2. Software protection techniques and integrated tool flow



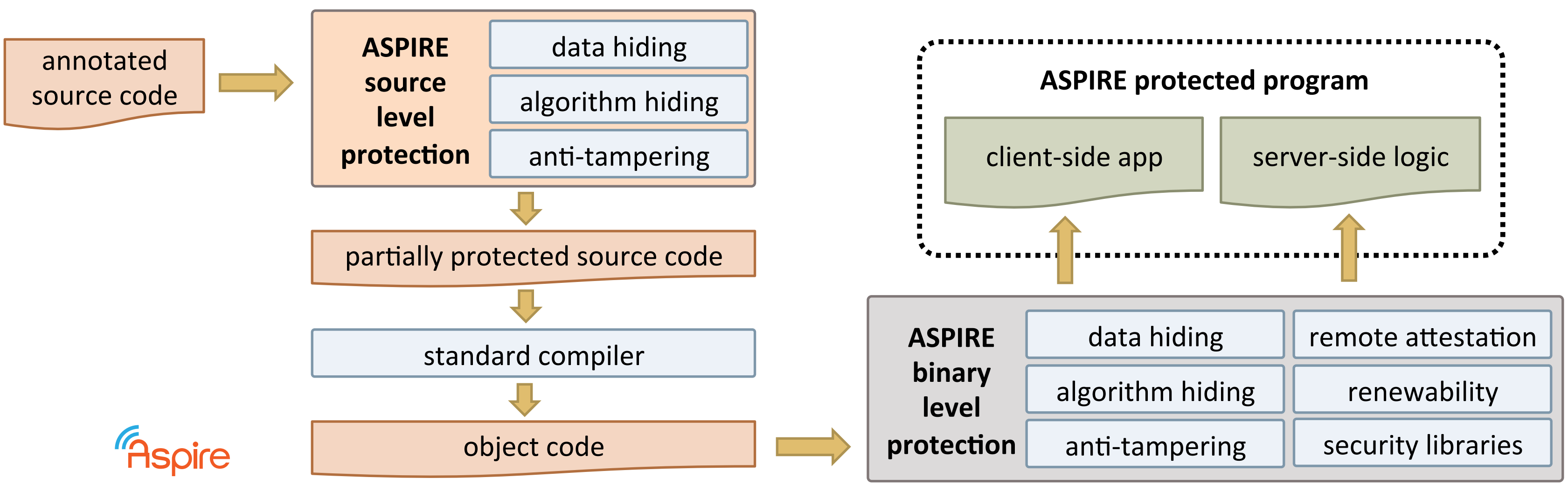
Aspire Goals

3. Decision Support System



- attack models & evaluation methodology
- security metrics
- experiments on human subjects (students + researchers)
- public challenge

2. Software protection techniques and integrated tool flow

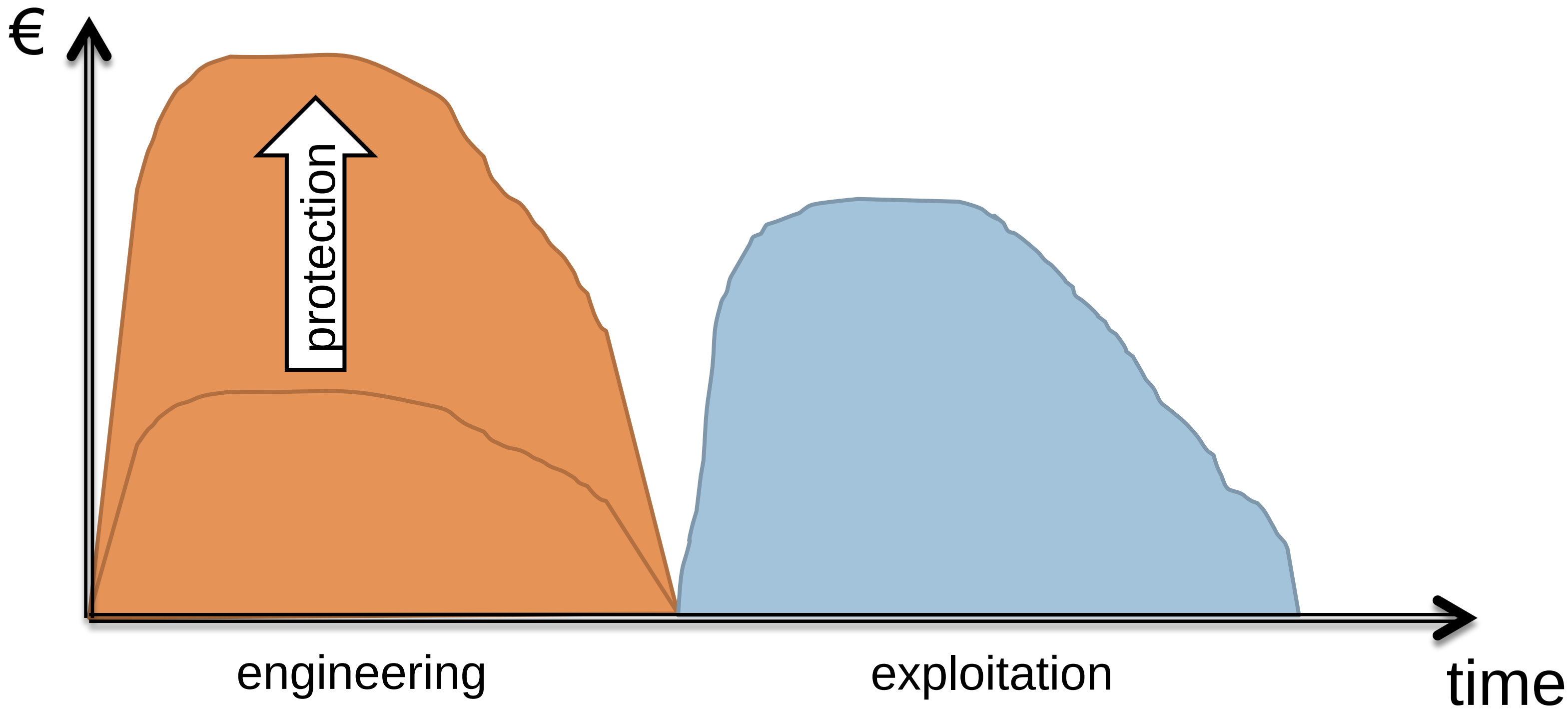


Protection againts MATE attacks

software analysis tools
FPGA sampler
oscilloscope

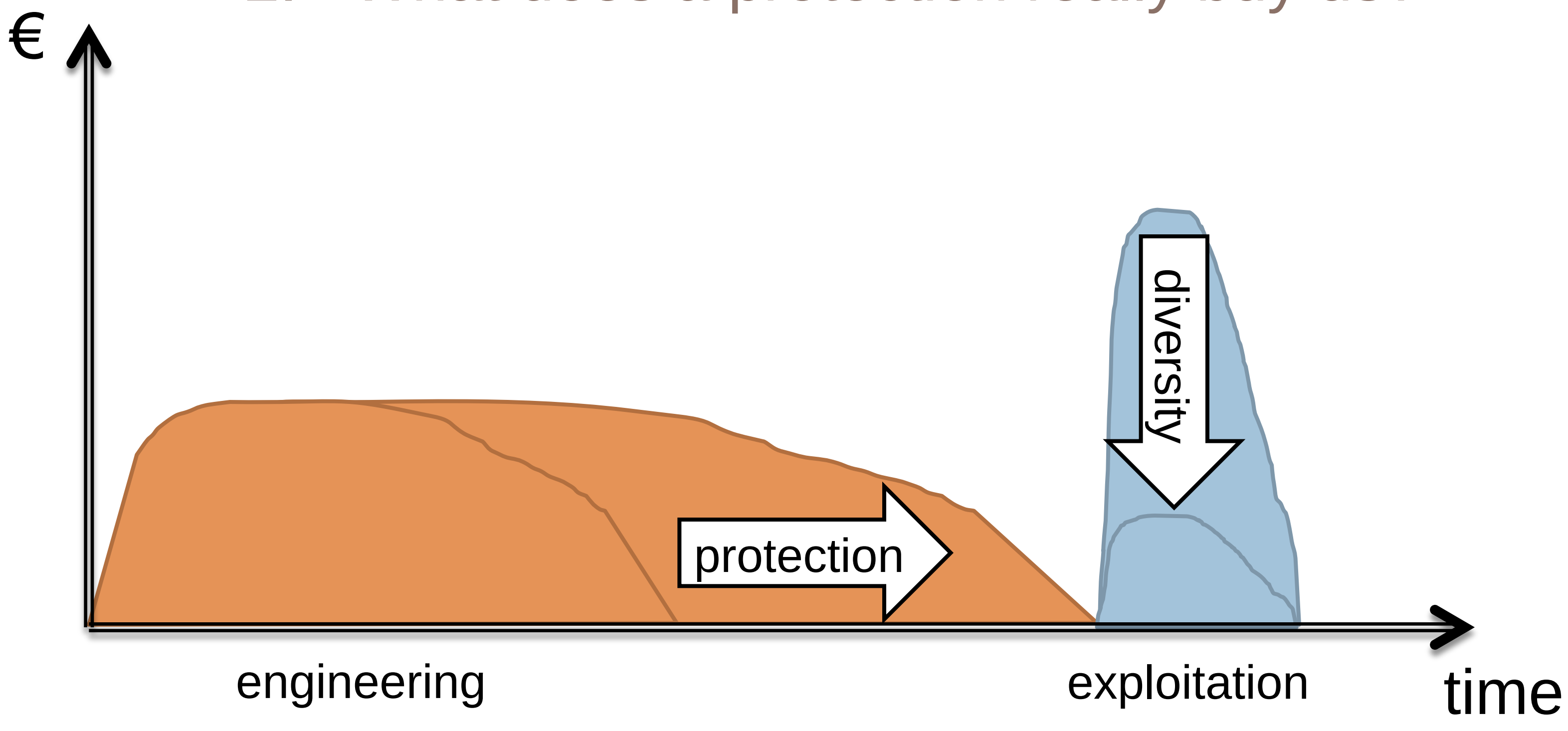
developer boards
screwdriver
JTAG debugger

Economics of MATE attacks



Economics of MATE attacks

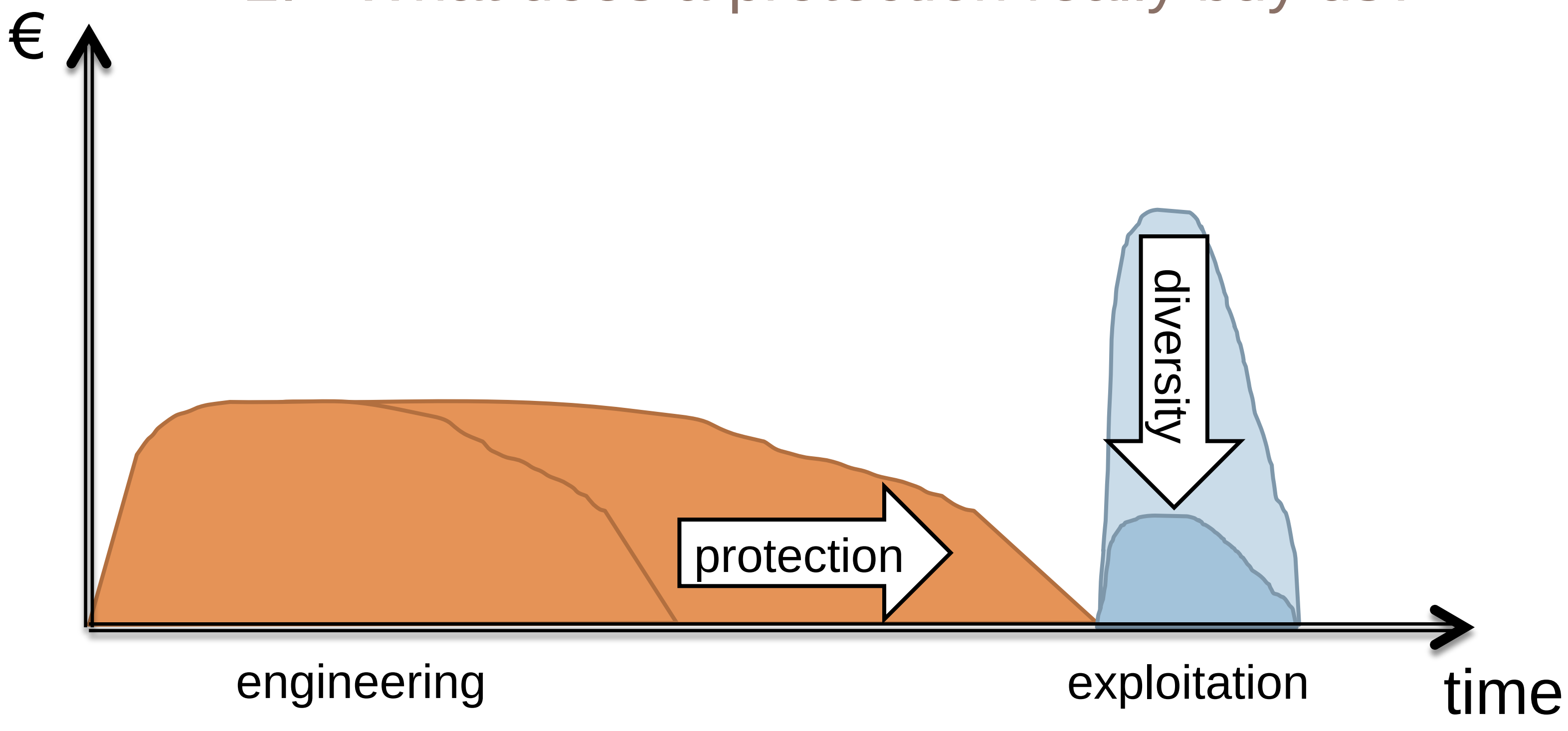
1. What is the most appropriate protection?
2. What does a protection really buy us?



Economics of MATE attacks

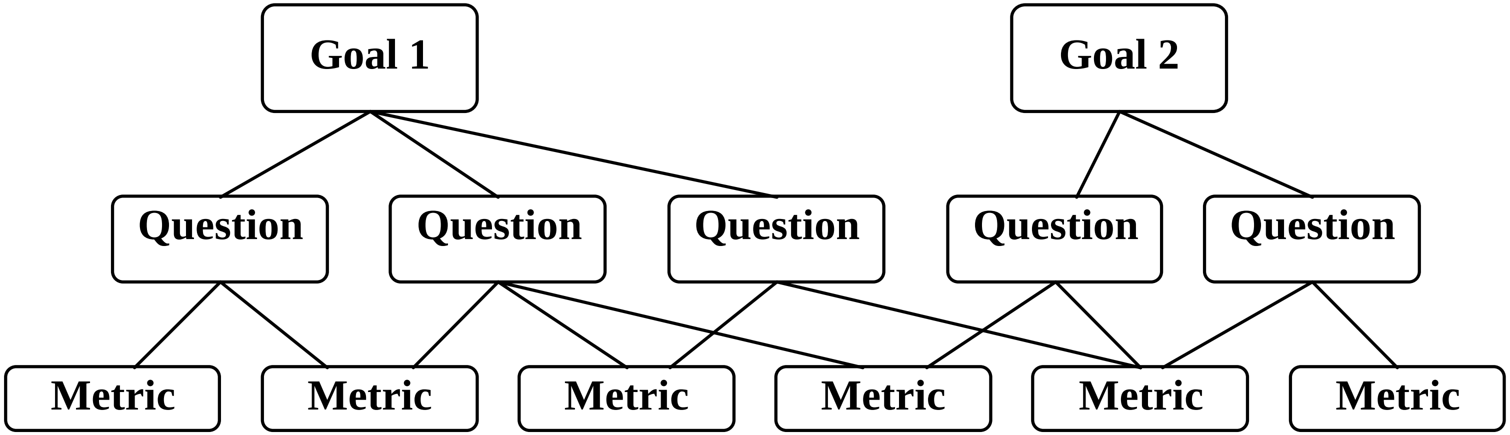
Economics of MATE attacks

1. What is the most appropriate protection?
2. What does a protection really buy us?



The Goal – Questions – Metrics Approach (Basili et al)

- **Goal**: What am I trying to achieve?
- **Questions**: What matters for achieving that?
- **Metrics**: How do we evaluate that?



The Goal – Questions – Metrics Approach

ASPIRE project level

- **Goal**:
  - Optimize protection process
- **Questions**:
  - Which assets, attack steps, tools, protections, ...
  - What is their potency, resilience, cost, value, ...
- **Metrics**:
  - Measurable features of attacks, of protections and of (un)protected software

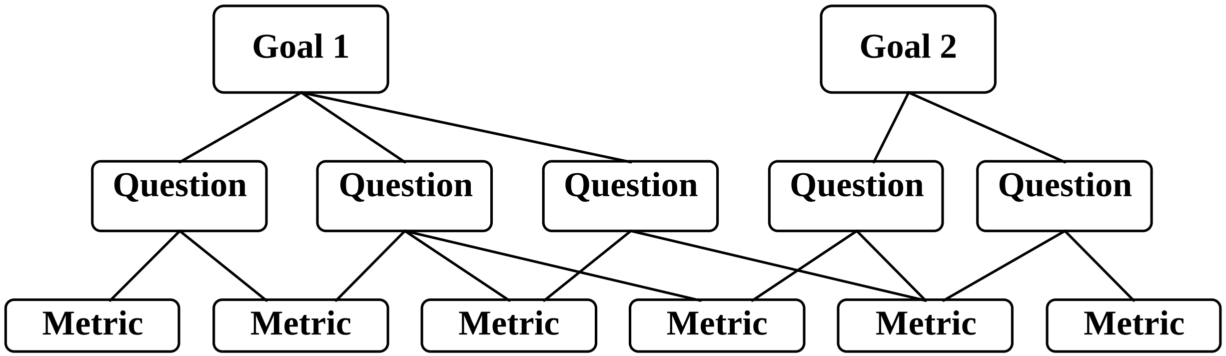
The Goal – Questions – Metrics Approach

Your Individual Protection

- **Goal**:
  - Protect specific software assets against specific attack(s)
- **Questions**:
  - What determines effort, what is delta in effort?
- **Metrics**:
  - Measurable features of attack steps and of (un)protected software

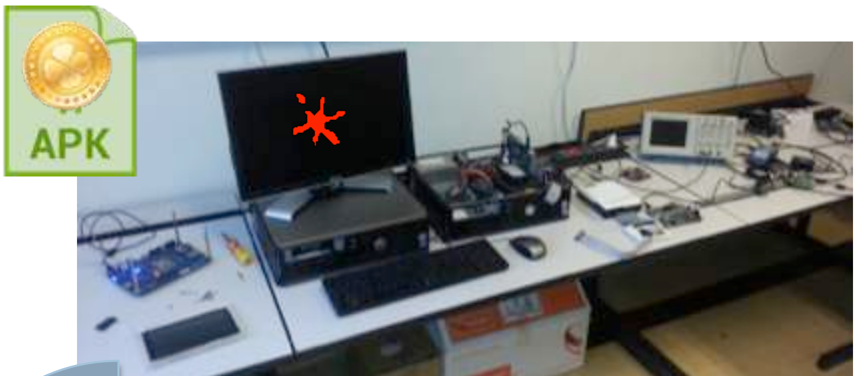
The Goal – Questions – Metrics Approach

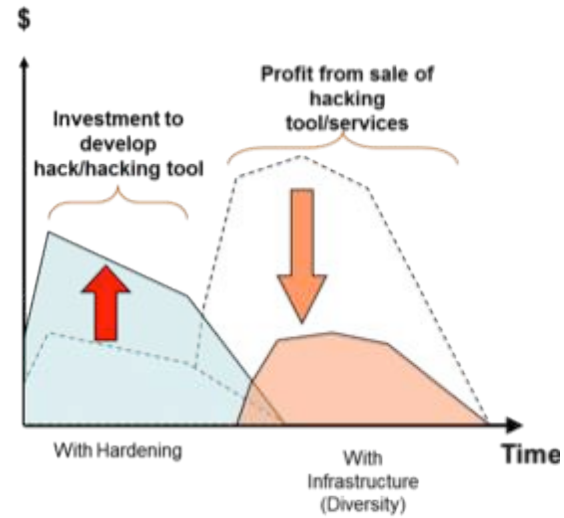
- Advance warning



- established metrics were designed for other goals!
  - maintainability, testability, reliability, ...
- custom metrics are very specific
- specific vs generic goals?

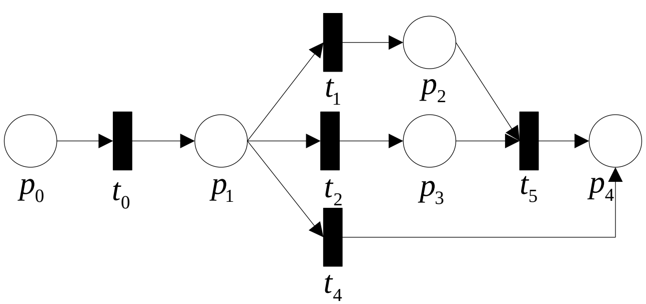
What is the most appropriate protection?

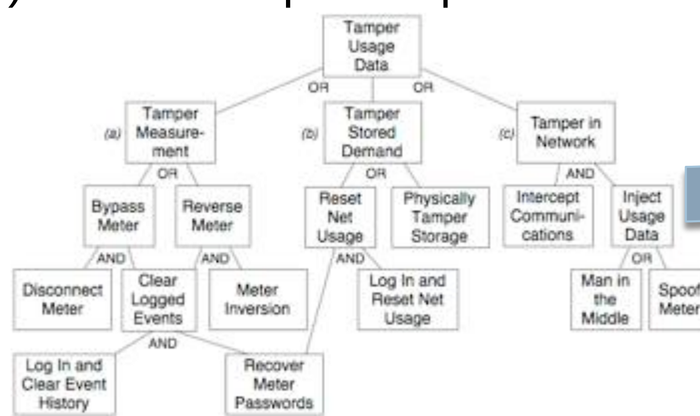




1) model the attack paths

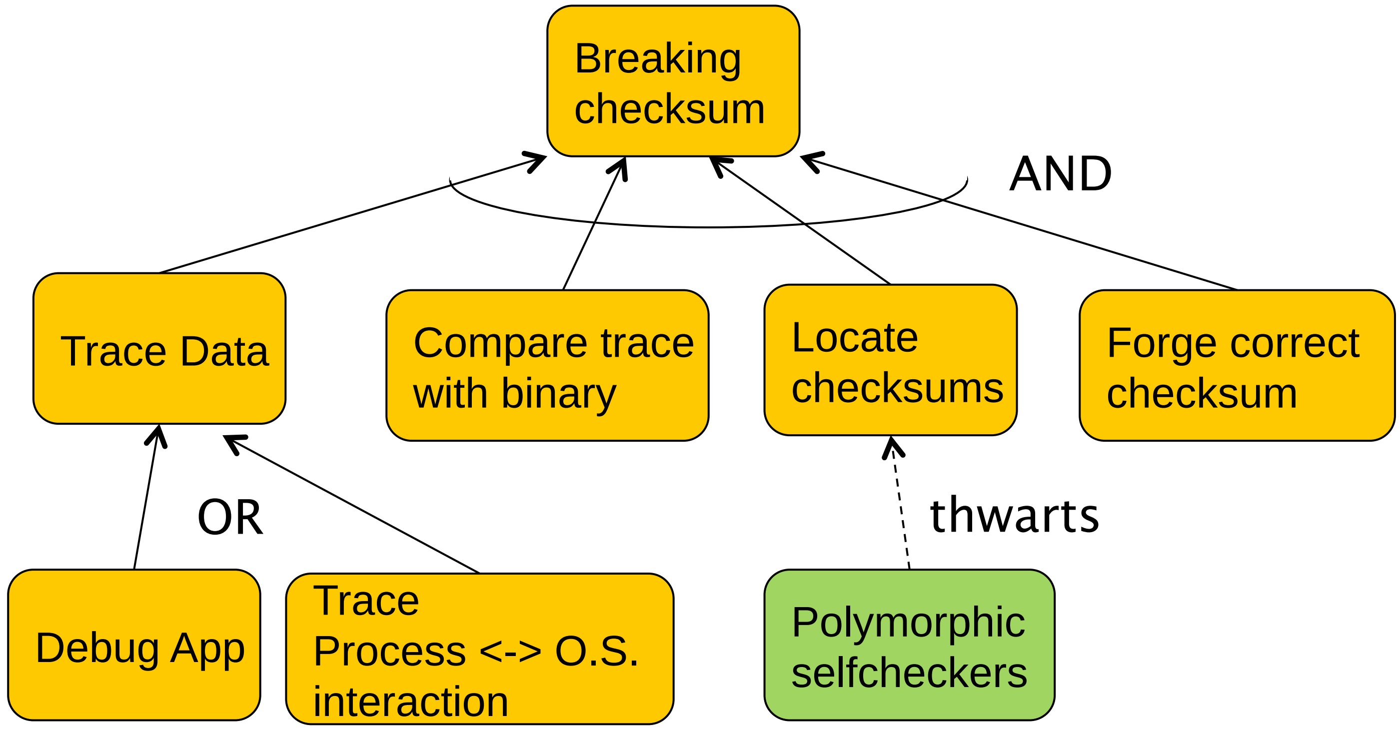
2) evaluate impact of protections





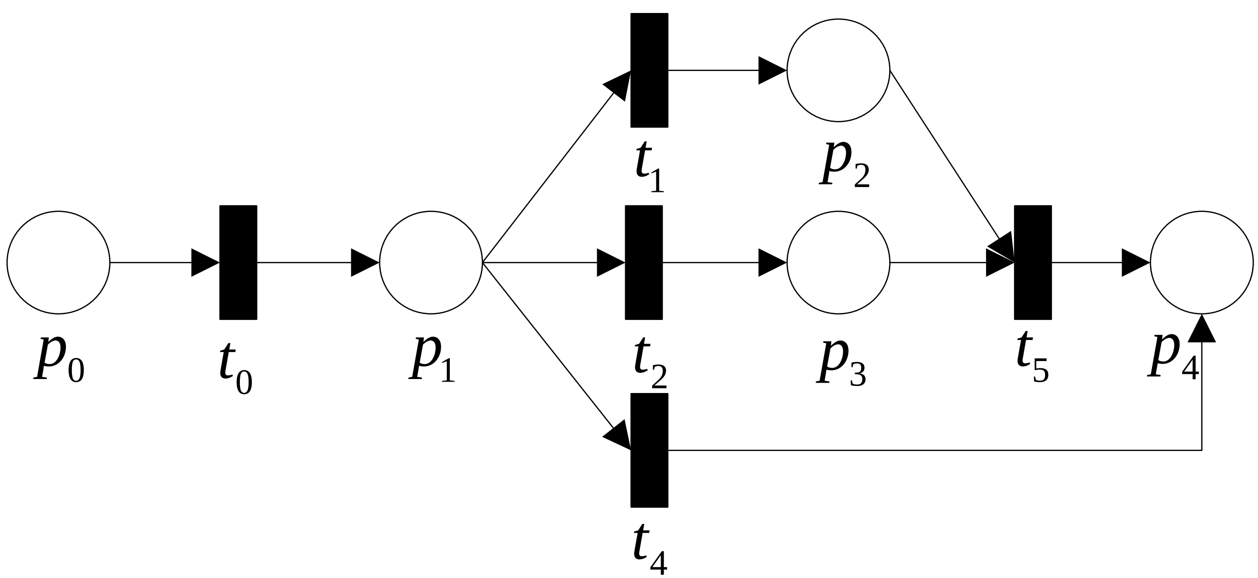
Attack Modelling: Attack Graphs (AND-OR Graphs)

- relate attack goal, subgoals, (and protections)



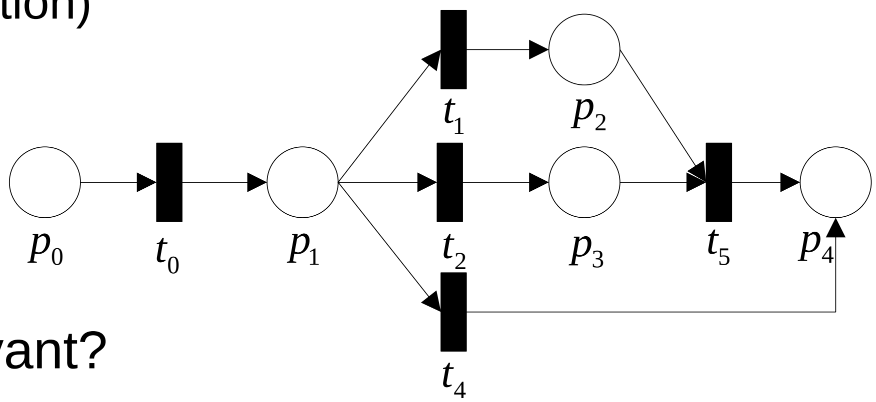
Attack Modelling: Petri Nets (Wang et al, 2012)

- Model attack paths
  - places are reached subgoals (with properties)
  - transitions are attack steps
  - can model AND-OR
  - can be simulated for protected and unprotected applications



Attack Modelling: Petri Nets

- What is outcome of transition?
  - Identification of feature or asset?
  - Simplified program (representation)
  - Tampered program
  - Reduced search space
  - Analysis result



- What determines effort?
- What code fragments are relevant?
- Generic attack steps vs. concrete attack steps?
- How to aggregate information?
  - Effort
  - Probability of success
- How to build the Petri Net? (backward reasoning & knowledge base)

Backward Reasoning

- Start from
  - assets & threats
  - application features (attack paths)
  - code features (protections, effort)
  - knowledge base on
    - attack steps
      - methods
      - tools & techniques
      - preconditions, postconditions
    - attack paths

Assets & Threats (B. Wyseur)

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

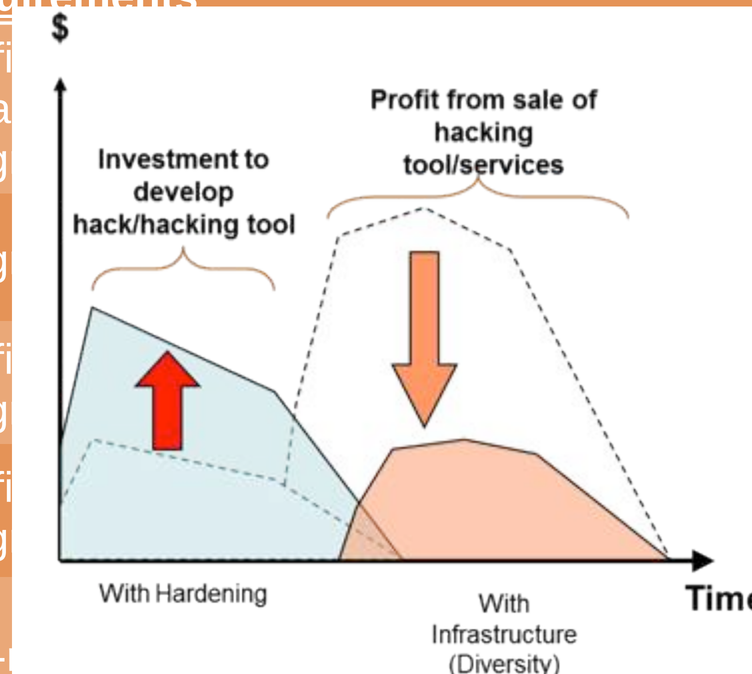
Attack Attributes (B. Wyseur)

- **Identification**: quantifies the effort to break an application once
- **Exploitation:** expresses the possibility that the attack can be repeated and scaled

	Identification				Exploitation		
	Guru	Expert	Geek	Amateur	High	Medium	Low
Attack 1	Insecure			Secure			
Attack 2							

Assets,Threats, and Attacks

Asset category	Security Requirements	Examples of threats
Private data (keys, credentials, tokens, private info)	Confi… Priva… Integ…	…ate authorization
Public data (keys, service info)	Integ…	
Unique data (tokens, keys, used IDs)	Confi… Integ…	…gitimate access
Global data (crypto & app bootstrap keys)	Confi… Integ…	…tion verification
Traceable data/code (Watermarks, finger-prints, traceable keys)	Non-…	…ossible
Code (… security…		
Applic… (license… authen… verification, protocols)		



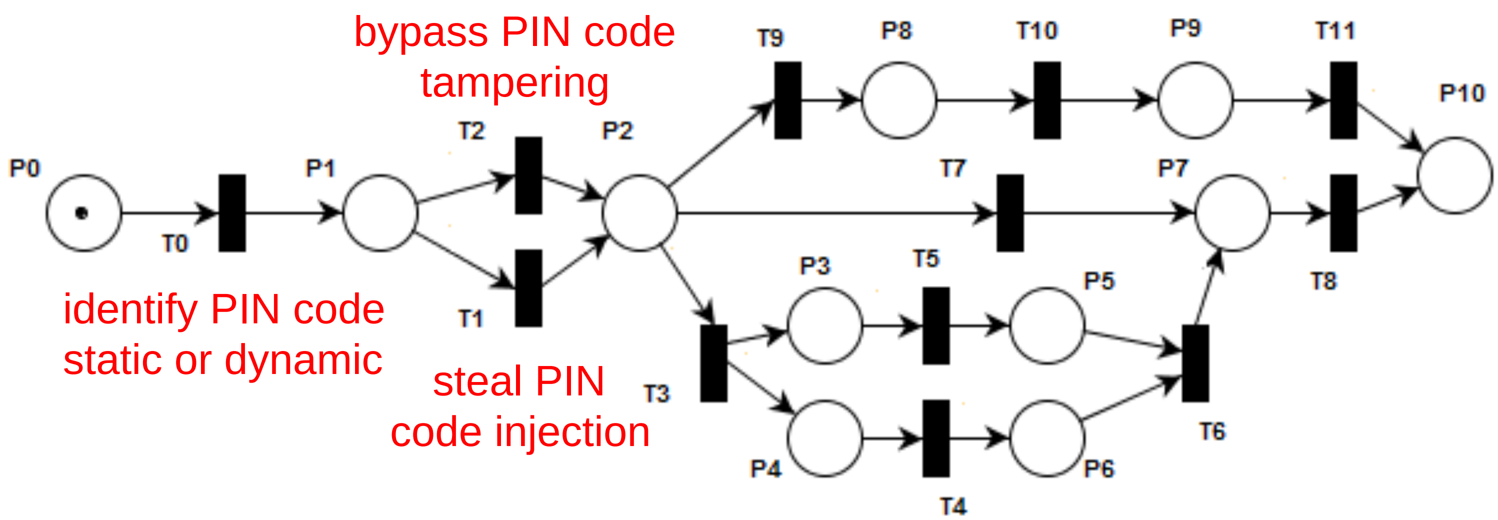
	Identification				Exploitation		
	Guru	Expert	Geek	Amateur	High	Medium	Low
Attack 1	Insecure			Secure			
Attack 2							

Attack (Step) Classification (B. Wyseur)

- static attacks
  - structural code and data recovery (e.g., disassembly, CFG reconstruction)
  - structural matching of binaries
    - against known code (e.g., library identification)
    - of related binaries (e.g., diffing)
  - tampering (e.g., code editing)
- dynamic attacks
  - attacks on communication channels (e.g., sniffing, spoofing, replay attacks)
  - fuzzing
  - debugging (e.g., software or hardware debugger, emulation)
  - structure and data analysis (e.g., unpacking, taint analysis)
  - tampering (e.g., code injection, custom emulation, custom OS)
- hybrid attacks (e.g., concolic execution, static analysis on dynamic graphs)

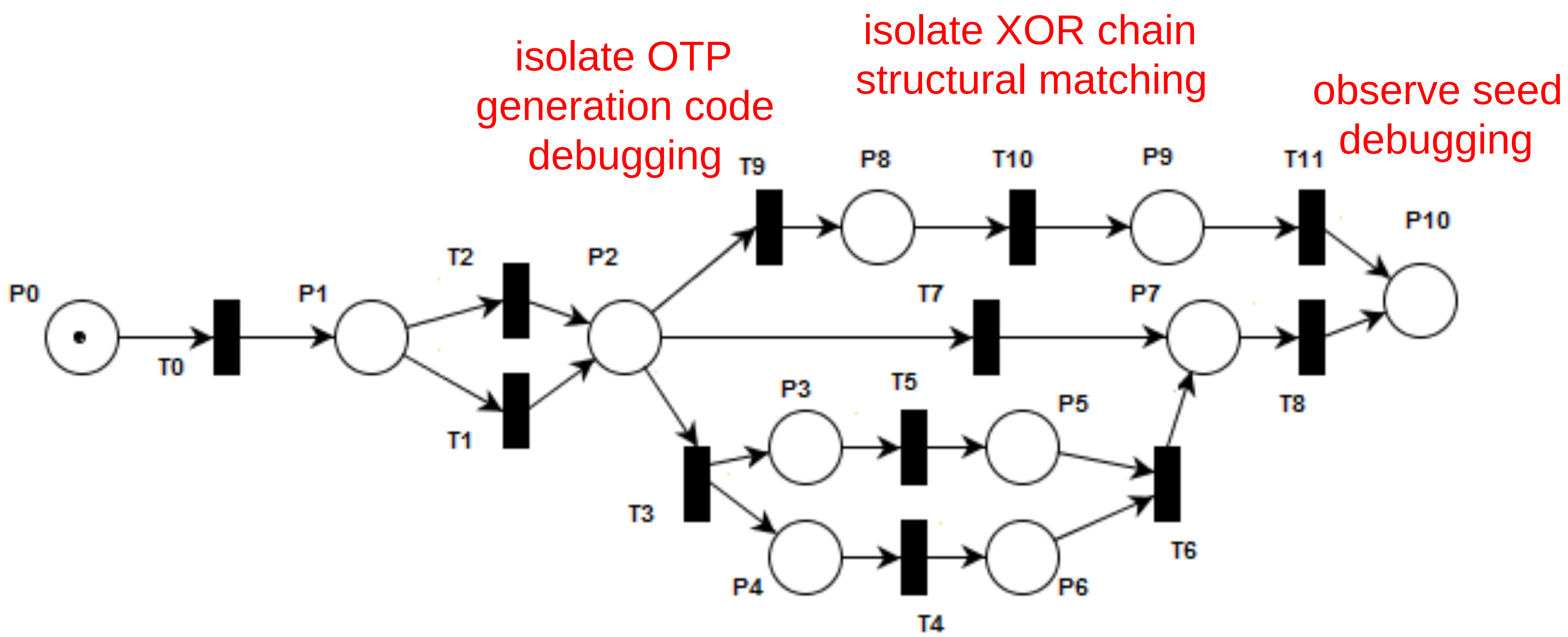
Example attack: One-Time Password Generator (P. Falcarin)

- Step 1: get working provisioning & OTP generation



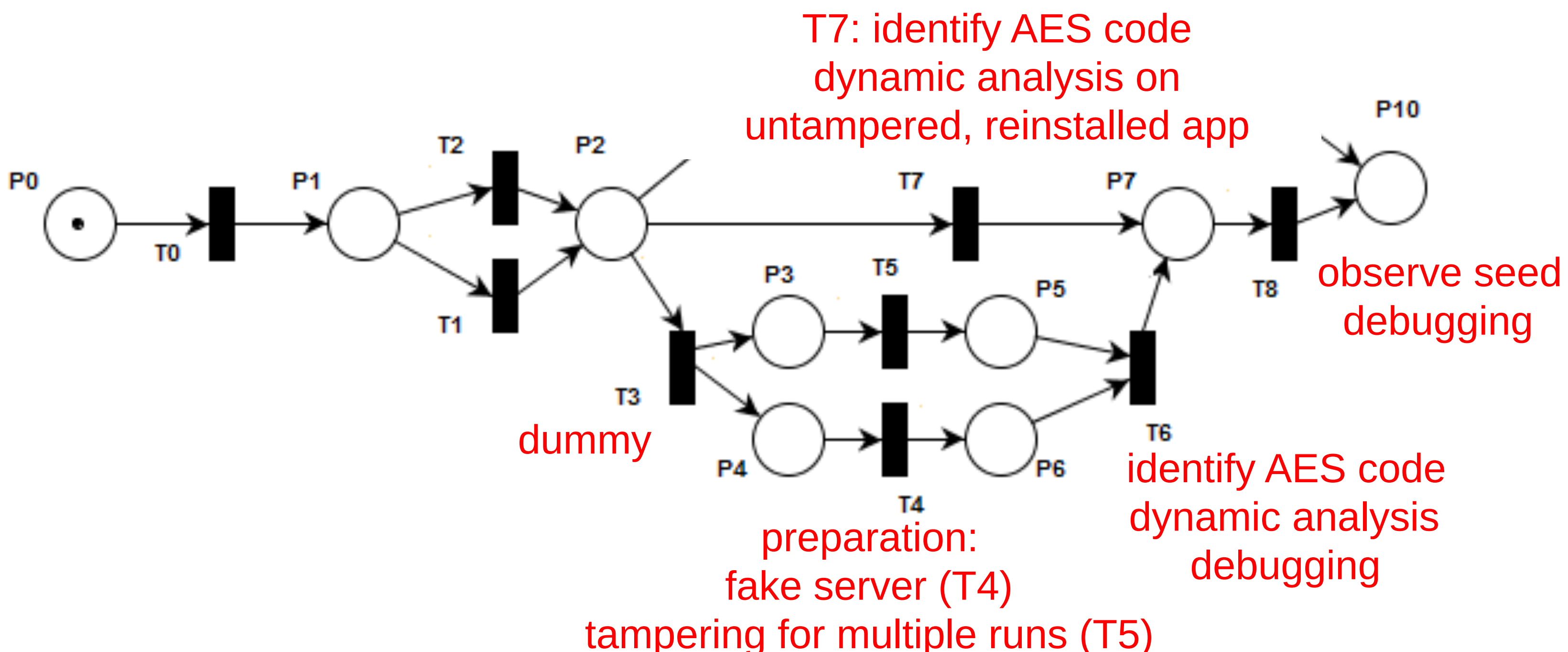
Example attack: One-Time Password generator (P. Falcarin)

- Step 2: retrieve seed of OTP generation
  - during OTP generation



Example attack: One-Time Password generator (P. Falcarin)

- Step 2: retrieve seed of OTP generation
  - alternatively, during provisioning



Overview

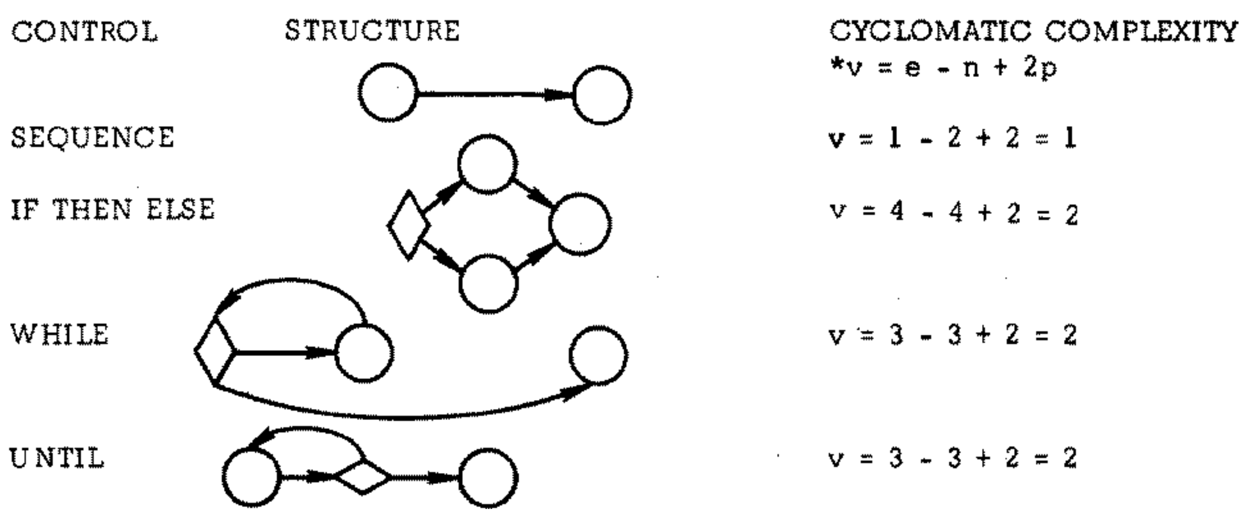
- ASPIRE in a nutshell
- Modelling attacks
- Evaluation Criteria
  - Metrics of complexity
  - Resilience
- Theory versus practice

Cyclomatic number (McCabe, 1976)

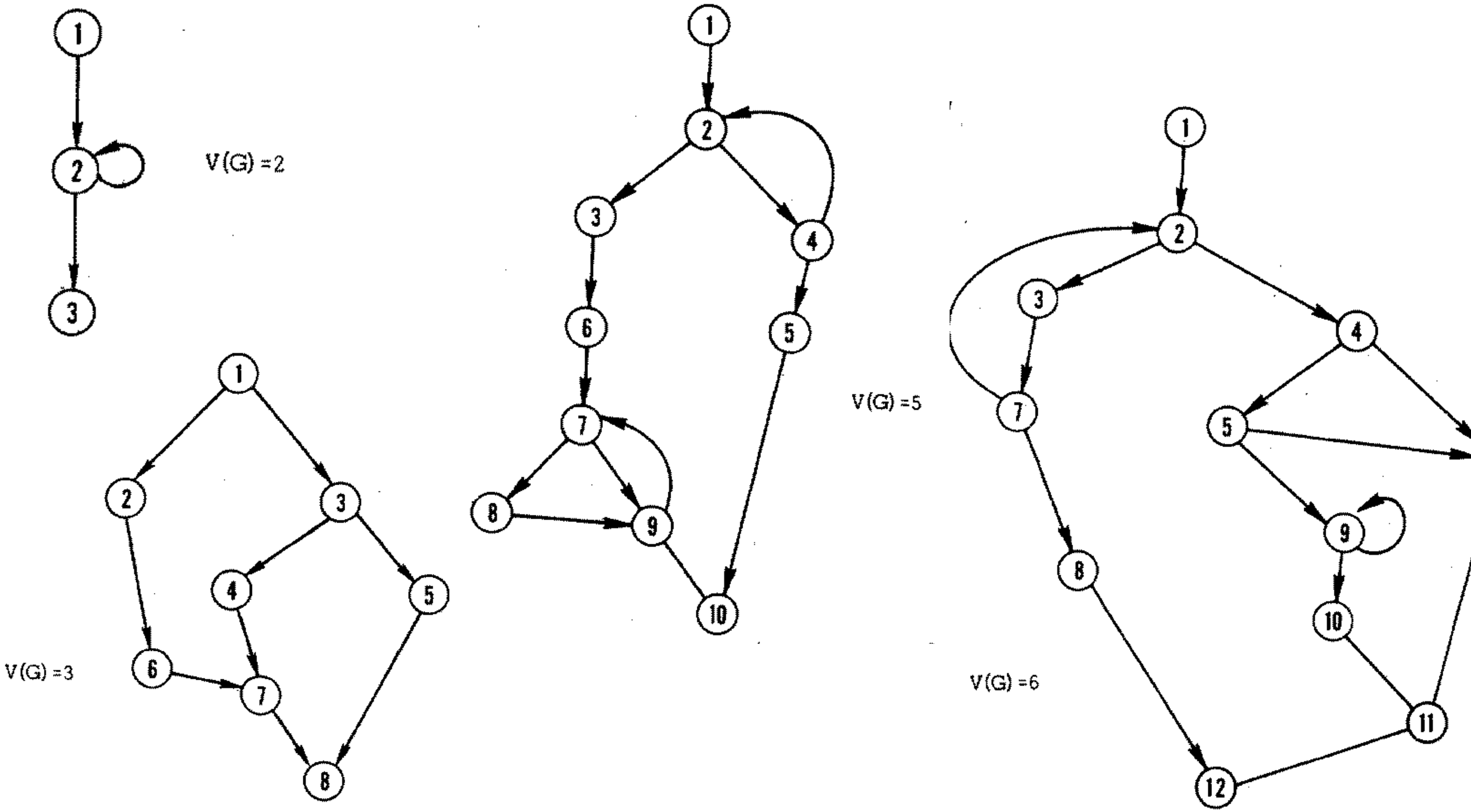
- control flow complexity

  V(cfg) = #edges − #nodes + 2 * #connected components

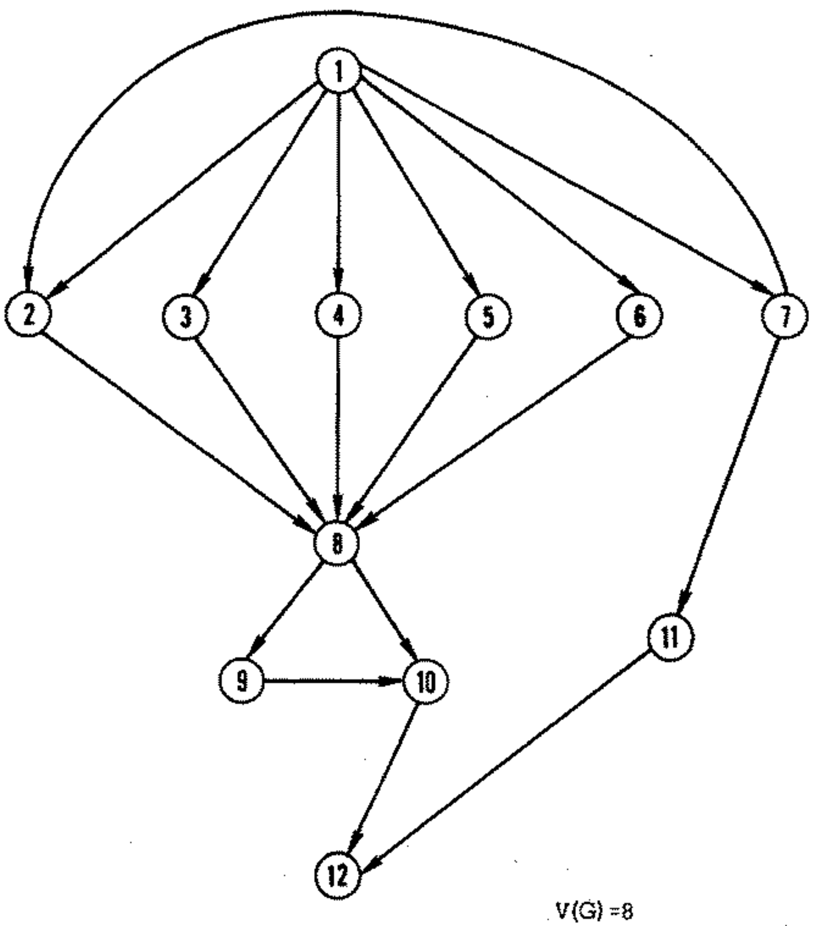
- single components: V(cfg) = #edges − #nodes + 2
- related to the number of linearly independent paths
- related to number of tests needed to invoke all paths

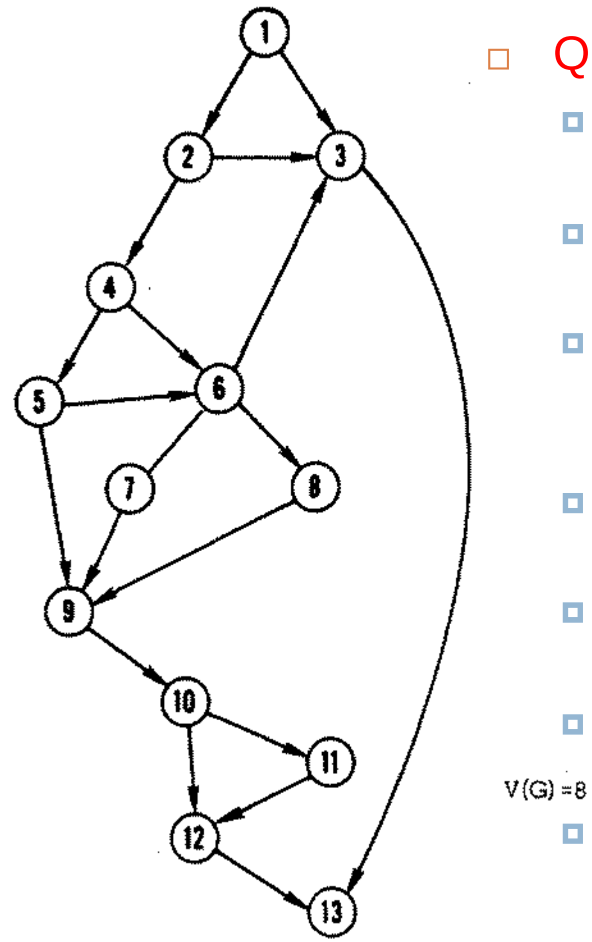


Cyclomatic number (McCabe, 1976)



Cyclomatic number (McCabe, 1976)



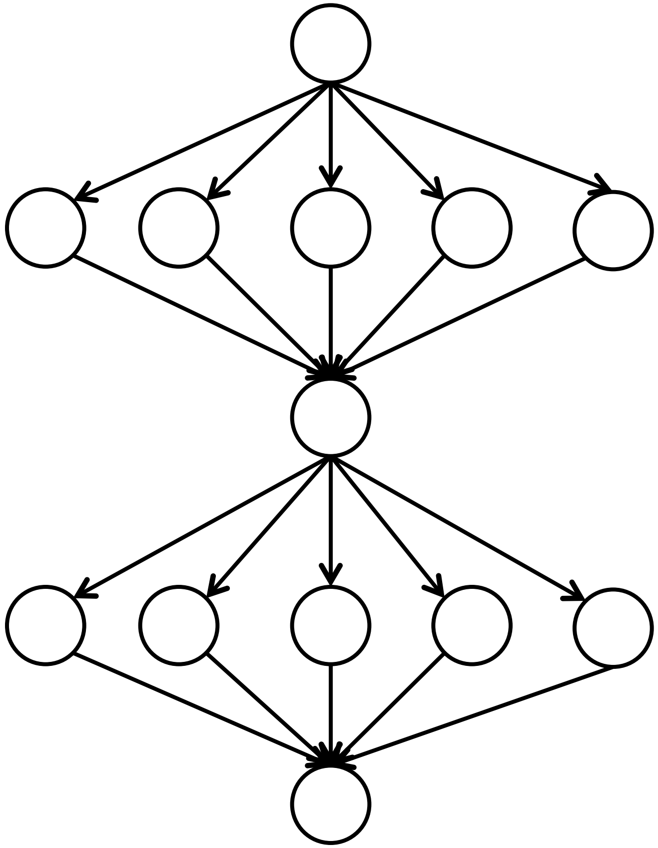


- Quite some problems:
  - no recognition of familiar structures
  - what about obfuscated unstructured CFGs?
  - what to do when functions are not identified well?
  - no recognition of data dependencies
  - what about object-oriented code?
  - what about conditional statements?
  - combinatoric issues

Combinatorics – cognitive problem

(Auprasert and Limpiyakorn, 2008)

- V(cfg) = 20 - 13 + 2 = 9
- But number of paths is 5 * 5 = 25
- Do these switch statements depend on each other?
- Extension by Stetters (1984): F(cfg) computed on ~ PDG



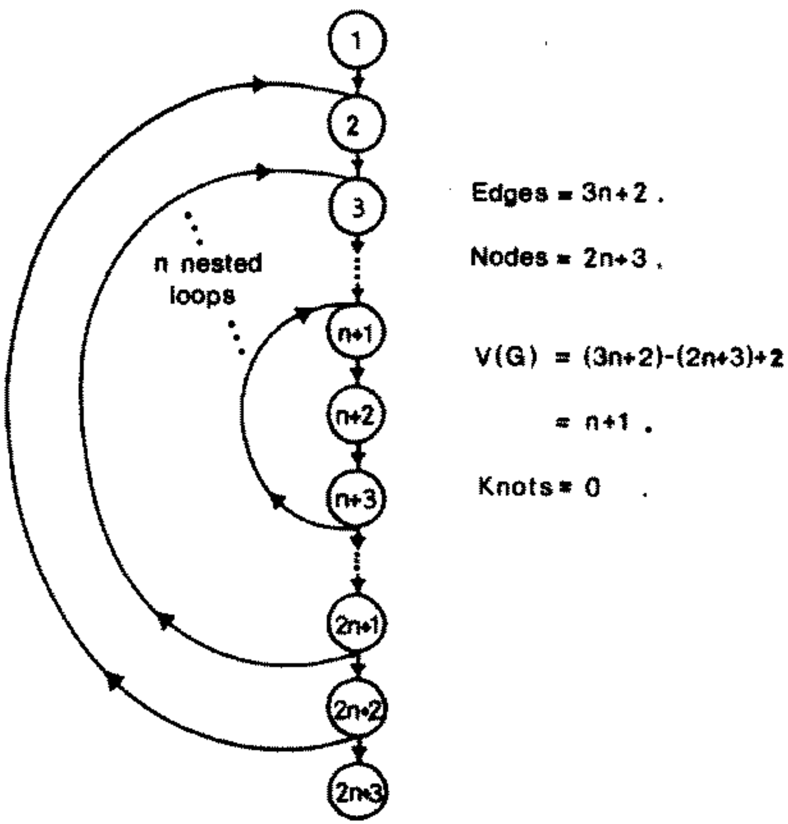
Knot Count (Woodward et al, 1979)

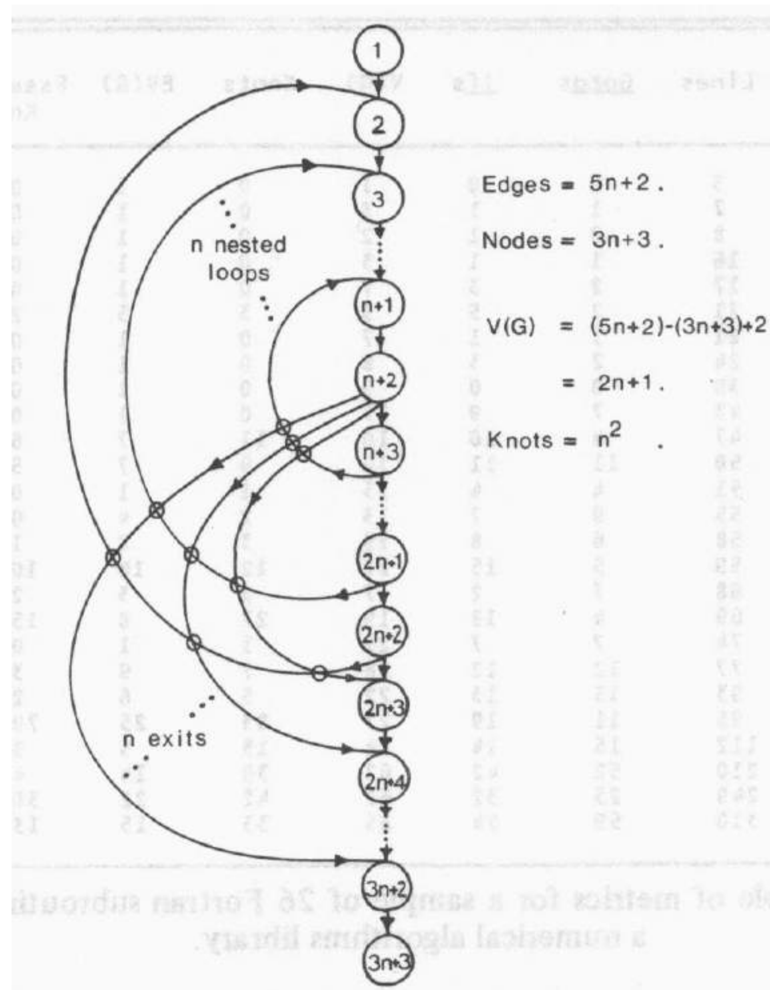
- Try to measure the "unstructuredness":

  knots = #edge crossings in drawn CFG

- Depends on ordering of (Fortran) code
- Complementary to cyclomatic number

Knot Count (Woodward et al, 1979)





Entropy (Giacobazzi & Toppan, 2012)

- Shannon entropy:

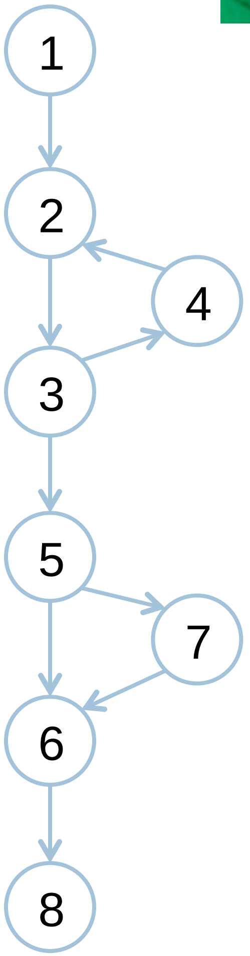
$$H(X) = -\sum_{x \in \mathcal{X}} p_x \cdot log_2(p_x)$$

with X = {nodes/paths/... in a CFG graph}
and p_x = probability of "observing" x

- Different distributions are possible

- Can also be applied to traces:

  ...$(234)^{10}$... $(234)^{10}$... $(234)^{10}$ ...
  ...$(234)^{6}$ ... $(234)^{3}$ ... $(234)^{9}$ ...



Program size & derivatives (Halstead, 1977)

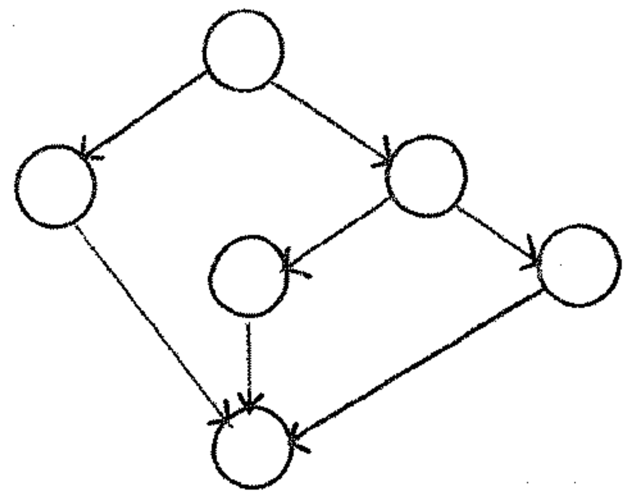
- Lines of code
- Derivatives

n_1 = number of distinct operators,
n_2 = number of distinct operands,
N_1 = total number of operator occurrences, and
N_2 = total number of operand occurrences.

Measure	Symbol	Formula
Program length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N*(\log_2 n)$
Estimated abstraction level	L	$L = (2 n_2) / (n_1*N_2)$
Difficulty	D	$D = 1 / L$
Effort	E	$E = V * D$
Time	T	$T = E / 18$
Remaining bugs	B	$B = E^{2/3} / 3000$

Nesting Depth (Harrison, 1981)

```
if P1 then
      if P2 then
              S1;
         else
              S2;
      else
         S3;
S4;
```



- Halstead metrics for local complexity (basic blocks)
- Complexity node = local complexity + complexity range of selection nodes

Information Flow (Henry & Kafura, 1981)

- Metric for procedures, interfaces and modules
- Measures complexity in relation to bug fixes
  - Count data dependencies between entities
  - Combine with code length

Measure	Correlation to Changes	Level of Significance
(fan-in * fan-out)**2	.98	.028
length*(fan-in * fan-out)**2	.94	.021
(fan-in * fan-out)	.83	.042
(length**2)	.60	.078

Combined code & data flow complexity (Oviedo, 1980)

C(cfg) = a * #edges + b * #live-in variables all nodes

Chunks (Davis, 1984)
Cognitive Functional Size (Wang and Shao, 2003)

- Based on cognitive sciences – psychology
- Different weight for different types of basic control structures

$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right]$$

Category	**BCS**	**W_i**
Sequence	Sequence (SEQ)	1
Branch	If–Then-Else (ITE)	2
	Case (CASE)	3
Iteration	For-do (R_i)	3
	Repeat-until (R_1)	3
	While-do (R_0)	3
Embedded Component	Function Call (FC)	2
	Recursion (REC)	3
Concurrency	Parallel (PAR)	4
	Interrupt (INT)	4

- What about unstructured programs?

Human Comprehension Models (Nakamura et al, 2003)

- Comprehension ~ mental simulation of a program
- Model the brain, pen & paper as a simple CPU
- CPU performance is driven by misses
  - cache misses
  - TLB misses
  - prediction
- Measure misses with small sizes of memory

Human Comprehension Models (Nakamura et al, 2003)

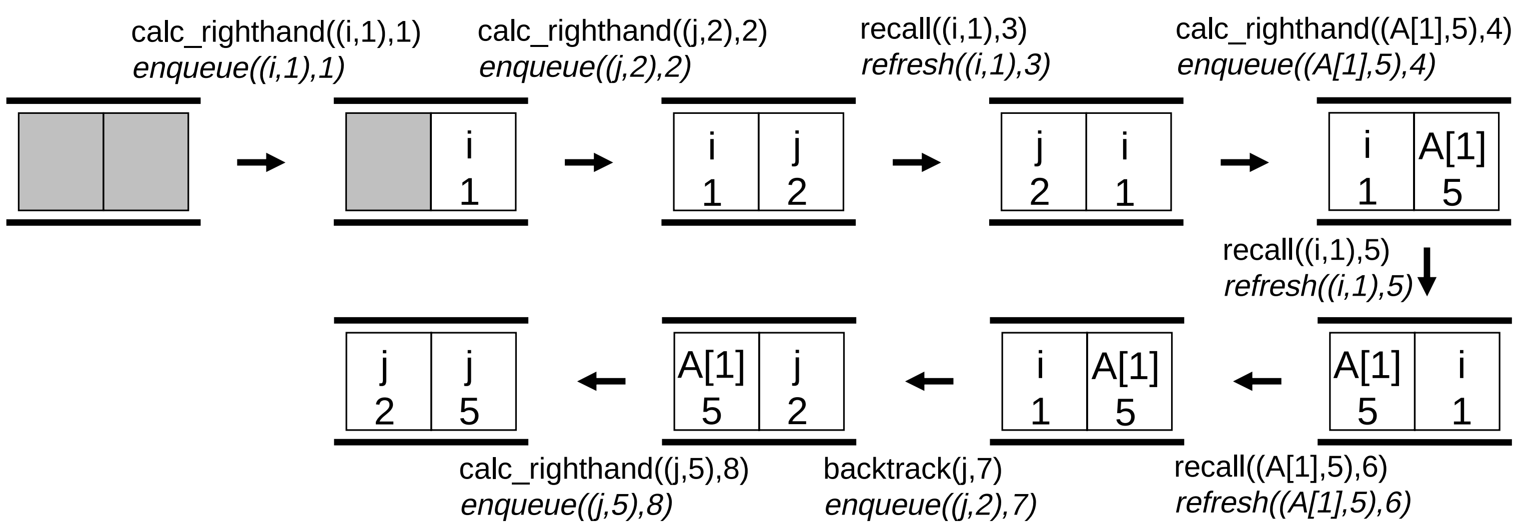

```
i = 1 ;
j = 2 ;
A[1] = i + 4 ;

j = A[i] - j ;
```

(a)

```
i     1  assignment
j     2  assignment
i     1  reference
A[1]  5  assignment
i     1  reference
A[1]  5  reference
j     2  reference
j     3  assignment
```

(b)



Combine all of them (Anckaert et al, 2007)

1. code & code size
   - e.g., #instructions, weighted by "complexity"
2. control flow complexity
3. data flow complexity
   - sizes slices
   - sizes live sets, working sets
   - sizes points-to sets
   - fan-in, fan-out (Oviedo)
   - data structure complexities (Munson and Khoshgoftaar, 1993)
4. data
   - application-specific

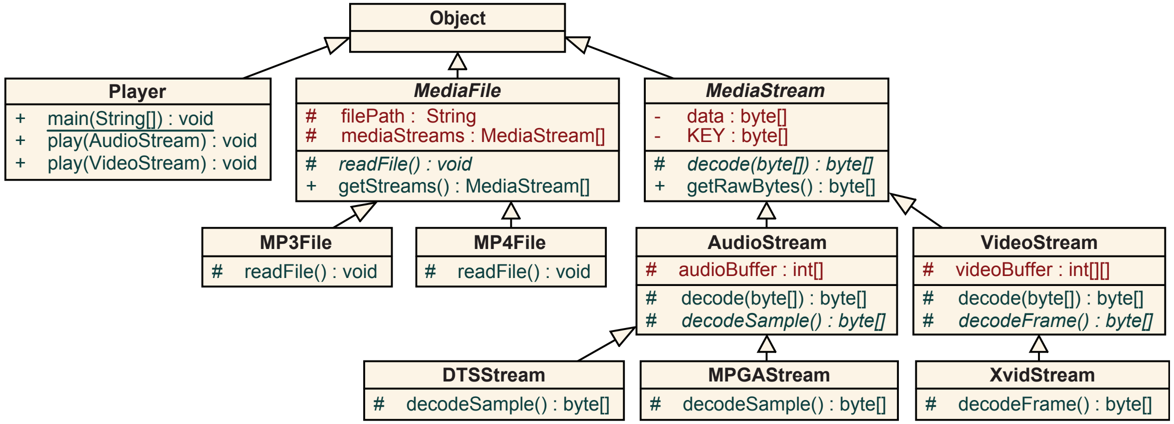
static -> graphs

dynamic -> traces

Object-Oriented Quality Metrics (Bansiya & Davis, 2002)

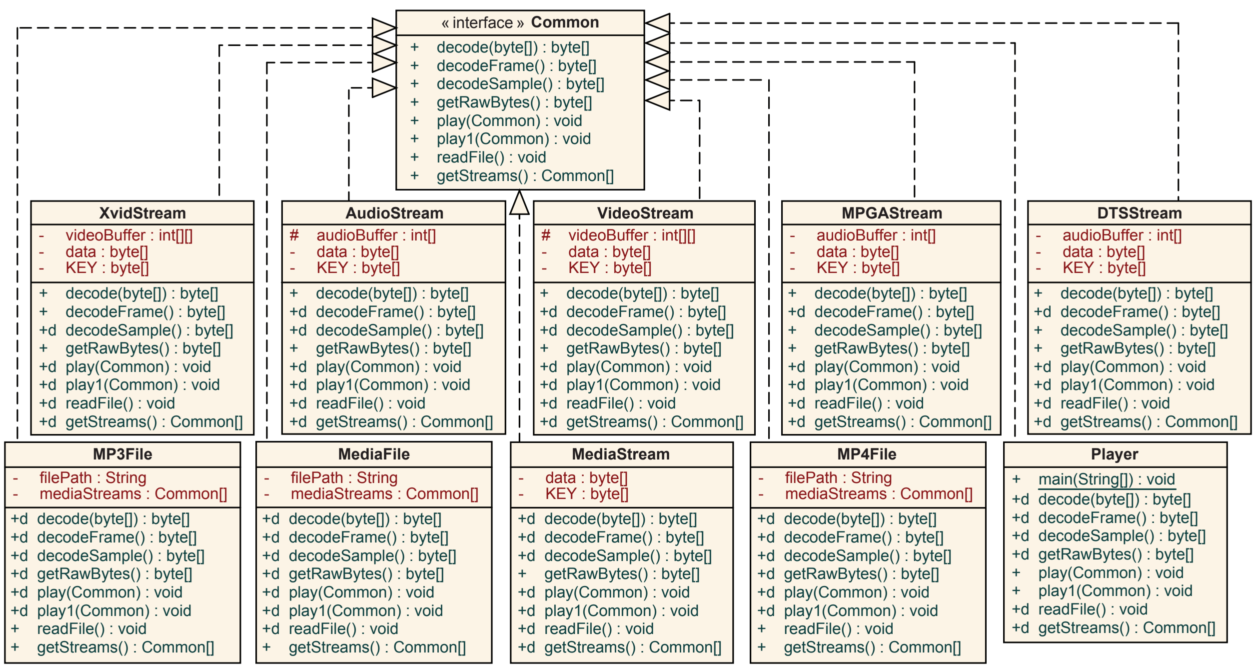
- QMOOD: Quality Model for Object-Oriented Design
  - abstraction
  - encapsulation
  - coupling
  - cohesion
  - polymorphism
  - complexity
  - design size
  - ...
- Weighted averages:
  - understandability
  - maintainability

Example: class hierarchy flattening (Foket et al, 2014)



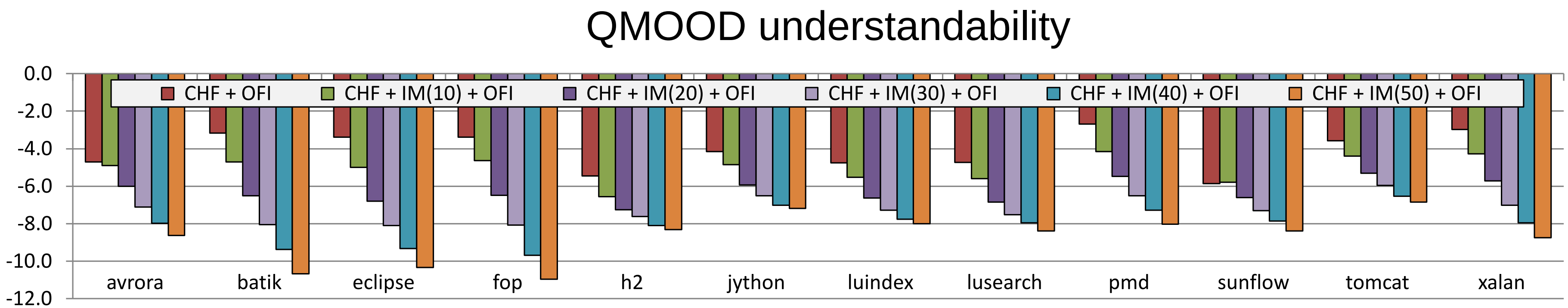

```
public class Player {
  public void play(AudioStream as) {
    /* send as.getRawBytes() to audio device */
  }
  public void play(VideoStream vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Player player = new Player();
    MediaFile[] mediaFiles = ...;
    for (MediaFile mf : mediaFiles)
      for (MediaStream ms : mf.getStreams())
        if (ms instanceof AudioStream)
          player.play((AudioStream)ms);
        else if (ms instanceof VideoStream)
          player.play((VideoStream)ms);
  }
}
public class MP3File extends MediaFile {
  protected void readFile() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    AudioStream as = new MPGAStream(data);
    mediaStreams = new MediaStream[]{as};
    return;
  }
}
public abstract class MediaStream {
  public static final byte[] KEY = ...;
  public byte[] getRawBytes() {
    byte[] decrypted = new byte[data.length];
    for (int i = 0; i < data.length; i++)
      decrypted[i] = data[i] ^ KEY[i];
    return decode(decrypted);
  }
  protected abstract byte[] decode(byte[] data);
}
```

Example: class hierarchy flattening (Foket et al, 2014)




```
public class Player implements Common {
  public byte[] merged1(Common as) {
    /* send as.getRawBytes() to audio device */
  }
  public Common[] merged2(Common vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Common player = CommonFactory.create(…);
    Common[] mediaFiles = ...;
    for (Common mf : mediaFiles)
      for (Common ms : mf.getStreams())
        if (myCheck.isInst(0, ms.getClass()))
          player.merged1(ms);
        else if (myCheck.isInst(1, ms.getClass()))
          player.merged2(ms);
  }
}
public class MP3File implements Common {
  public byte[] merged1() {
    InputStream inputStream = ...;
    byte[] data = new byte[...];
    inputStream.read(data);
    Common as = CommonFactory.create(…);
    mediaStreams = new Common[]{as};
    return data;
  }
}
public class MediaStream implements Common {
  public static final byte[] KEY = ...;
  public byte[] getRawBytes() {
    byte[] decrypted = new byte[data.length];
    for (int i = 0; i < data.length; i++)
      decrypted[i] = data[i] ^ KEY[i];
    return decode(decrypted);
  }
  public byte[] decode(byte[] data){ … }
}
```

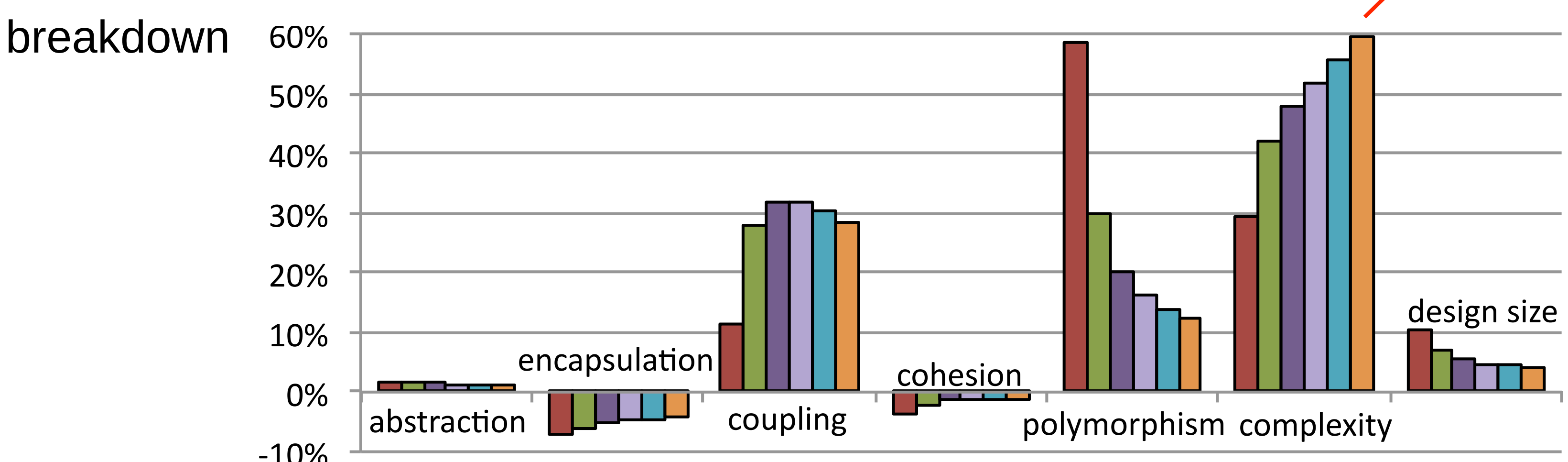
Object-Oriented Quality Metrics (Bansiya & Davis, 2002)



90% of classes transformed

25% of classes transformed

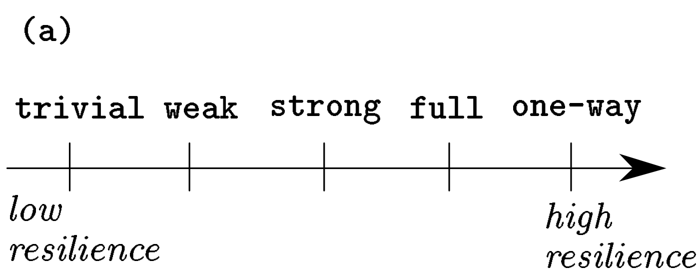
dominating term

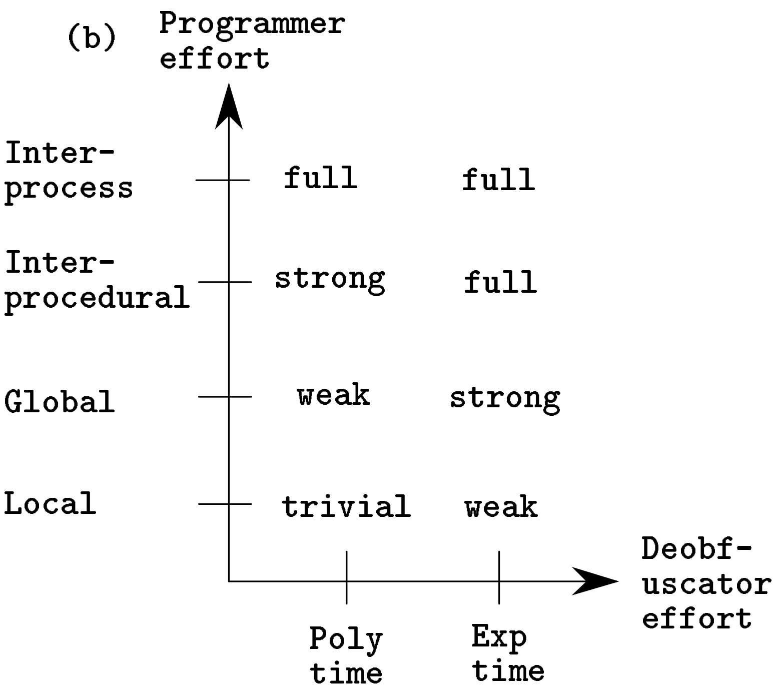


Overview

- ASPIRE in a nutshell
- Modelling attacks
- Evaluation Criteria
  - Metrics of complexity
  - Resilience
- Theory versus practice: involving the humans

Resilience (Collberg et al, 1997)





Abstract Interpretation (Dalla Preda, Giacobazzi et al)

- Abstract domains model program properties
  Abstract interpretation computes properties
- Domains are partially ordered in terms of concreteness
- Obfuscating transformation is less potent if it preserves more concrete properties
- Automatic deobfuscation of opaque predicates, e.g., f(x) | nZ
- Not clear how this scales ...

Abstract Interpretation (Dalla Preda, Giacobazzi et al)

```
c = a * b;
d = c + 3;
```

$var_1 = var_2 \; op \; var_3 + const$

```
c = a * b;
p = algebraic_function(a,b,c,x,y,z);
if (p == 0)
  d = c + 3;
else
  a = c - d;
```

```
c = a * b;
p = check_aliasing_in_a_graph(a,b,c,g,h,k);
if (p == false)
  d = c + 3;
else
  a = c - d;
```

Tool-based metrics

- Use attacker's tools and heuristics
  1. Model effort/time in terms of input size
  2. Compute output size
  3. Compute relevance of output
     - false positives/negatives
     - receiver operator curves (ROC)
     - recall and precision
     - pruning factors
- Major problems:
  - predicting tool output
  - generallity of the results

Example 1: Disassembly Thwarting (Linn & Debray, 2003)

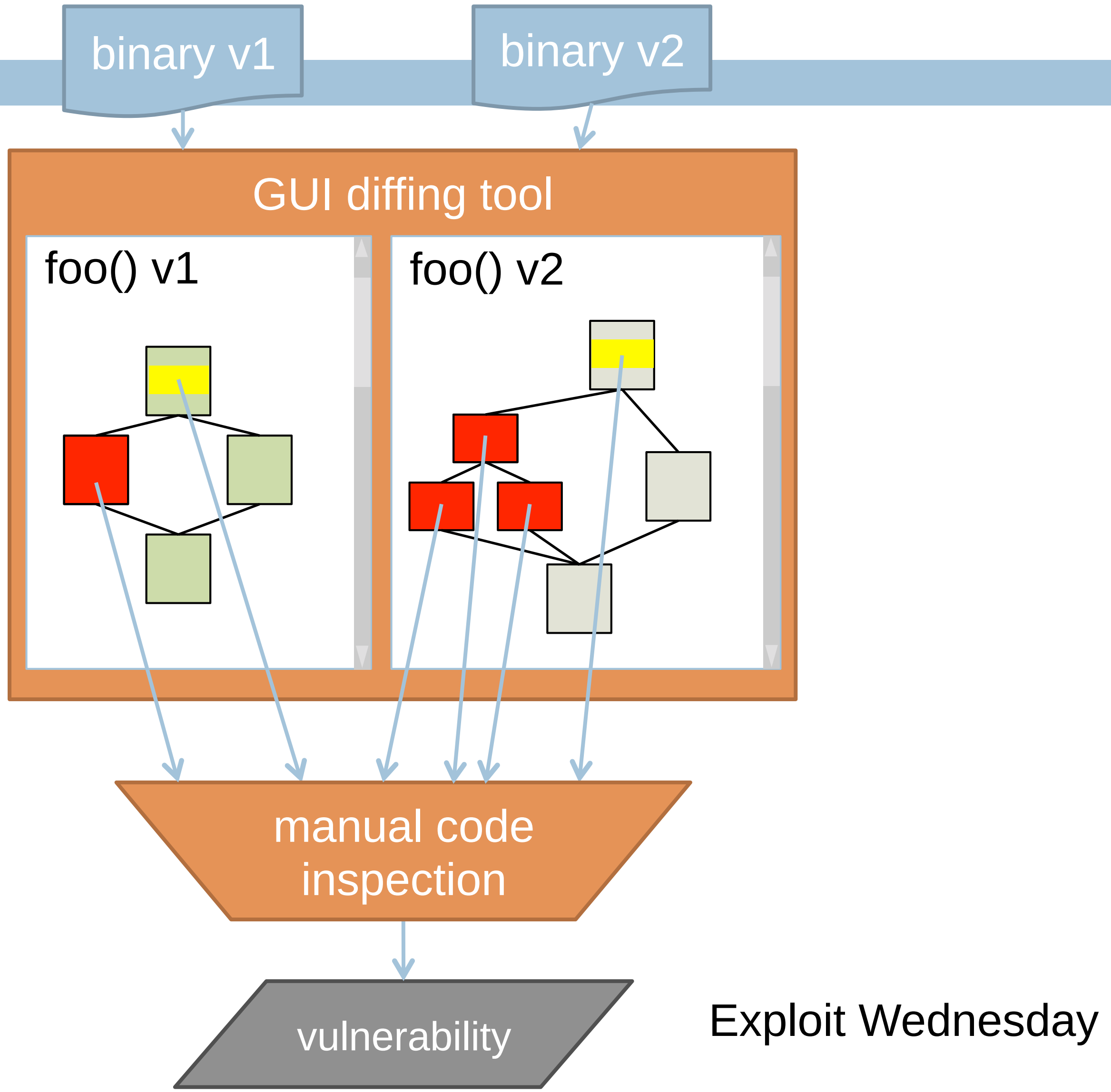
- Confusion factor

$$CF = |A - P| / |A|.$$

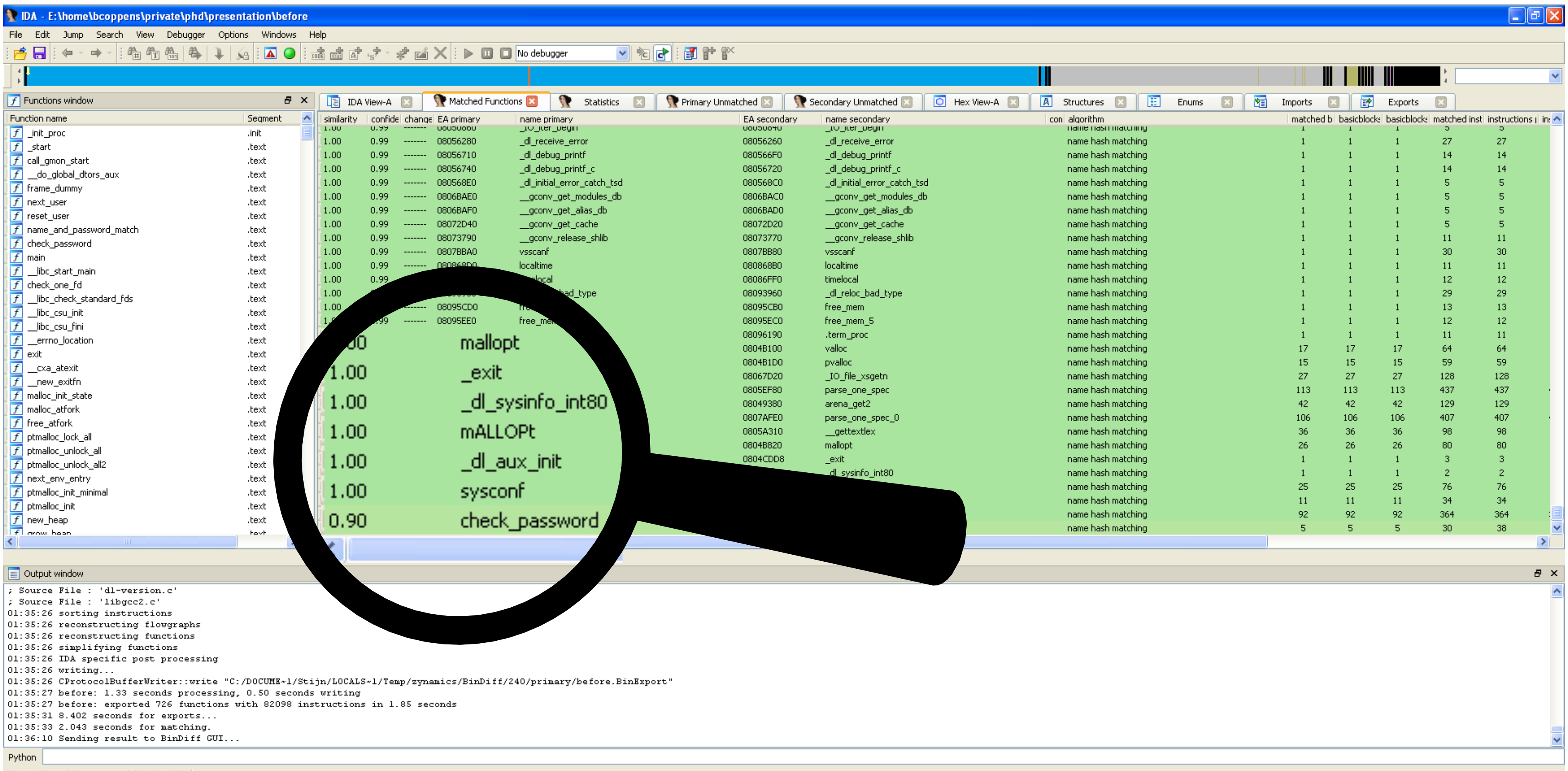
with A = ground truth set of instruction addresses and P = set determined by static disassembly

Program	Confusion factor (%)								
	Linear sweep (Objdump)			Recursive traversal			Commercial (IDA Pro)		
	Instructions	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*
compress95	43.93	63.68	100.00	30.04	40.42	75.98	75.81	91.53	87.37
gcc	34.46	53.34	99.53	17.82	26.73	72.80	54.91	68.78	82.87
go	33.92	51.73	99.76	21.88	30.98	60.56	56.99	70.94	75.12
ijpeg	39.18	60.83	99.75	25.77	38.04	69.99	68.54	85.77	83.94
li	43.35	63.69	99.88	27.22	38.23	76.77	70.93	87.88	84.91
m88ksim	41.58	62.87	99.73	24.34	35.72	77.16	70.44	87.16	87.16
perl	42.34	63.43	99.75	27.99	39.82	76.18	68.64	84.62	87.13
vortex	33.98	55.16	99.65	23.03	35.61	86.00	57.35	74.55	91.29
Geo. mean	39.09	59.34	99.75	24.76	35.69	74.43	65.45	81.40	84.97

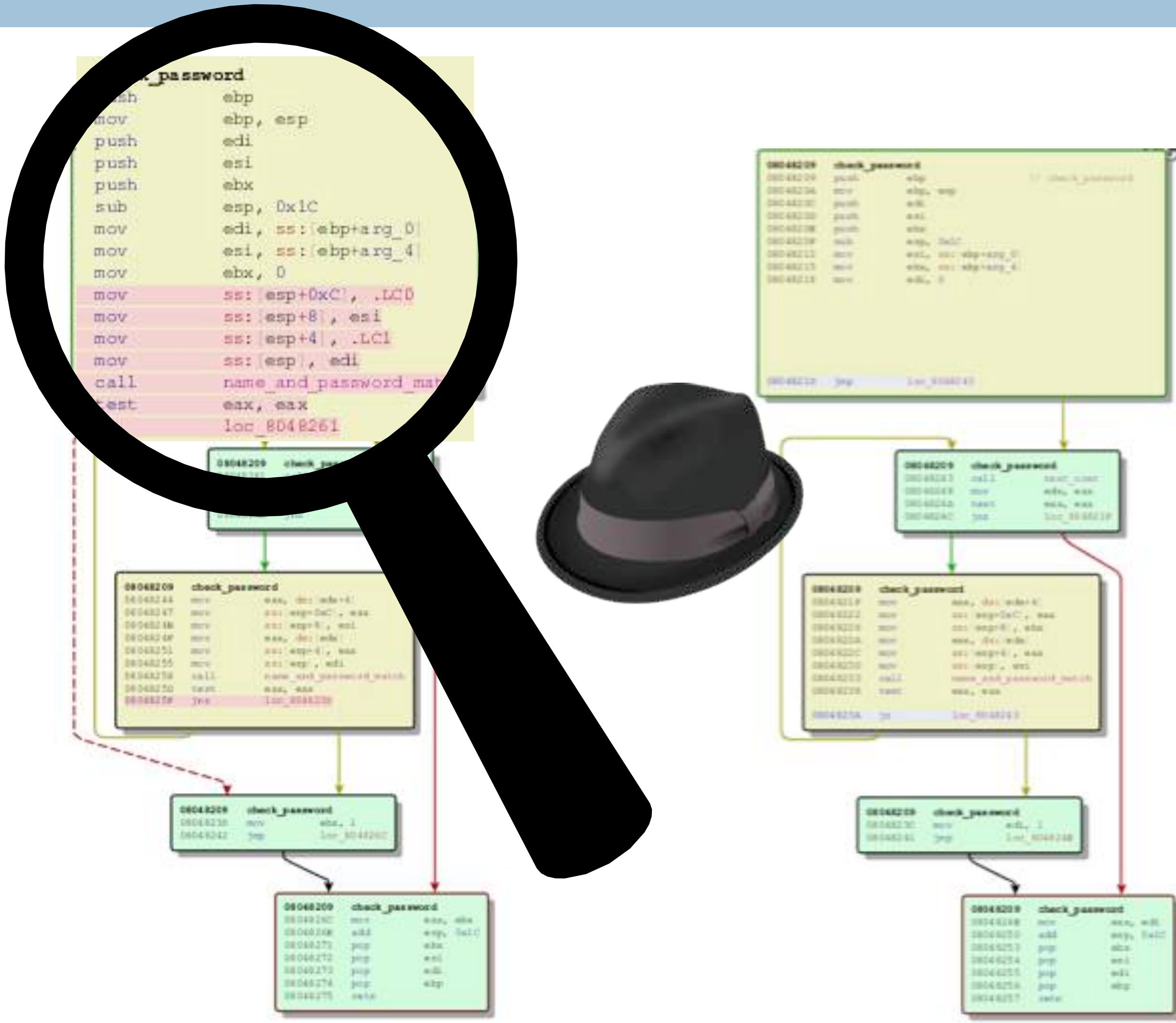
Example 2: Patch Tuesday (Coppens et al, 2013)



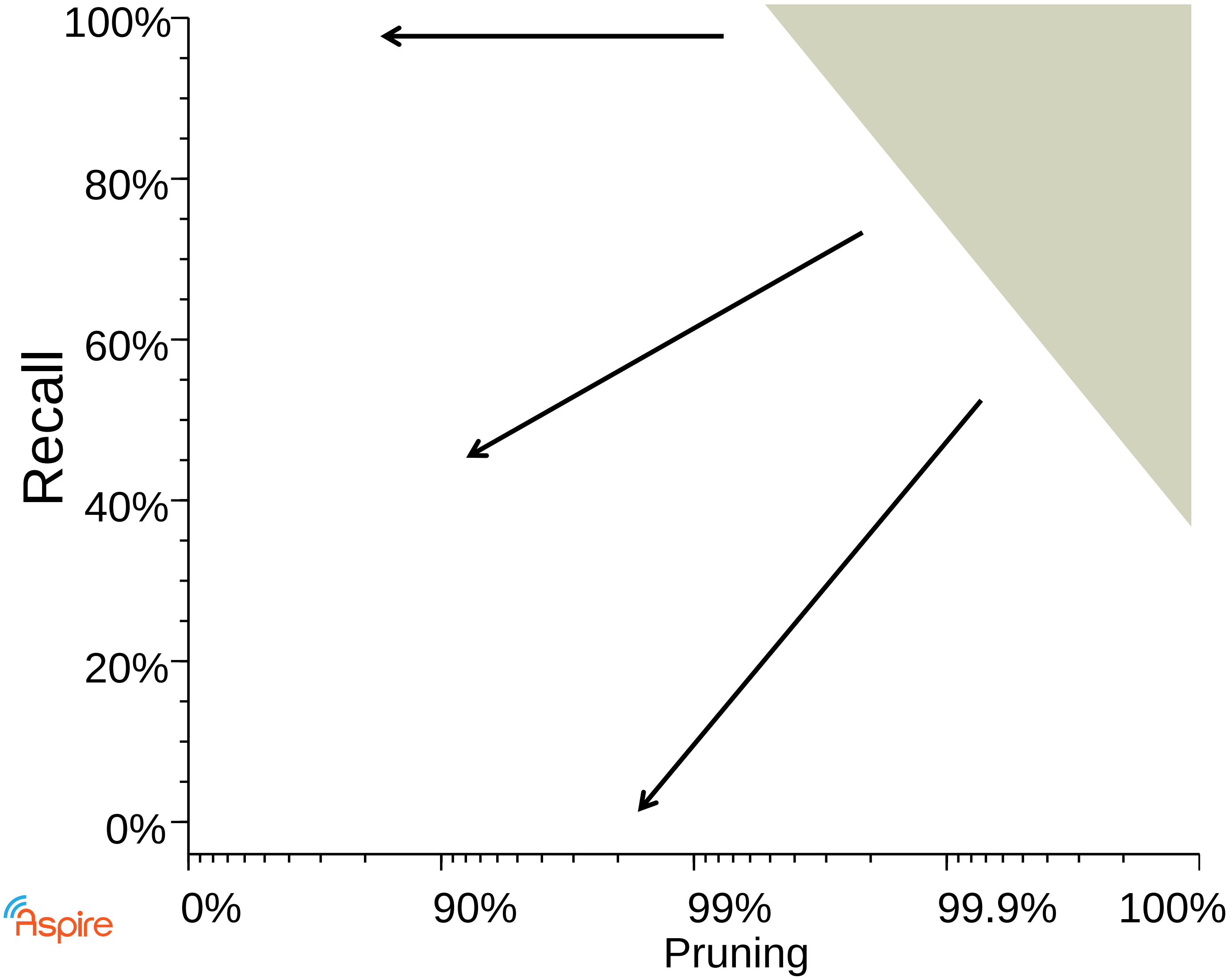
BinDiff on Patch Tuesday



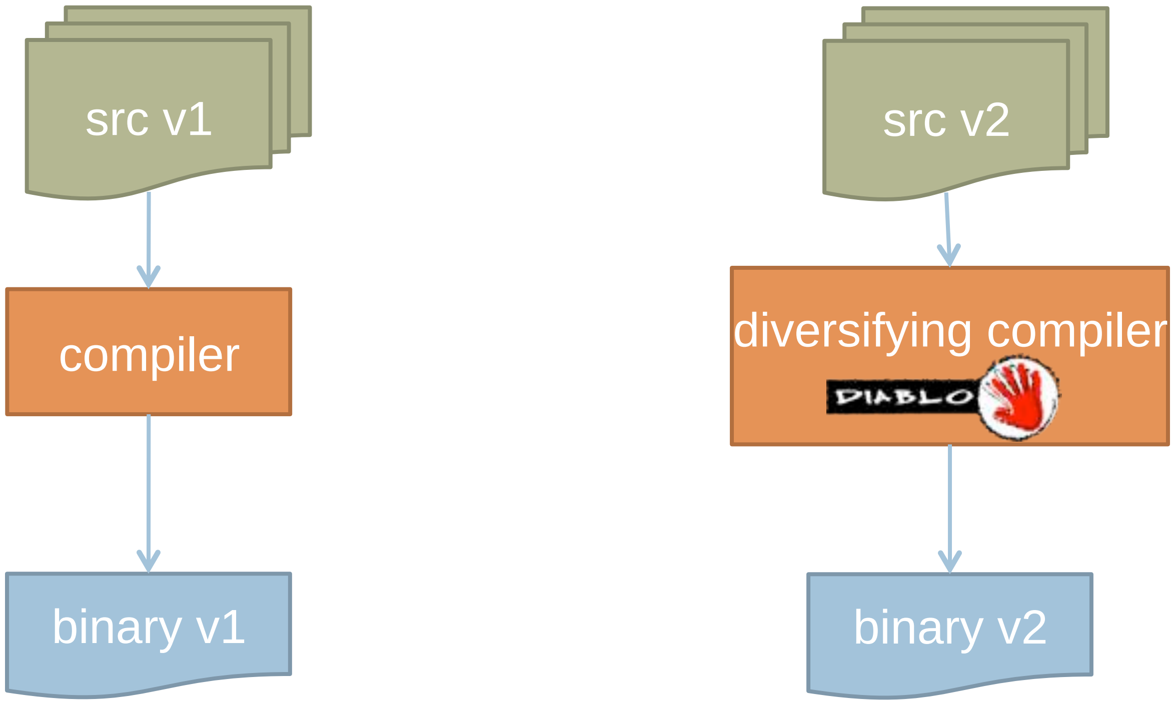
BinDiff on Patch Tuesday



BinDiff on Patch Tuesday



Software Diversification



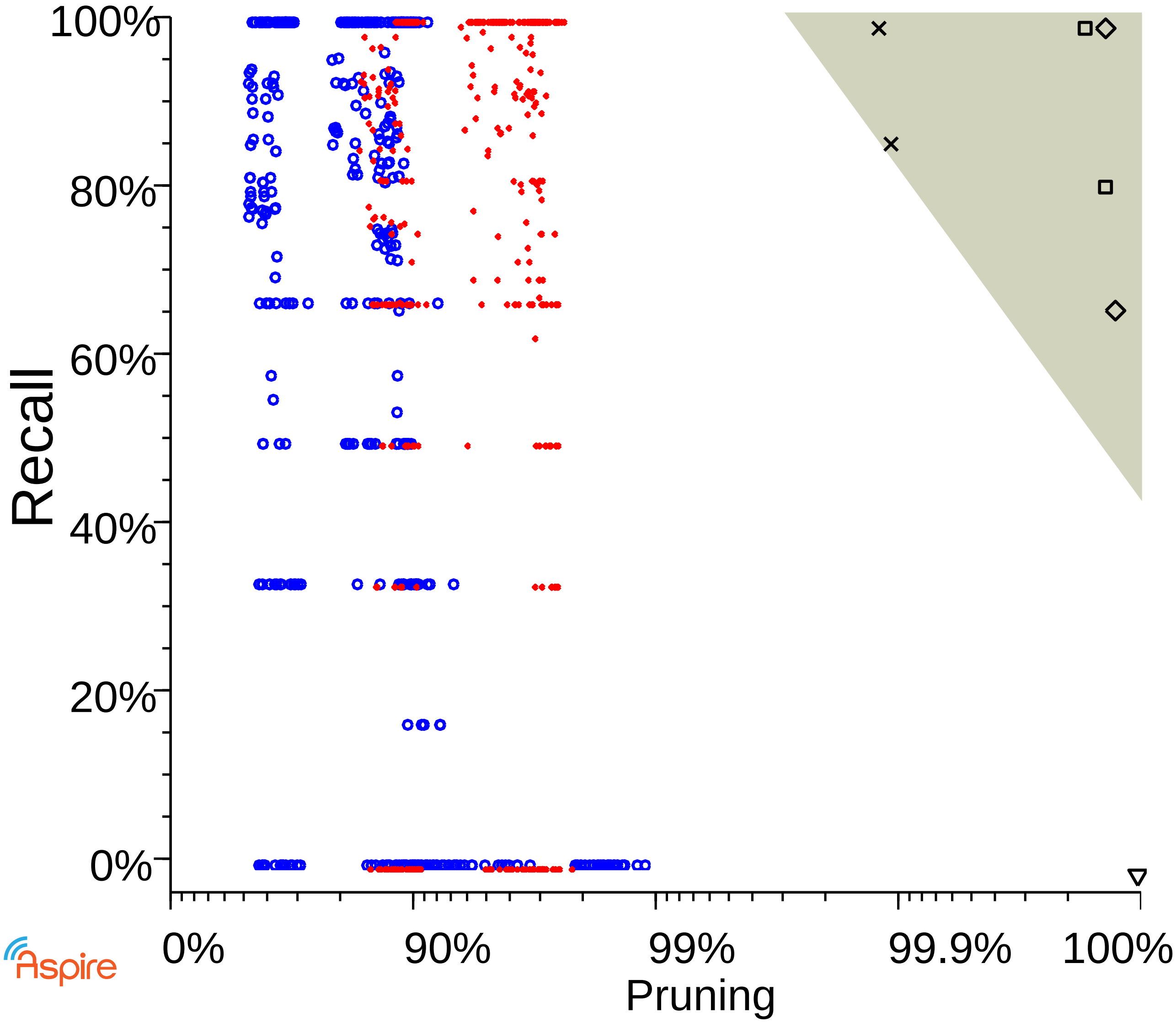
Bindiff on Patch Tuesday

IDA View-A | Matched Functions | Statistics | Primary Unmatched | Secondary Unmatched | Hex View-A | Structures | Enu

similarit	confide	change	EA primary	name primary	EA secondary	name secondary	con	algorithm
0.24	0.44	GI--E--	08076887	sub_8076887_384	0808D8C1	sub_808D8C1_1458		call reference matching
0.25	0.40	GI--E--	08063B6D	sub_8063B6D_265	0804F0A3	sub_804F0A3_701		call sequence matching(sequence)
0.25	0.83	GI-J-L-	0807C115	sub_807C115_453	0804EE07	sub_804EE07_698		call sequence matching(exact)
0.25	0.71	GI-JE--	080907C9	sub_80907C9_607	08055303	sub_8055303_785		call sequence matching(sequence)
0.26	0.47	GI-JE...	0804A8FC	sub_804A8FC_21	0805CEC1	sub_805CEC1_866		call sequence matching(sequence)
0.26	0.48	GI--E--	08057B75	sub_8057B75_86	080582C9	sub_80582C9_834		edges callgraph MD index
0.29	0.54	GI--E--	0805719A	sub_805719A_74	08058655	sub_8058655_836		edges callgraph MD index
0.29	0.69	GI-JEL-	08054BA4	sub_8054BA4_43	080872D6	sub_80872D6_1374		call sequence matching(sequence)
0.30	0.99	G----L-	0808223A	sub_808223A_535	08063A05	sub_8063A05_949		call reference matching
0.31	0.94	GI---LC	080484E8	sub_80484E8_7	080613BD	sub_80613BD_916		call reference matching
0.31	0.41	GI--E--	0807F7FA	sub_807F7FA_506	08050C49	sub_8050C49_714		call sequence matching(sequence)
0.32	0.64	GI--E--	0808D103	sub_808D103_599	0807E1CE	sub_807E1CE_1261		call sequence matching(sequence)
0.35	0.99	GI-----	08078564	sub_8078564_415	08094E92	sub_8094E92_1545		string references
0.37	0.66	GI--EL-	0806379D	sub_806379D_263	0804E306	sub_804E306_690		call sequence matching(sequence)
0.37	0.99	GI-----	08084439	sub_8084439_573	080810BC	sub_80810BC_1304		call reference matching
0.39	0.99	G----L-	0807E025	sub_807E025_473	08077DDC	sub_8077DDC_1193		call reference matching
0.39	0.99	G----L-	08064C7E	sub_8064C7E_277	08082C32	sub_8082C32_1330		string references
0.39	0.73	GI--E--	0806146A	sub_806146A_244	0804ED78	sub_804ED78_697		call sequence matching(sequence)
0.40	0.99	G------	08048C37	sub_8048C37_13	0808B713	sub_808B713_1424		call reference matching
0.40	0.99	G----L-	0805A8AE	sub_805A8AE_153	08068268	sub_8068268_1005		call reference matching
0.41	0.99	GI---L-	08077B5D	sub_8077B5D_412	0807F3D5	sub_807F3D5_1278		call reference matching
0.42	0.73	GI-JE--	080841A5	sub_80841A5_572	0805B05A	sub_805B05A_863		call sequence matching(sequence)
0.42	0.99	GI---L-	0805510E	sub_805510E_46	0805A265	sub_805A265_854		call reference matching
0.42	0.98	GI---L-	0805ABB4	sub_805ABB4_155	0807BB38	sub_807BB38_1234		string references
0.43	0.99	GI---L-	0807D67A	sub_807D67A_466	08089E5C	sub_8089E5C_1406		call reference matching
0.44	0.81	GI-J-L-	080486F6	sub_80486F6_11	08080EAA	sub_8080EAA_1303		call reference matching
0.44	0.99	G----L-	0805F728	sub_805F728_232	08073AAD	sub_8073AAD_1154		call reference matching

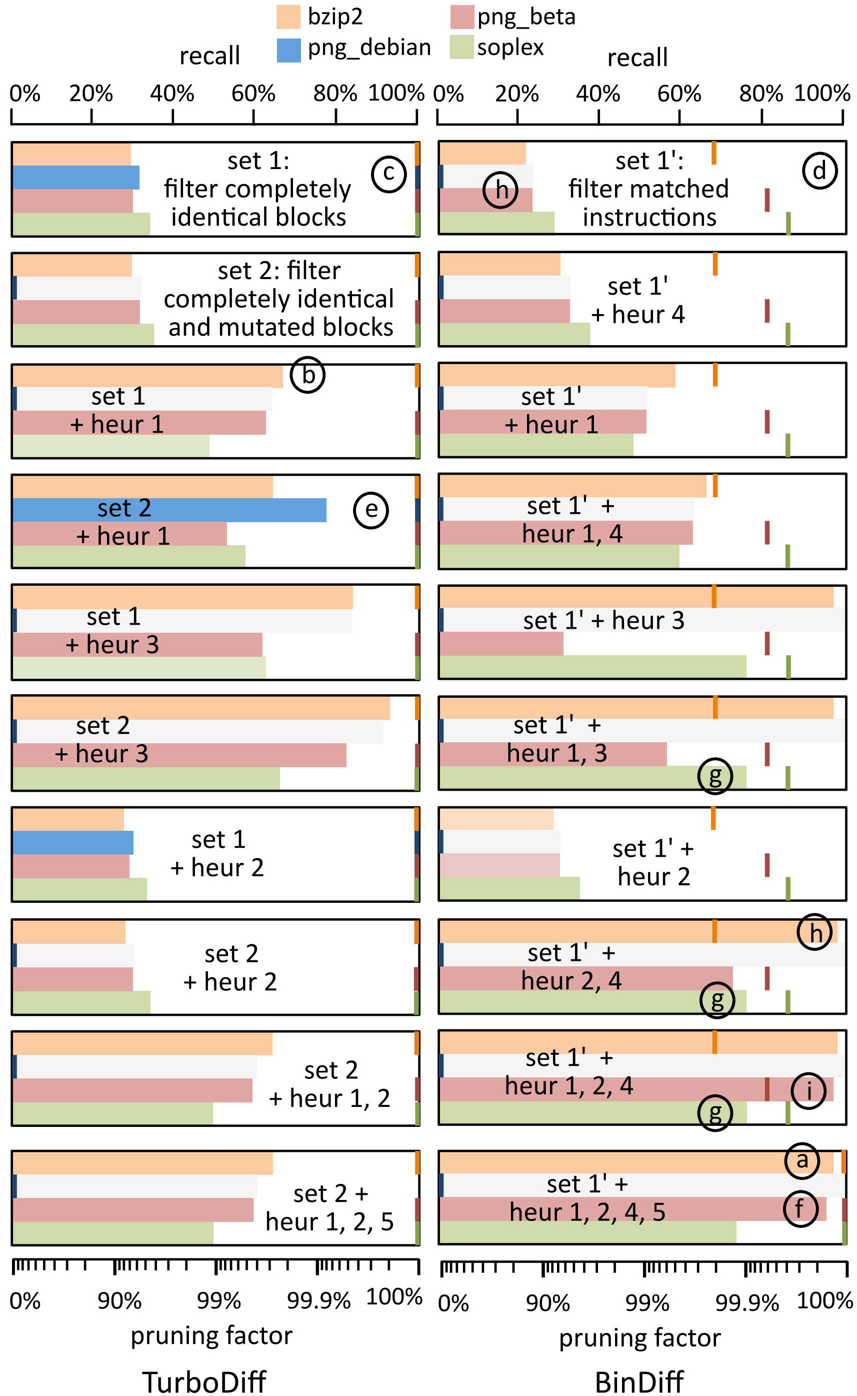
Line 616 of 616

BinDiff on Diversified Code



Aspire

Other tools


bzip2
png_beta
png_debian
soplex
recall
0%
20%
40%
60%
80%
100%
set 1:
filter completely
identical blocks
set 2: filter
completely identical
and mutated blocks
set 1
+ heur 1
set 2
+ heur 1
set 1
+ heur 3
set 2
+ heur 3
set 1
+ heur 2
set 2
+ heur 2
set 2
+ heur 1, 2
set 2 +
heur 1, 2, 5
set 1':
filter matched
instructions
set 1'
+ heur 4
set 1'
+ heur 1
set 1' +
heur 1, 4
set 1' + heur 3
set 1' +
heur 1, 3
set 1' +
heur 2
set 1' +
heur 2, 4
set 1' +
heur 1, 2, 4
set 1' +
heur 1, 2, 4, 5
a
b
c
d
e
f
g
h
i
90%
99%
99.9%
pruning factor
TurboDiff
BinDiff

25 Years of Software Obfuscation – Can It Keep Pace with Progress in Code Analysis? (Schrittwieser et al, 2013)

Code analysis categories	Example
Pattern matching	Malware signatures
Automated static analysis	Heuristic malware detection
Automated dynamic analysis	Malware analysis in the labs of anti-virus vendors
Human-assisted analysis	Reverse engineering

Attacker's aims	Example
Finding the location of data (LD)	Extraction of licensing keys from binary
Finding the location of program functionality in the code (LC)	Finding the location of a copy protection mechanism
Extraction of code fragments (EC)	Extraction of code fragments for rebuilding verification routines for licensing keys
Understanding the program (UC)	Understand a proprietary cipher in order to start cryptanalysis attempts

25 Years of Software Obfuscation – Can It Keep Pace with Progress in Code Analysis?

(Schrittwieser et al, 2013)

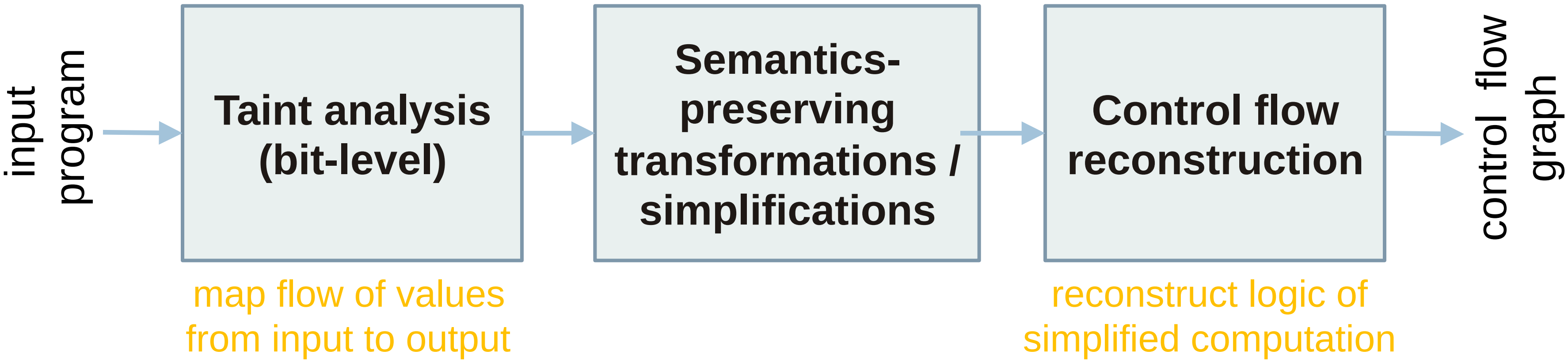
	Patterns		Automated static				Automated dynamic				Human assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Data obfuscation														
Reordering data														
Changing encodings														
Converting static data to procedures														
Static code rewriting														
Replacing instructions														
Opaque predicates														
Inserting dead code														
Inserting irrelevant code														
Reordering														
Loop transformations														
Method splitting/recombination														
Aliasing														
Control flow flattening														
Parallelized code														
Name scrambling														
Removing standard library calls														
Breaking relations														
Dynamic code rewriting														
Packing/Encryption														
Dynamic code modifications														
Environmental requirements														
Hardware-assisted code obfuscation														
Virtualization														
Anti-debugging techniques														

Discussion

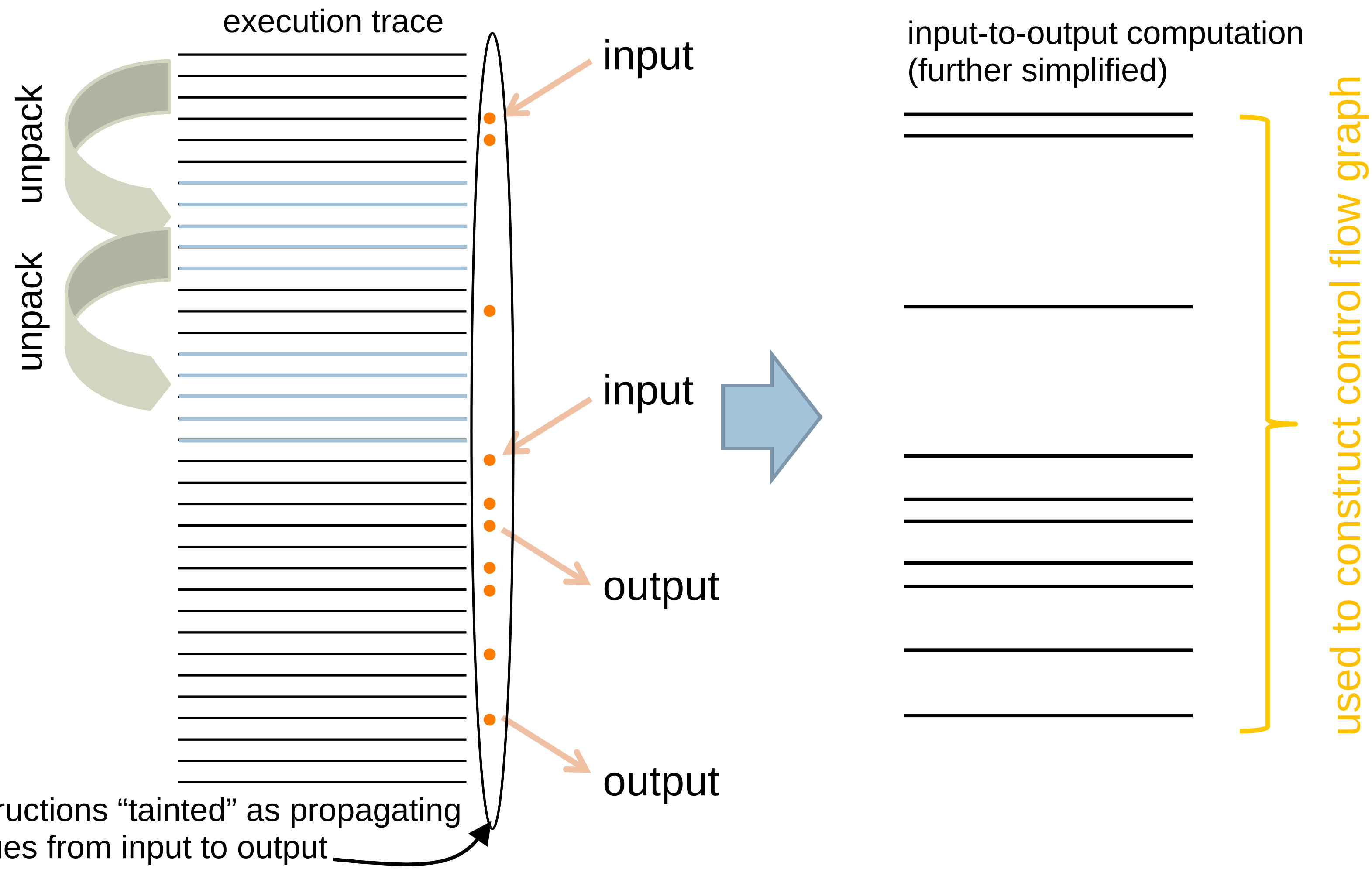
- What program fragments matter?
- What representation to use?
  - Sound vs unsound
  - Static vs. dynamic vs. hybrid
- Depends on level of expertise, application, type of assets, threat on the asset, attack step

Reverse-engineering obfuscated programs (Debray et al, 2014)

- 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



Trace simplification

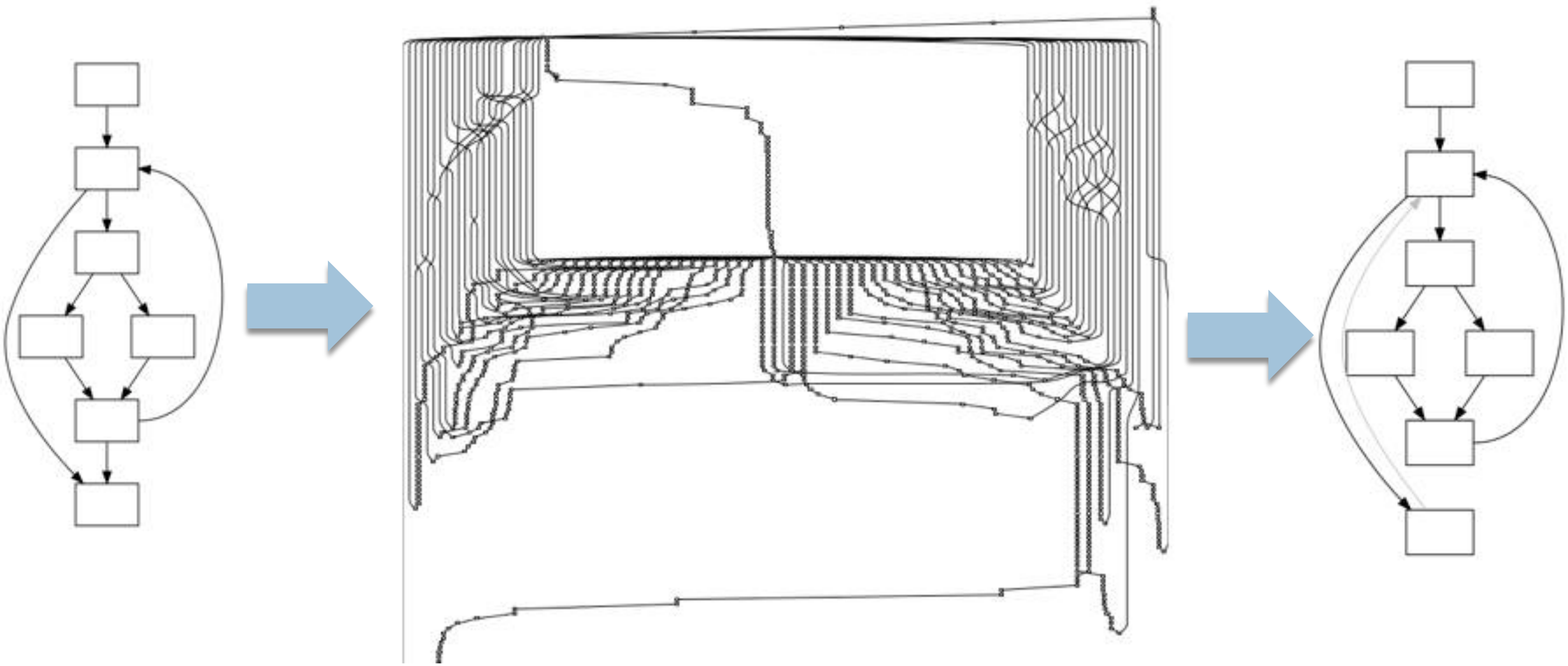


"Semantic-preserving" simplification

- *Quasi-invariant locations*: locations that have the same value at each use.
- Their transformations (currently):
  - 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.

Example: Themida Emulation Obfuscation

original | obfuscated (cropped) | deobfuscated



Discussion

- What program fragments matter?
- What representation to use?
  - Sound vs unsound
  - Static vs. dynamic vs. hybrid
- Depends on level of expertise, application, type of assets, threat on the asset, attack step

Overview

- ASPIRE in a nutshell
- Modelling attacks
- Evaluation Criteria
  - Metrics of complexity
  - Resilience
- Theory versus practice: involving the humans

Experiments with Human Subjects

- What is the real protection provided?
  - For identification/engineering
  - For exploitation
- Which protection is better?
- Against which type of attacker?
- How fast do subjects learn to attack protections?
- Which attack methods are more likely to be used?
- Which attack methods are more likely to succeed?

Experiments with Human Subjects

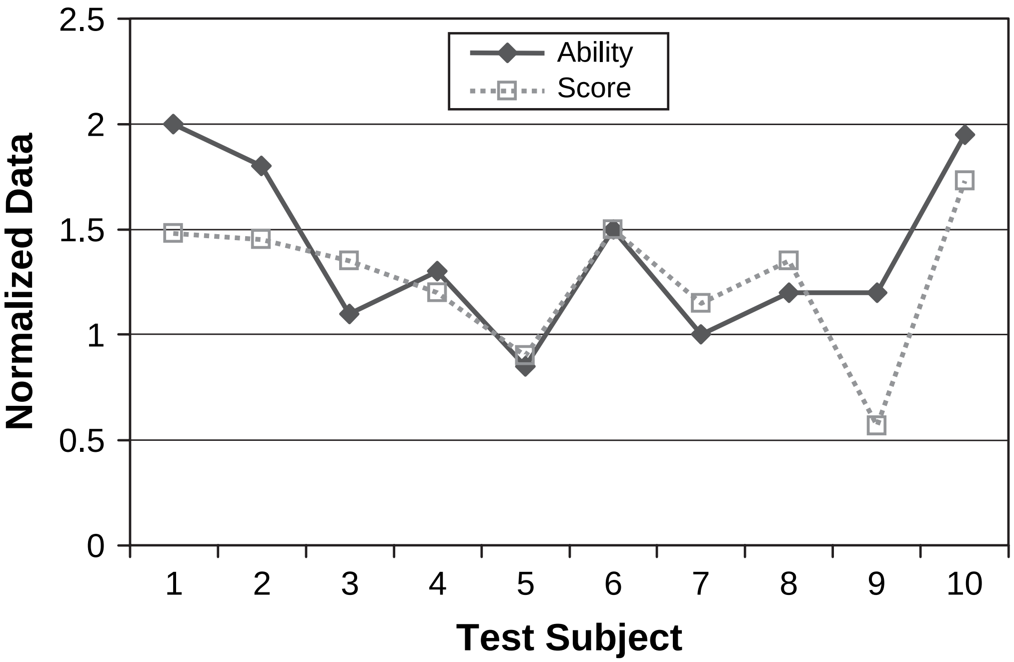
- Very hard to set up and get right
  - with students: cheap but representative?
  - with experts: expensive, but controlled?
  - what to test? (Dunsmore & Roper, 2000)
    - maintenance
    - recall
    - subjective rating
    - fill in the blank
    - mental simulation
  - How to extrapolate

How not to do it (Sutherland, 2006)

Table 1 Reverse engineering experiment framework

Session	Event	Test object	Program function	Task	Duration (min)	Total duration (min)
Morning session	Initial assessment					
	Program Set A (debug option enabled)	1	Hello World	Static	15	35
				Dynamic	10	
				Modify	10	
		2	Date	Static	10	30
				Dynamic	10	
				Modify	10	
		3	Bubble Sort	Static	15	45
				Dynamic	15	
				Modify	15	
		4	Prime Number	Static	15	45
				Dynamic	15	
				Modify	15	
Lunch						
Afternoon session	Program Set B (debug option disabled)	5	Hello World	Static	10	30
				Dynamic	10	
				Modify	10	
		6	Date	Static	10	30
				Dynamic	10	
				Modify	10	
		7	GCD	Static	15	45
				Dynamic	15	
				Modify	15	
		8	LIBC	Static	15	45
				Dynamic	15	
				Modify	15	
	Exit questionnaire					

How not to do it (Sutherland, 2006)



How not to do it (Sutherland, 2006)

Table 4 Source code metrics debug disabled

Source program	Hello World	Date	GCD	LIBC	Correlation
Test object	5	6	7	8	
Mean grade per test object	1.350	1.558	1.700	1.008	
Metric					
Lines of code	6	10	49	665	–0.3821
Software length[a]	7	27	40	59	–0.3922
Software vocabulary[a]	6	14	20	21	–0.0904
Software volume[a]	18	103	178	275	–0.4189
Software level[a]	0.667	0.167	0.131	0.134	–0.1045
Software difficulty[a]	1.499	5.988	7.633	7.462	0.0567
Effort[a]	27	618	2346	5035	–0.5952
Intelligence[a]	12	17	17	19	–0.1935
Software time[a]	0.001	0.001	0.2	0.4	–0.5755
Language level[a]	8	2.86	2.43	2.3	–0.0743
Cyclomatic complexity	1	1	3	11	–0.7844

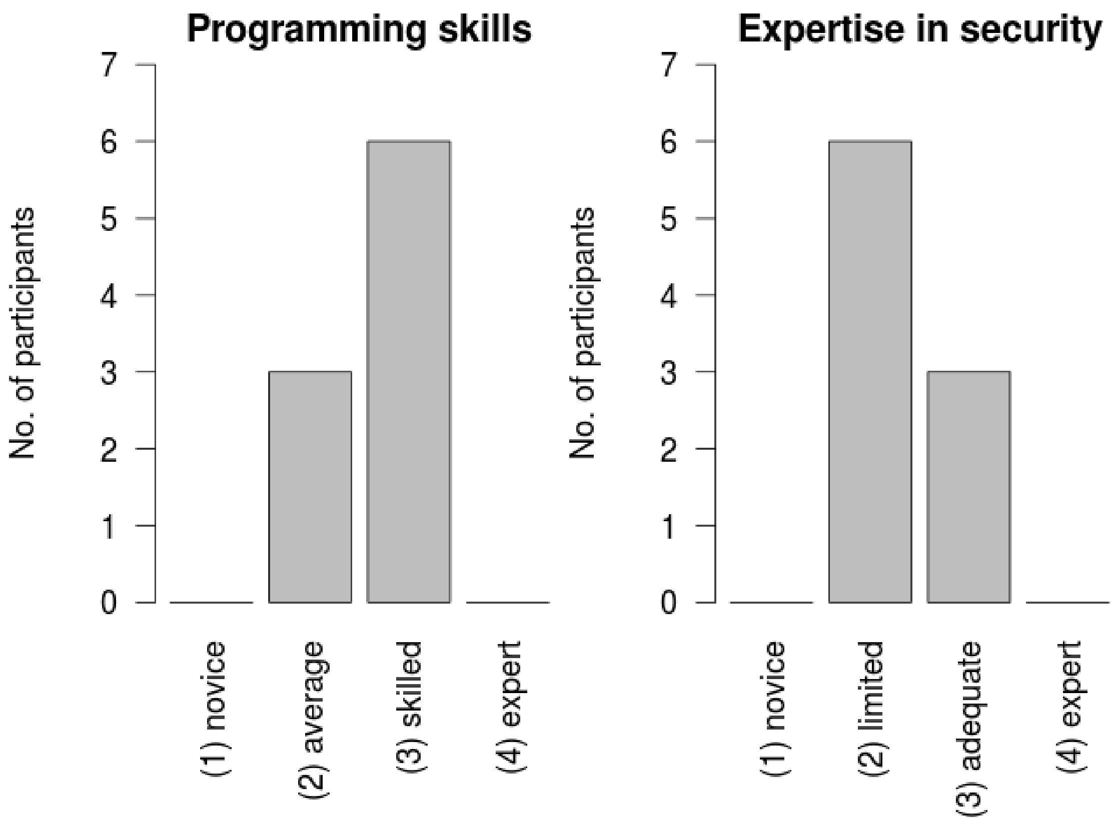
[a] Halstead metrics.

How to do it?

(Tonella et al, 2007; Ceccato et al, 2014; Scandariato et al, 2013)

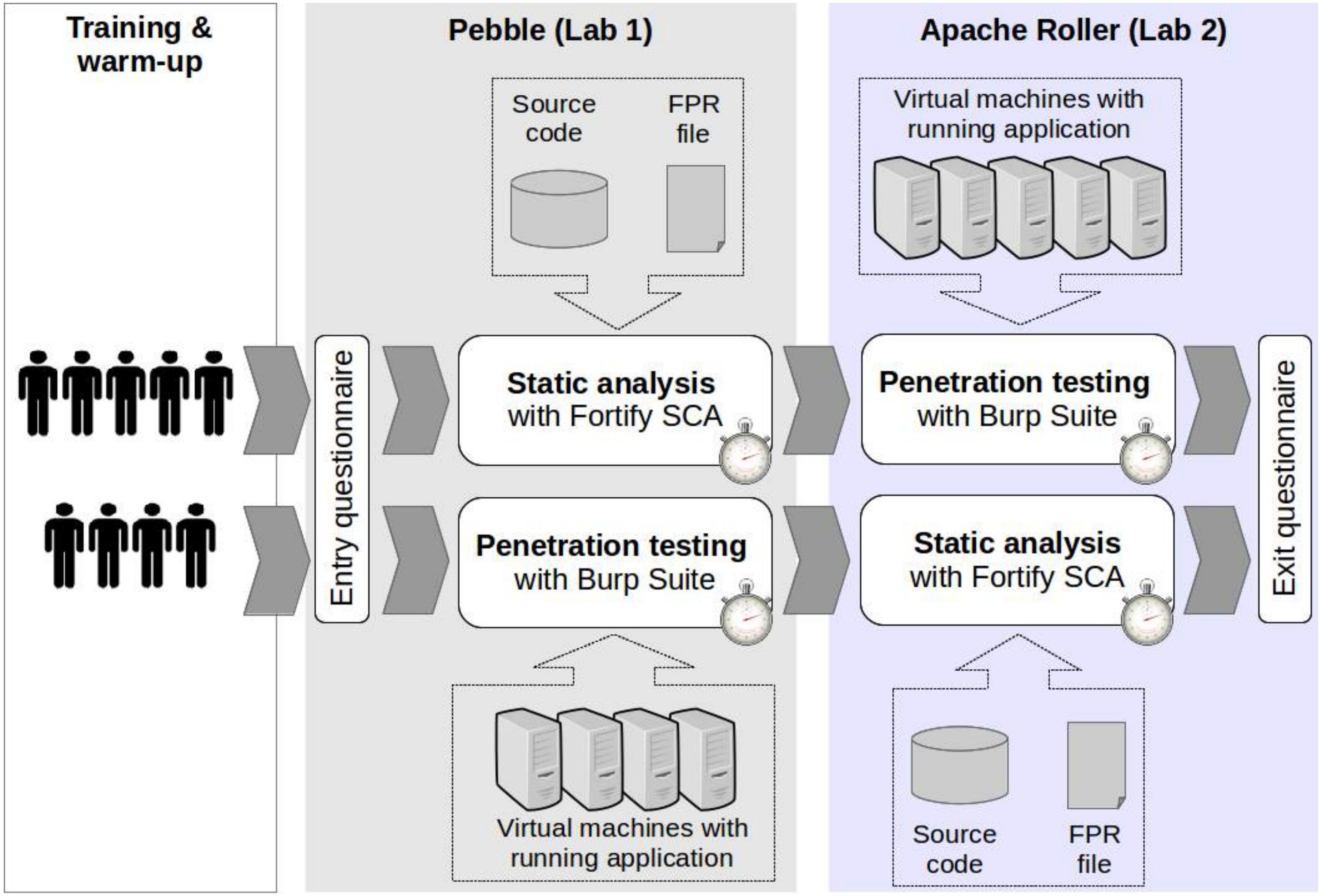
Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Subjects described in detail



Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Training and experiment described in detail



Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Rigorous statistical analysis of the results

	Measure	*Definition*	*Formula*	*Wish*
TP	True positive	An actual vulnerability is correctly reported by the participant (a.k.a. correct result)		high
FP	False positive	A vulnerability is reported by the participant but it is not present in the code (a.k.a. error, incorrect result, false alarm)		low
TOT	Reported vulnerabilities	The total number of vulnerabilities reported by the participant	TP + FP	–
TIME	Time	The time (in hours) that it takes the participant to complete the task		low
PREC	Precision	Percentage of the reported vulnerabilities that are correct	TP / TOT	high
PROD	Productivity	Number of correct results produced in a unit of time	TP / TIME	high

$$H_0^{TP} : \mu\{TP_{SA}\} = \mu\{TP_{PT}\}$$

Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Rigorous statistical analysis of the results

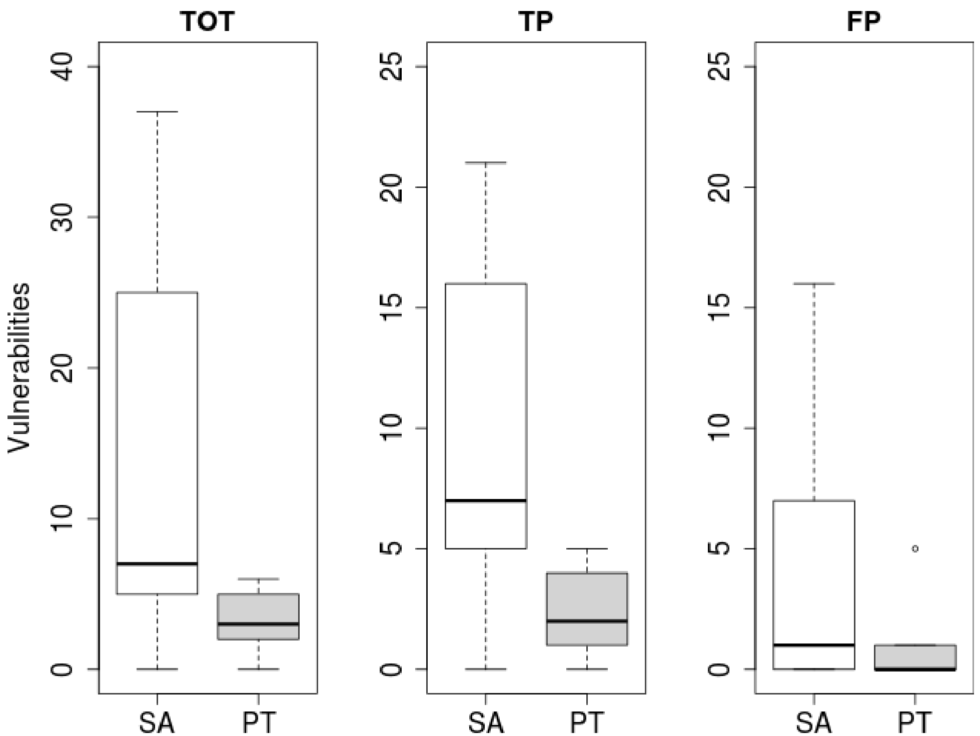


Fig. 5. Boxplot of reported results (TOT), correct results (TP) and false alarms (FP)

Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Rigorous statistical analysis of the results

In order to enable the replication of this study, all the data used in this paper is available online [11]. The data analysis is performed with R. Given the limited sample size, the analysis presented in this section makes use of non parametric tests. In particular, the location shifts between the two treatments are tested by means of the Wilcoxon signed-rank test for paired samples. The same test is used to analyze the exit questionnaire. A significance level of 0.05 is always used. The 95% confidence intervals are computed by means of the one-sample Wilcoxon rank-sum test. The association between two variables is studied by means of the Spearman rank correlation coefficient. A correlation is considered only if the modulus of the coefficient is at least 0.70 and the p-value of the significance test is smaller than 0.05.

We can reject the null hypothesis H_0^{TP} and conclude that static analysis produces, on average, a higher number of correct results than penetration testing.

Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Threats to validity discussed
  - conclusion validity
    - conclusions about the relationship among variables based on the data
  - internal validity
    - causal conclusion based on a study is warranted
  - external validity
    - generalized (causal) inferences
  - ...

Effectiveness & effeciency source code obfuscation
(Ceccato et al, 2014)

- Compare identifier renaming with opaque predicates
- All positive aspects seen before
- Much more extensive experiment
- And still they screw up somewhat ...

Clear code fragment chat program


```
public void addUserToList(String strRoomName, String strUser)
{
    RoomTabItem tab = getRoom(strRoomName);
    if(tab != null)
        tab.addUserToList(strUser);
}

public void removeUserFromList(String strRoomName, String strUser)
{
    RoomTabItem tab = getRoom(strRoomName);
    if(tab != null)
        tab.removeUserFromList(strUser);
}
```

Fragment with renamed identifiers


```
public void k(String s, String s1)
{
    h h1 = h(s);
    if(h1 != null)
        h1.k(s1);
}

public void l(String s, String s1)
{
    h h1 = h(s);
    if(h1 != null)
        h1.l(s1);
}
```

Fragment with opaque predicates


```
public void removeUserFromList(String strRoomName, String strUser) {
   RoomTabItem tab = null;
   if (Node.getI() != Node.getH()) {
      Node.getI().getLeft().swap(Node.getI().getRight());
      tab.transferFocusUpCycle();
   } else {
      Node.getF().swap(Node.getI());
      tab = getRoom(strRoomName);
   }
   if (Node.getI() != Node.getH()) {
      receiver.getClass().getAnnotations();
      Node.getH().getRight().swap(Node.getG().getLeft());
   } else {
      if (tab != null)
         if (Node.getI() != Node.getH()) {
            Node.getF().setLeft(Node.getG().getRight());
            roomList.clearSelection();
         } else {
            Node.getI().swap(Node.getH());
            tab.removeUserFromList(strUser);
         }
      Node.getI().getLeft().swap(Node.getF().getRight());
   }
}
```

References

Christophe Foket, Bjorn De Sutter, Koen De Bosschere. Pushing Java Type Obfuscation to the Limit. To appear in IEEE Trans. on Dependable en Secure Computing.

McCabe, T.J. A Complexity Measure. IEEE Transactions on Software Engineering, vol.SE-2, no.4, pp.308-320, Dec. 1976 doi: 10.1109/TSE.1976.233837

Roberto Giacobazzi and Andrea Toppan: On Entropy Measures for Code Obfuscation. Proceedings of the ACM SIGPLAN Software Security and Protection Workshop 2012

Jagdish Bansiya and Carl G. Davis. 2002. A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Trans. Softw. Eng. 28, 1 (January 2002), 4-17. DOI=10.1109/32.979986 http://dx.doi.org/10.1109/32.979986

Bart Coppens, Bjorn De Sutter, Koen De Bosschere: Protecting Your Software Updates. IEEE Security & Privacy 11(2): 47-54 (2013)

Bart Coppens, Bjorn De Sutter, Jonas Maebe: Feedback-driven binary code diversification. ACM TACO 9(4): 24 (2013)

Schrittwieser et al, 2013: 25 Years of Software Obfuscation – Can It Keep Pace with Progress in Code Analysis? SBA Research. http://www.sba-research.org/wp-content/uploads/2012/03/gesamte_Mappe_klein.pdf

Saumya K. Debray: Understanding software that doesn't want to be understood: Reverse engineering obfuscated binaries, Seminar 14241 - Challenges in Analysing Executables: Scalability, Self-Modifying Code and Synergy. http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=14241

Iain Sutherland, George E. Kalb, Andrew Blyth, Gaius Mulley: An empirical examination of the reverse engineering process for binary files. Computers & Security 25(3): 221-228 (2006)

Paolo Tonella, Marco Torchiano, Bart Du Bois, Tarja Systä: Empirical studies in reverse engineering: state of the art and future trends. Empirical Software Engineering 12(5): 551-571 (2007)

References

N. Wang, D. Fang, Y.X. Gu, Z. Tang, H. Wang. The Effectiveness Evaluation of Software Protection based on Attack Modeling. In Proceedings of ACM SIGPLAN Software Security and Protection Workshop, 2012, 8 pages.

Riccardo Scandariato, James Walden, Wouter Joosen: Static analysis versus penetration testing: A controlled experiment. ISSRE 2013: 451-460

Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, Paolo Tonella: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering 19(4): 1040-1074 (2014)

Woodward, M.R.; Hennel, M.A. and Hedley, D.(1979) "A Measure of Control Flow Complexityin Program Test", IEEE Trans. Software Eng., Vol. SE-5, No. 1, pp. 45 - 50

R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. I: The Goal Question Metric Approach Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. I, John Wiley & Sons, (1994)

Warren A. Harrison and Kenneth I. Magel. 1981. A complexity measure based on nesting level. SIGPLAN Not. 16, 3 (March 1981), 63-74.

Sallie M. Henry, Dennis G. Kafura: Software Structure Metrics Based on Information Flow. IEEE Trans. Software Eng. 7(5): 510-518 (1981)

Oviedo, Enrique I. · Control Flow, Data Flow, and Program Complexity, Proceedings of the Fourth International COMPSAC. 146-152. New York: IEEE Computer Society, October 1980.

Yingxu Wang, Jingqiu Shao: Measurement of the Cognitive Functional Complexity of Software. IEEE ICCI 2003: 67-74

John S. Davis: Chunks: A basis for complexity measurement. Inf. Process. Manage. 20(1-2): 119-127 (1984)

References

Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken-ichi Matsumoto, Yuichiro Kanzaki, Hirotsugu Satoh: Queue-Based Cost Evaluation of Mental Simulation Process in Program Comprehension. IEEE METRICS 2003: 351-

Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, Bart Preneel: Program obfuscation: a quantitative approach. QoP 2007: 15-20

Christian S. Collberg, Clark D. Thomborson, Douglas Low: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. POPL 1998: 184-196

C Collberg, C Thomborson, D Low: A taxonomy of obfuscating transformations. Technical Report, Department of Computer Science, The University of Auckland, New Zealand, 1997

Mila Dalla Preda, Roberto Giacobazzi: Semantics-based code obfuscation by abstract interpretation. Journal of Computer Security 17(6): 855-908 (2009)

Mila Dalla Preda, Matias Madou, Koen De Bosschere, Roberto Giacobazzi: Opaque Predicates Detection by Abstract Interpretation. AMAST 2006: 81-95

Cullen Linn, Saumya K. Debray: Obfuscation of executable code to improve resistance to static disassembly. ACM Conference on Computer and Communications Security 2003: 290-299

Halstead, Maurice H. (1977). Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7.

Benjapol Auprasert and Yachai Limpiyakorn: Underlying Cognitive Complexity Measure Computation with Combinatorial Rules. World Academy of Science, Engineering and Technology Vol:2 2008-09-29

Aspire Grant Agreement No 609734

The Aspire project has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 609734.

If you need further information, please contact the coordinator:
Bjorn De Sutter, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
Tel: +32 9 264 33 67 Fax: +32 9 264 35 94
Email: coordinator@aspire-fp7.eu
Website: http://www.aspire-fp7.eu