Software Protection Evaluation

Bjorn De Sutter

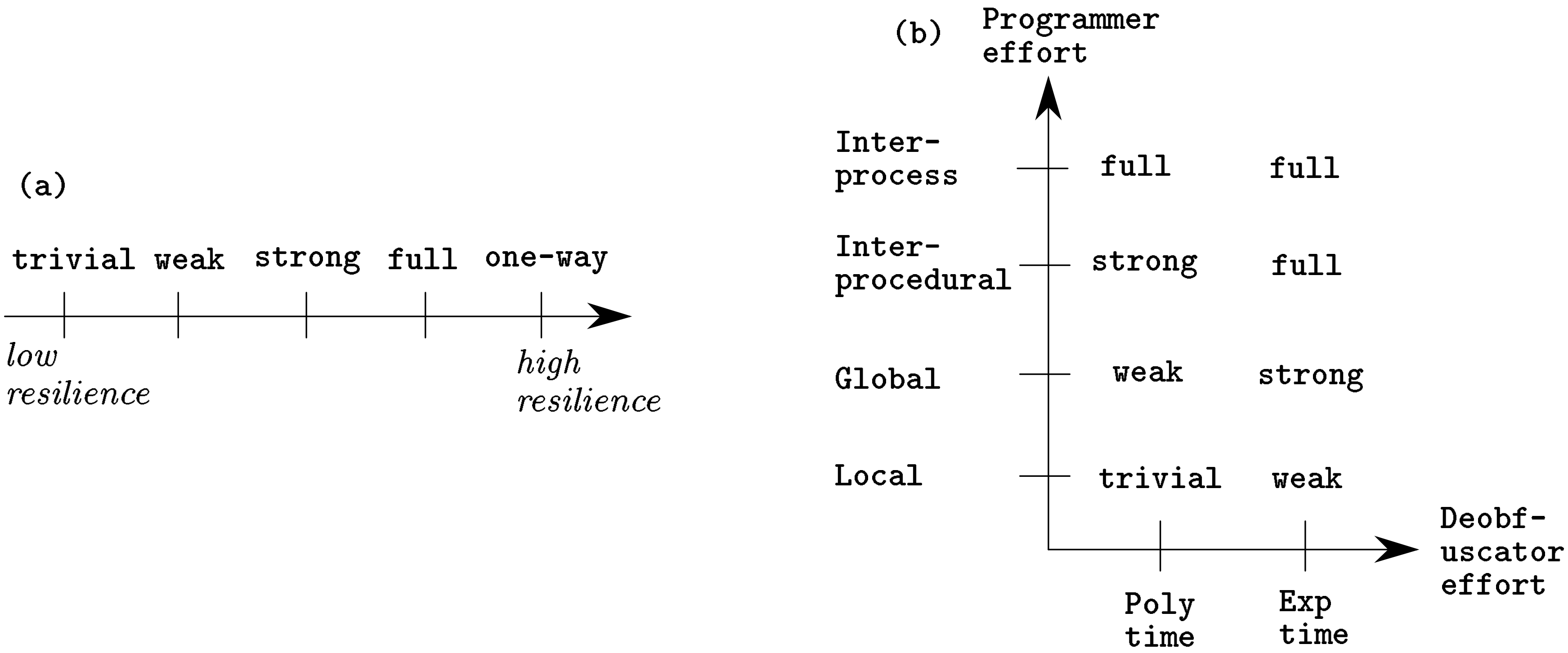
ISSISP 2017 – Paris

GHENT UNIVERSITY

Software Protection Evaluation

- Four criteria (Collberg et al)
  - **Potency**: confusion, complexity, manual effort
  - **Resilience**: resistance against (automated) tools
  - **Cost**: performance, code size
  - **Stealth**: identification of (components of) protections

Resilience (Collberg et al, 1997)



Software Protection Evaluation

- Four criteria (Collberg et al)

  of what? what task?

  - **Potency**: confusion, complexity, manual effort

  how computed? by who?

  existing and non-existing?

  - **Resilience**: resistance against (automated) tools

  operated by who? to achieve what?

  - **Cost**: performance, code size

  no other impacts on software-development life cycle?

  - **Stealth**: identification of (components of) protections

  where and when does this matter?

  which identification techniques?

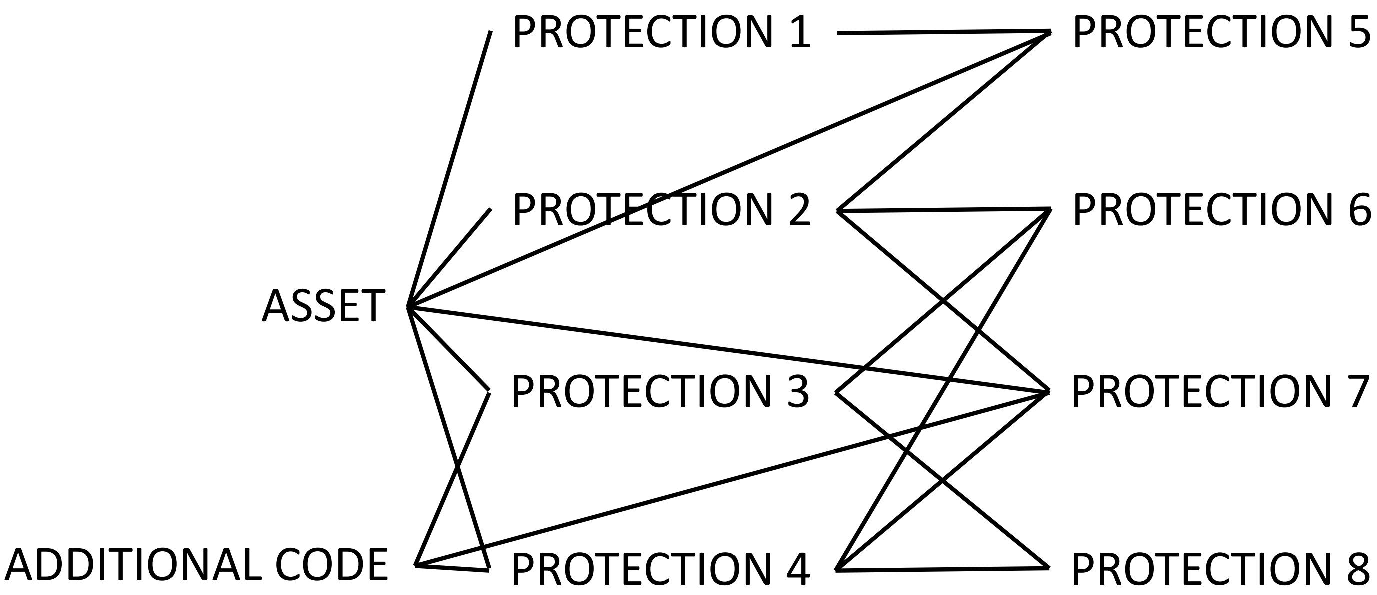
Lecture Overview

1. Protection vis-à-vis attacks
   - attacks on what?
   - attack and protection models
2. Qualitative Evaluation
3. Quantitative Evaluation
   - complexity metrics
   - tools
4. Human Experiments

What is being attacked?

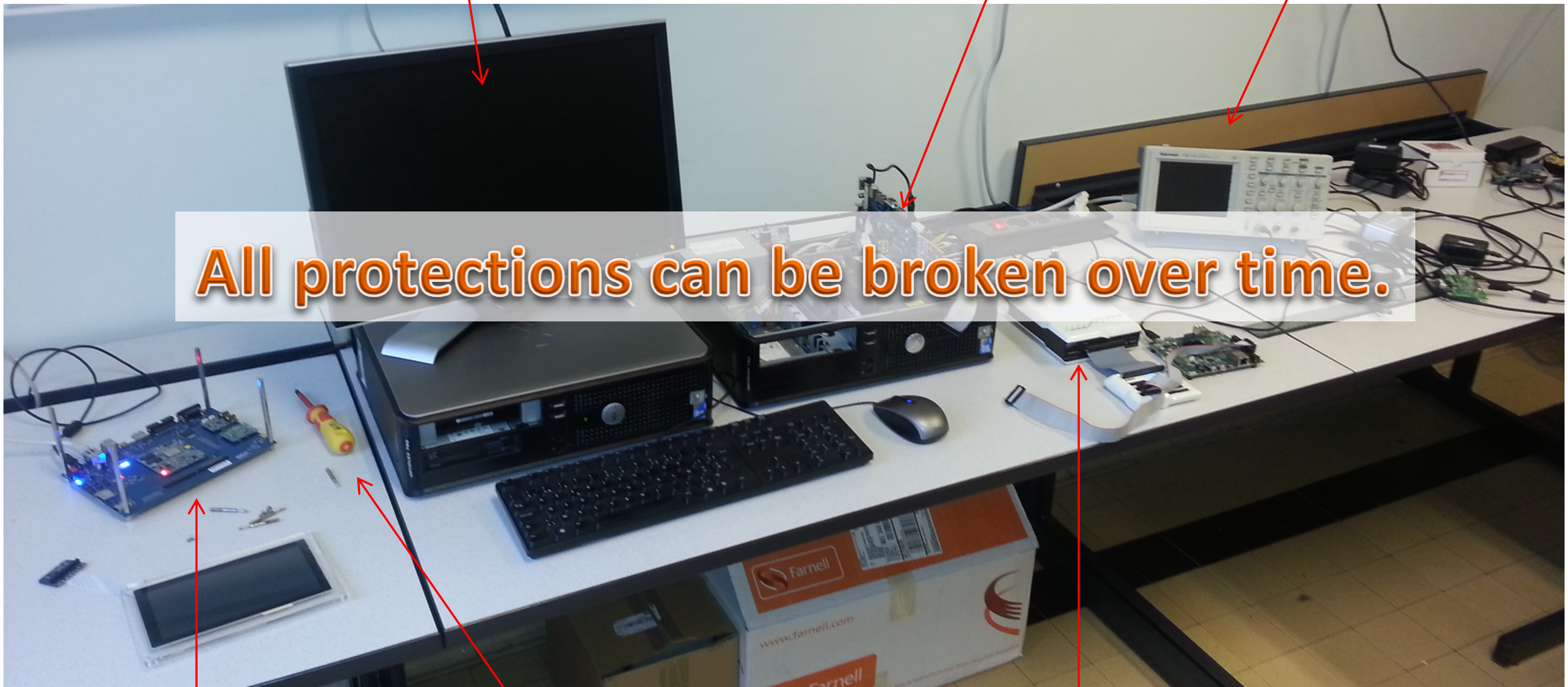
Asset category	Security Requirements	Examples of threats
Private data (keys, credentials, tokens, private info)	Confidentiality Privacy Integrity	Impersonation, illegitimate authorization Leaking sensitive data Forging licenses
Public data (keys, service info)	Integrity	Forging licenses
Unique data (tokens, 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?

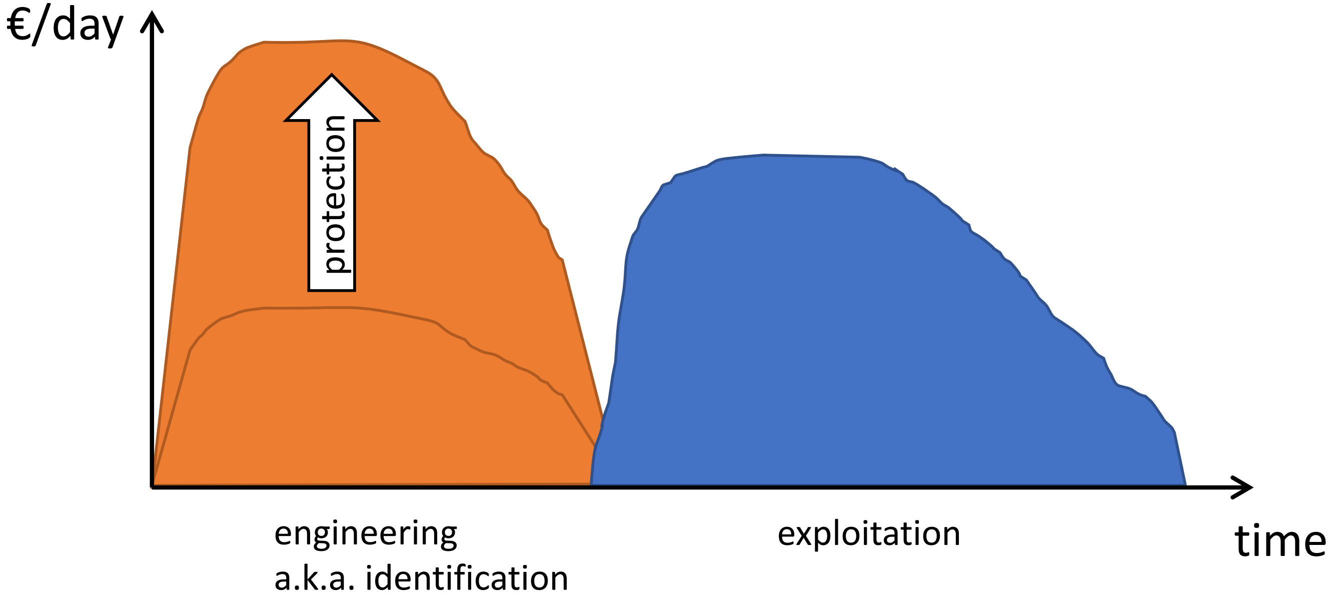


1. Attackers aim for assets, layered protections are only obstacles
2. Attackers need to find assets (by iteratively zooming in)
3. Attackers need tools & techniques to build a program representation, to analyze, and to extract features
4. Attackers iteratively build strategy based on experience and 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

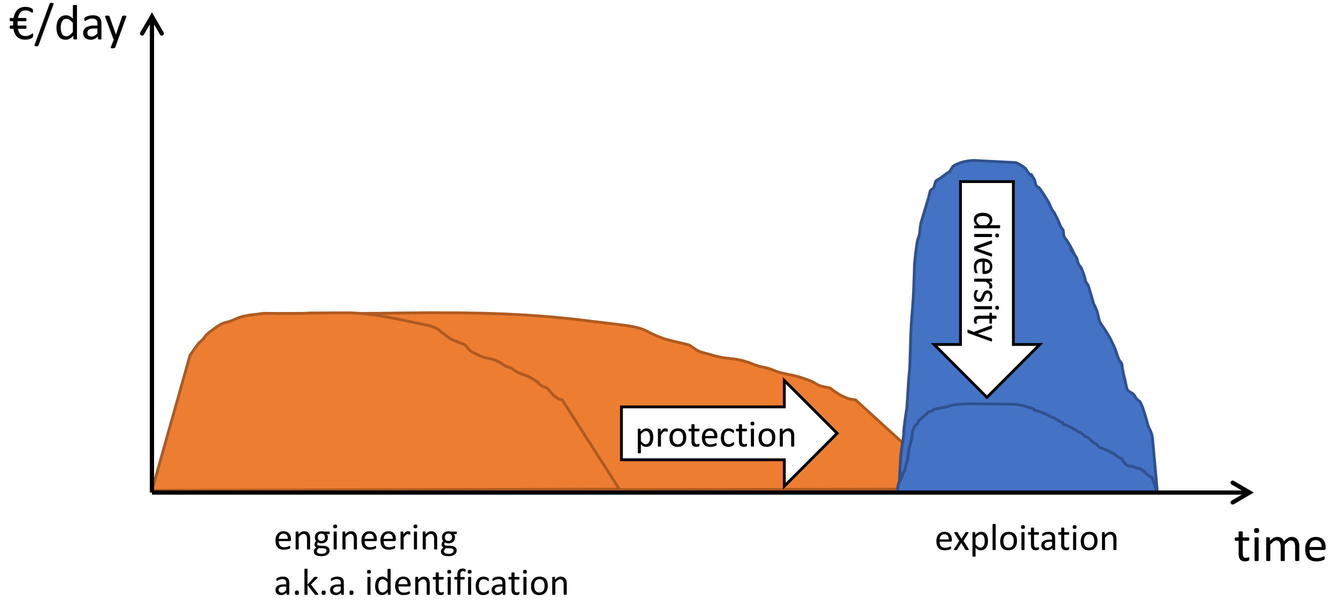
Protection againts MATE attacks



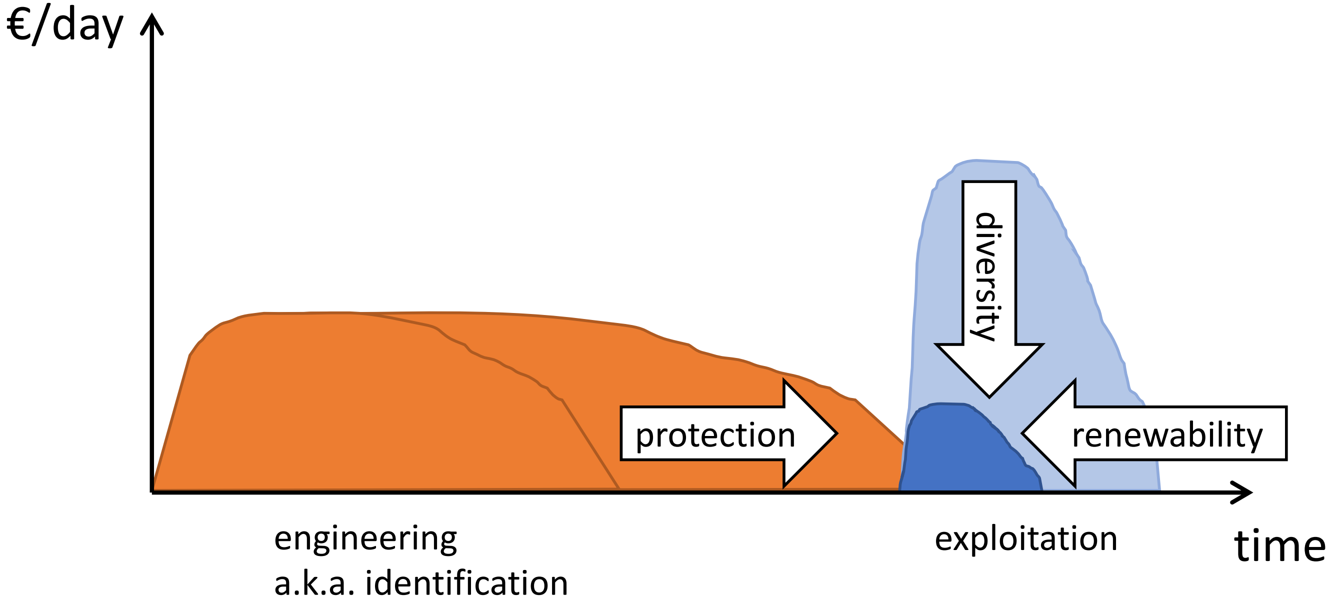
Economics of MATE attacks



Economics of MATE attacks

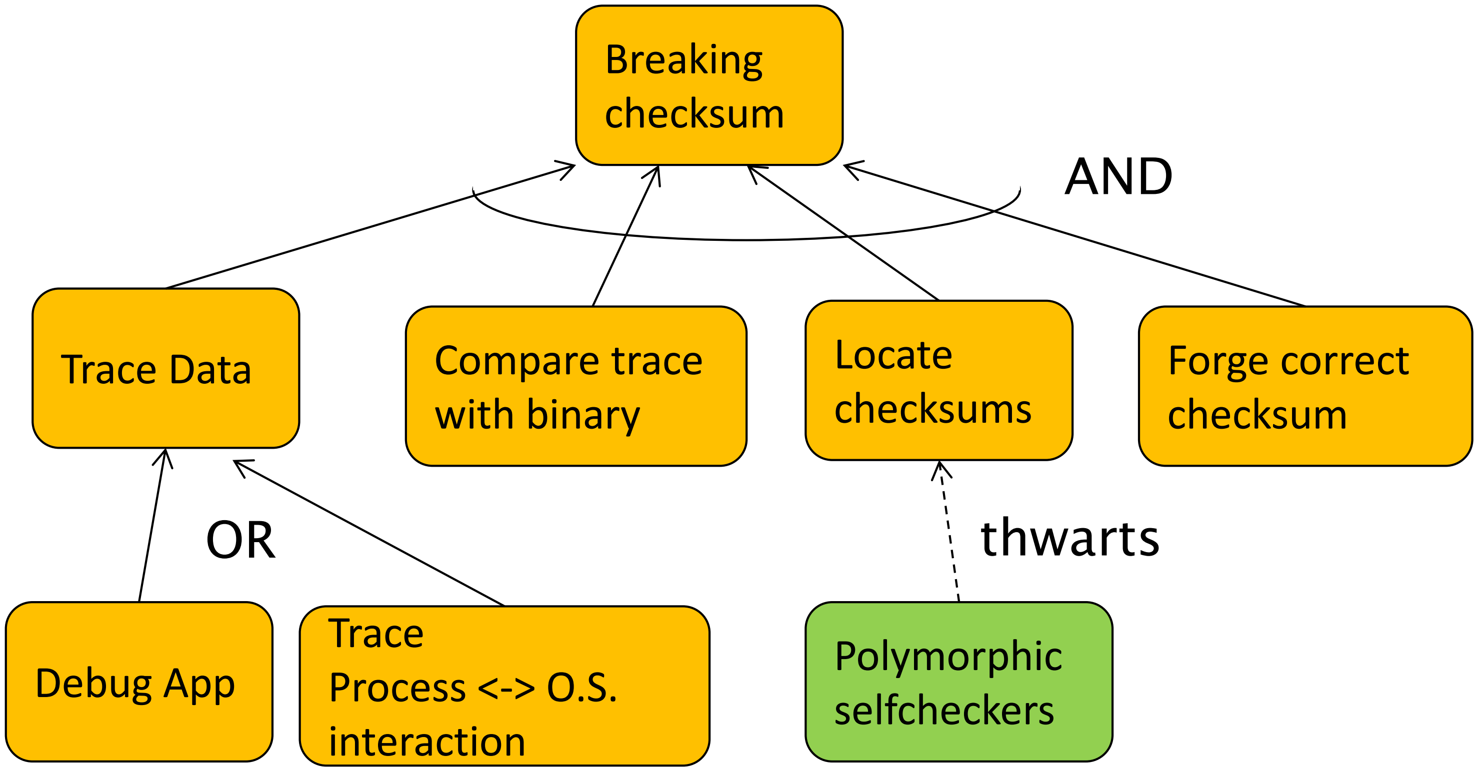


Economics of MATE attacks



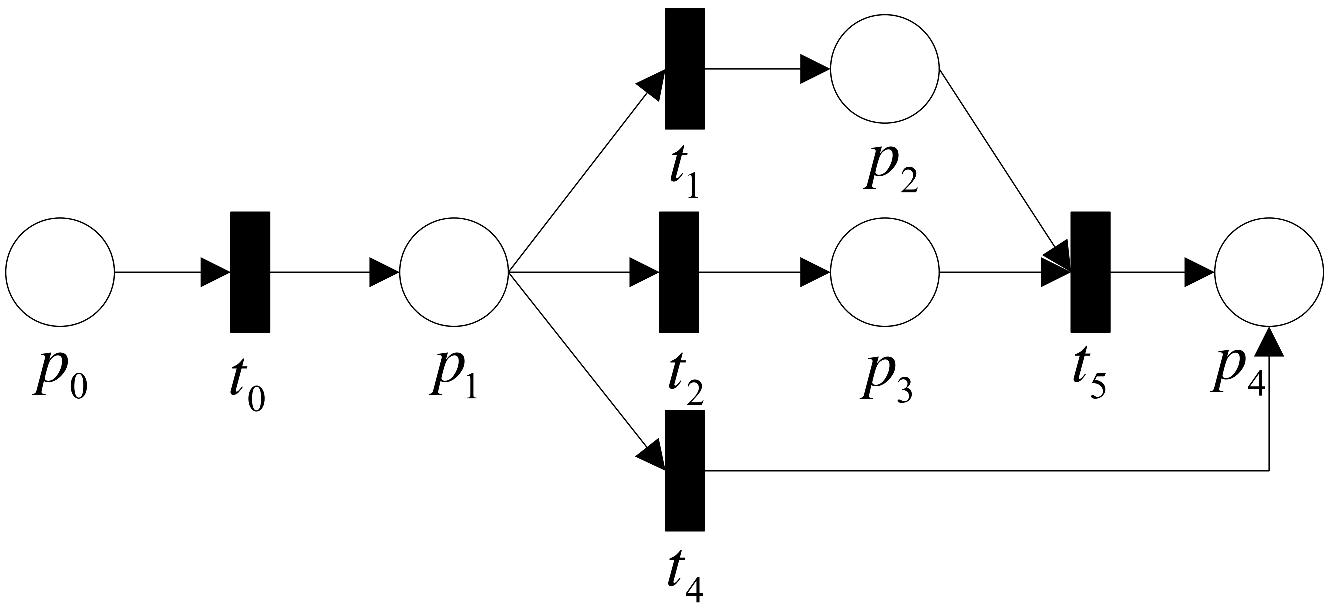
Attack Modelling: Attack Graphs (AND-OR Graphs)

- relate attack goal, subgoals, (and protections)



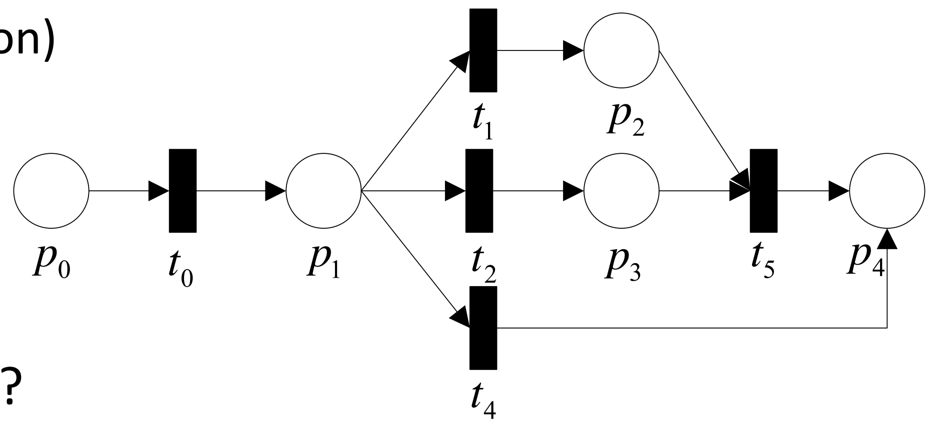
Attack Modelling: Petri Nets (Wang et al, 2012)

- 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



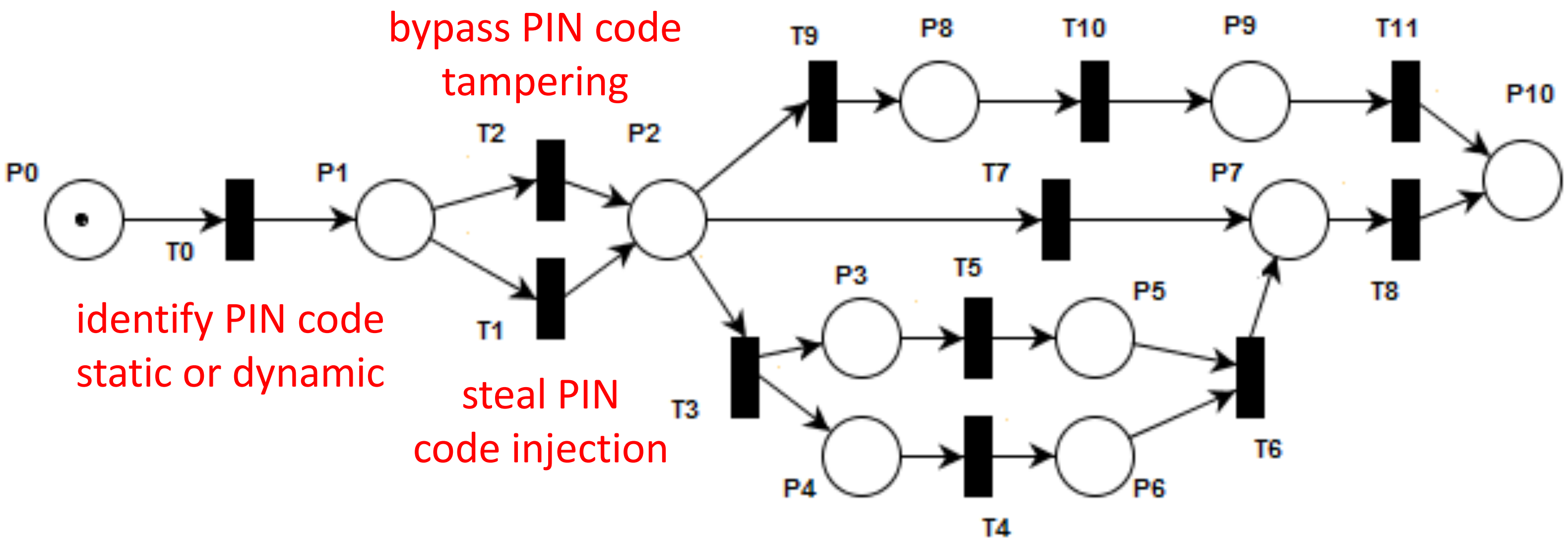
Attack Modelling: Petri Nets

- 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)



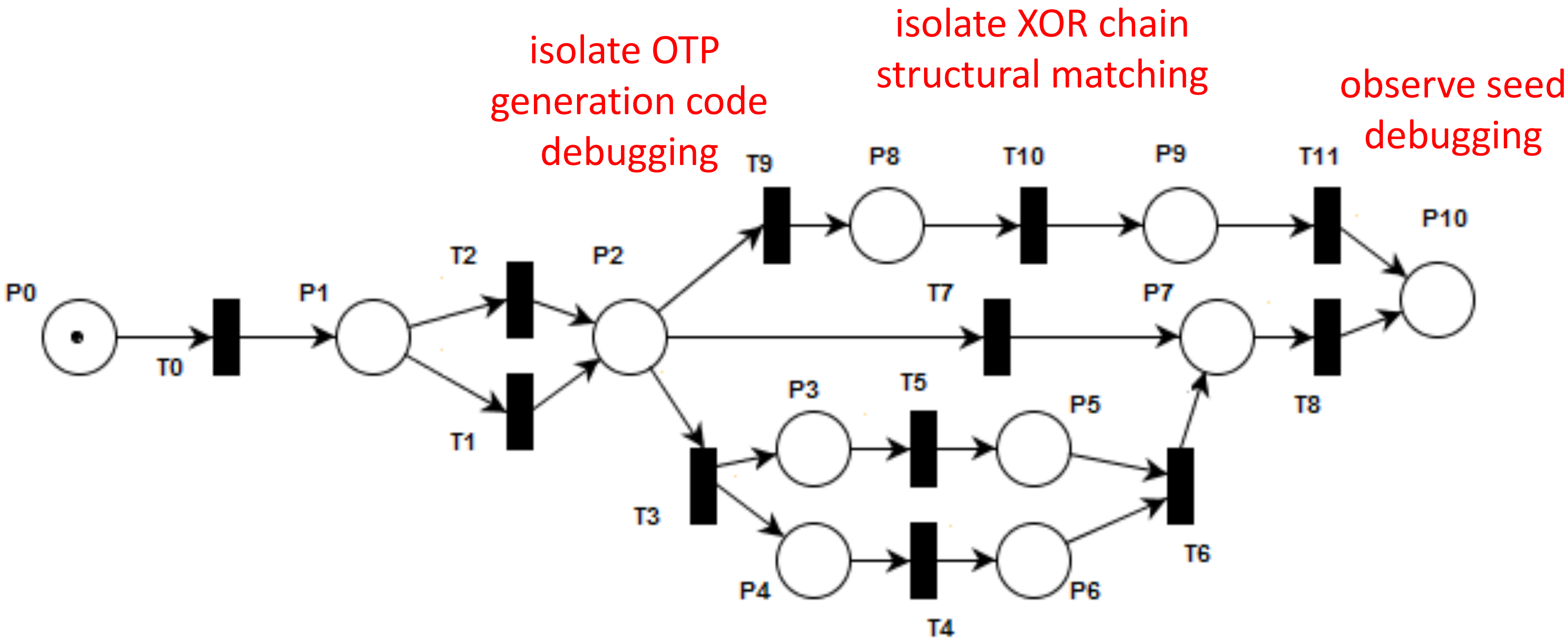
Example attack: One-Time Password Generator (P. Falcarin)

- Step 1: get working provisioning & OTP generation



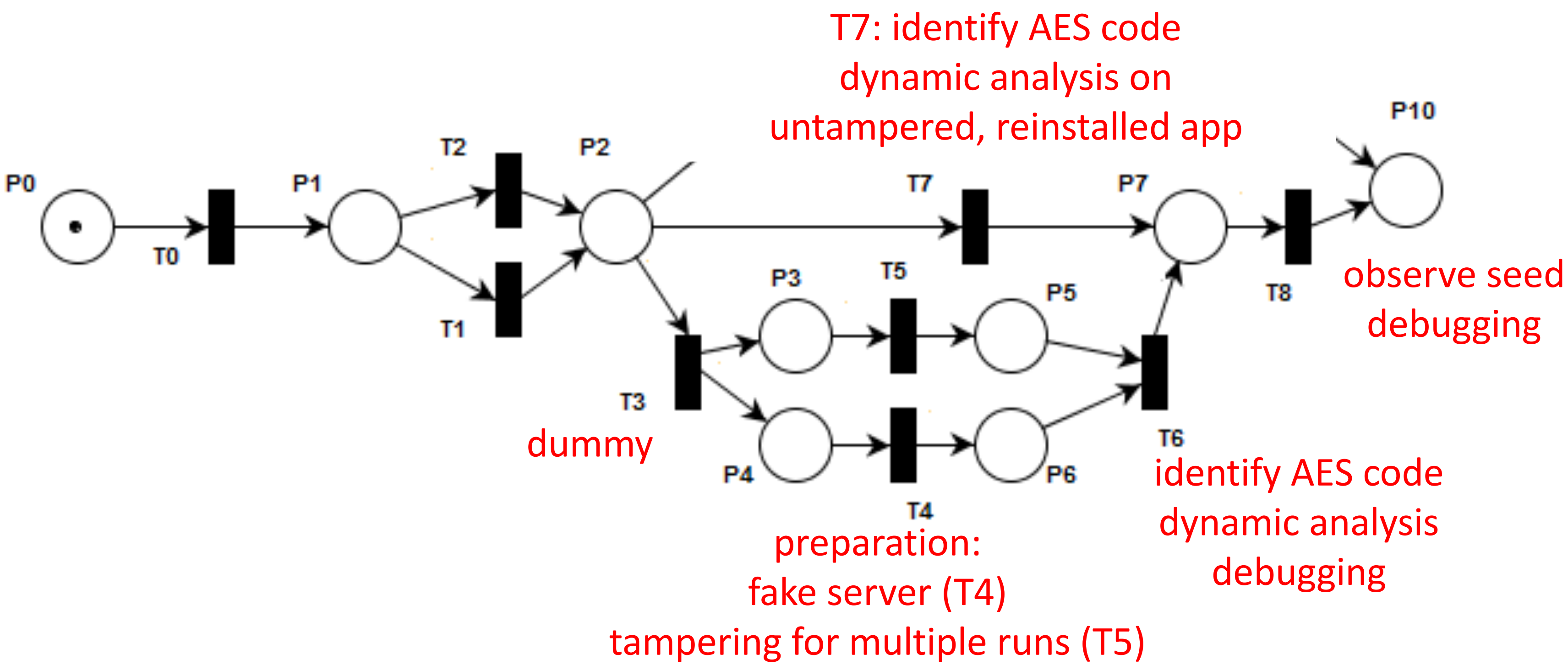
Example attack: One-Time Password generator (P. Falcarin)

- 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



Lecture Overview

1. Protection vis-à-vis attacks
   - attacks on what?
   - attack and protection models

2. Qualitative Evaluation

3. Quantitative Evaluation
   - complexity metrics
   - tools

4. Human Experiments

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

(Schrittwieser et al, 2013)

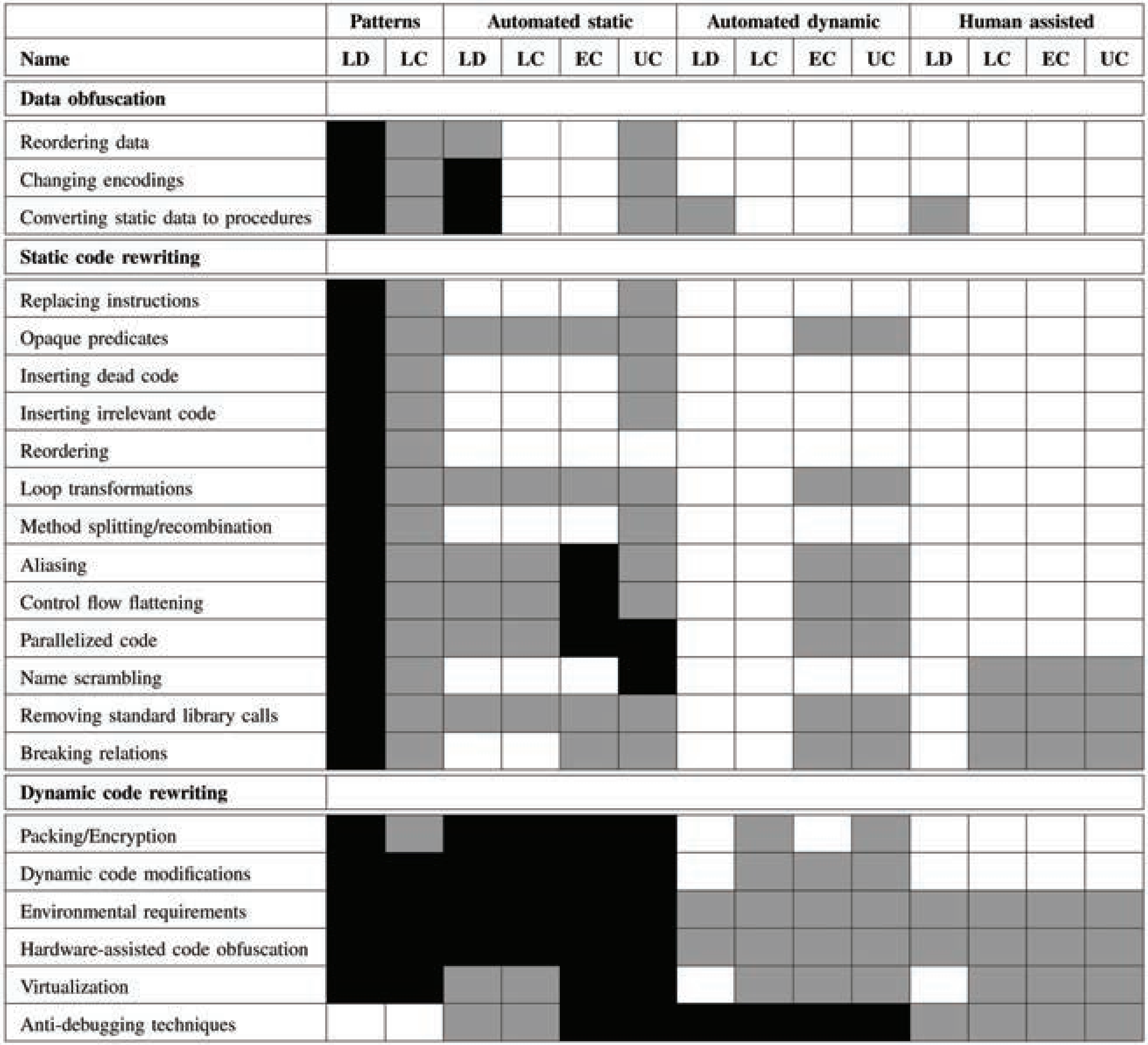
Code analysis categories	**Example**
Pattern matching	Malware signatures
Automated static analysis	Heuristic malware detection
Automated dynamic analysis	Malware analysis in the labs of anti-virus vendors
Human-assisted analysis	Reverse engineering

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

Car

(Sch


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

Lecture Overview

1. Protection vis-à-vis attacks
   - attacks on what?
   - attack and protection models

2. Qualitative Evaluation

3. Quantitative Evaluation
   - complexity metrics
   - tools

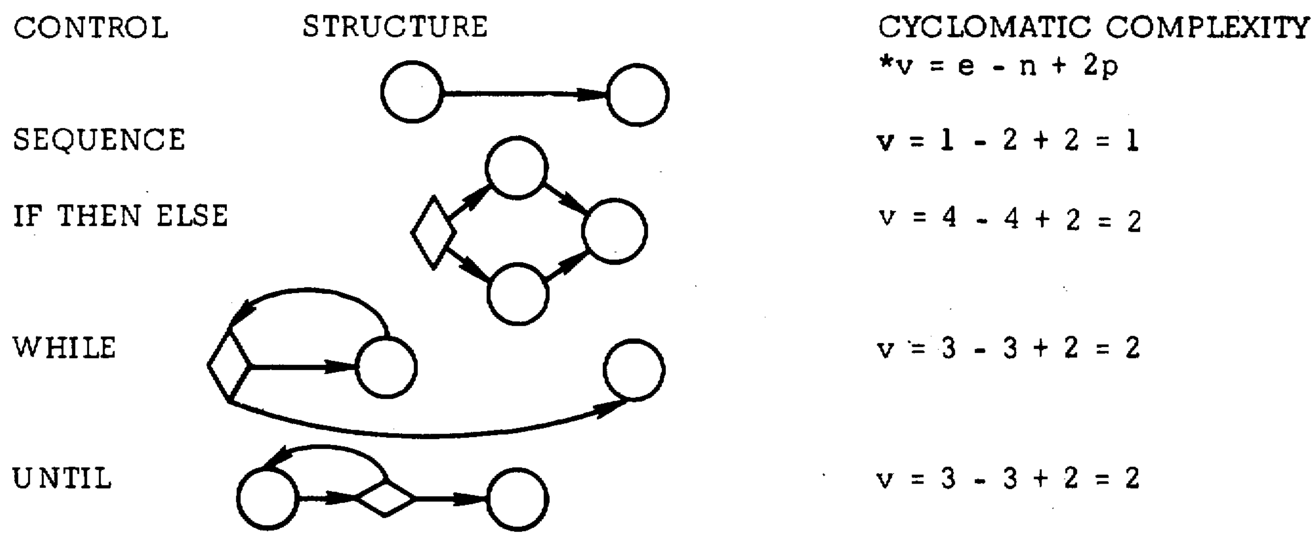
4. Human Experiments

Cyclomatic number (McCabe, 1976)

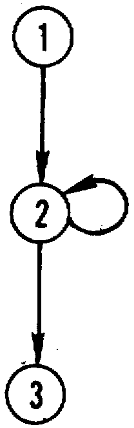
- control flow complexity

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

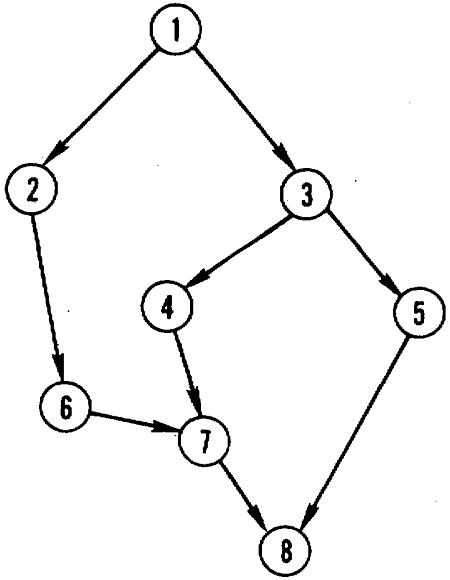
- single components: V(cfg) = #edges − #nodes + 2
- related to the number of linearly independent paths
- related to number of tests needed to invoke all paths



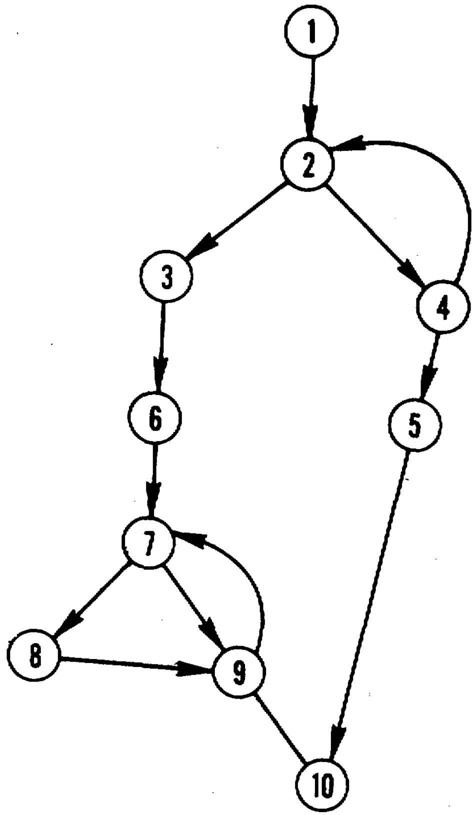
Cyclomatic number (McCabe, 1976)



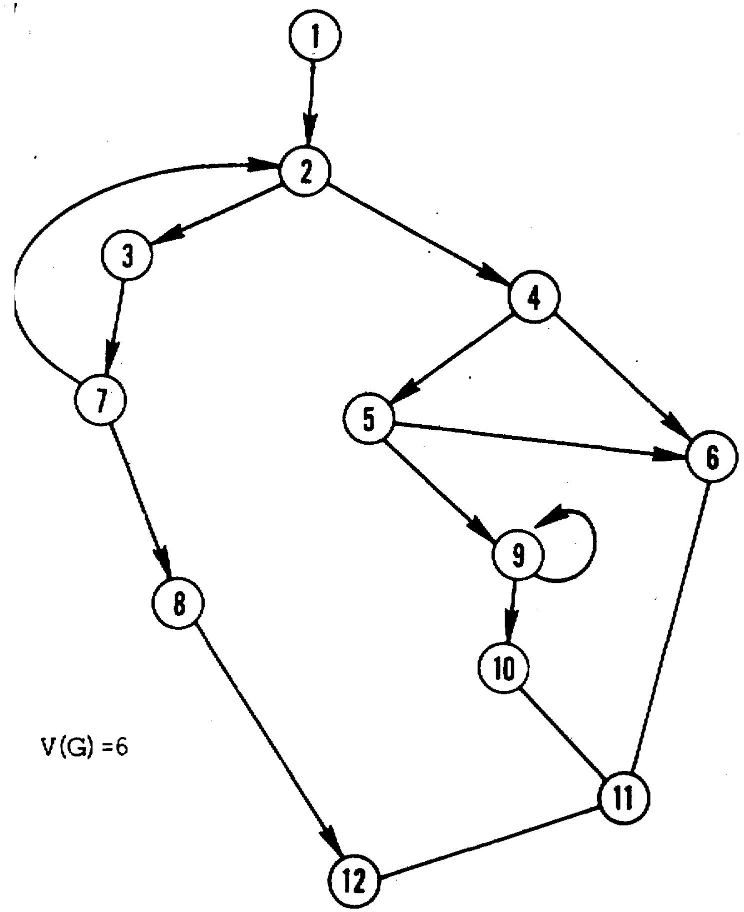
V(G) =2



V(G) =3

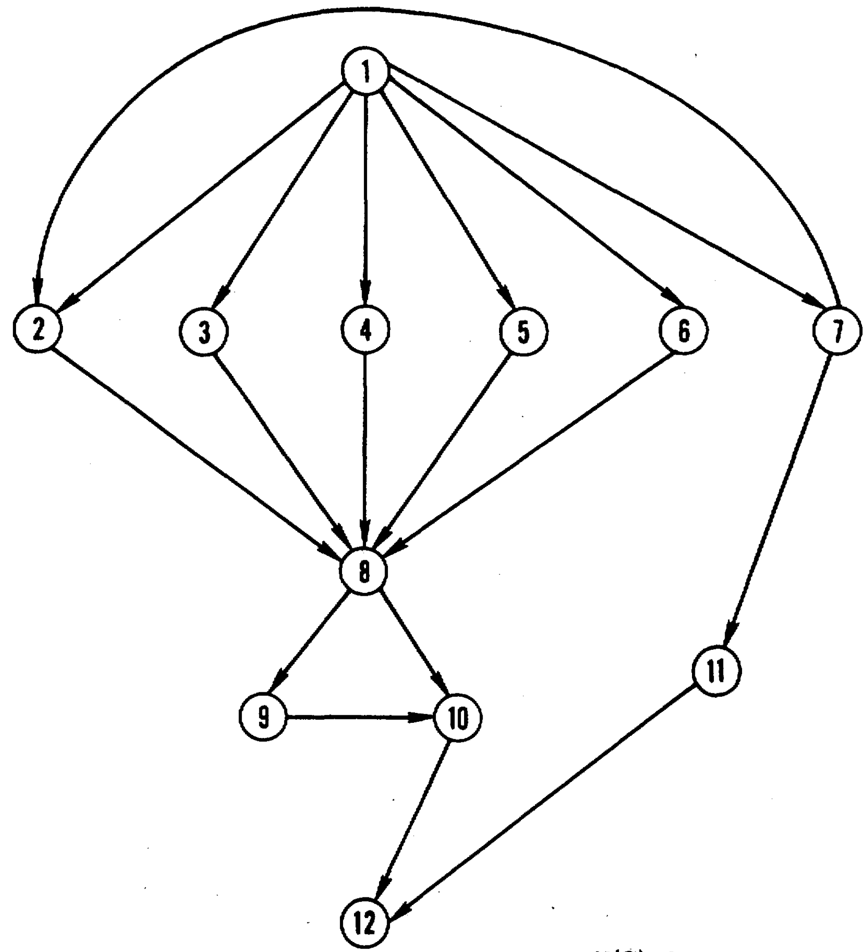


V(G) =5

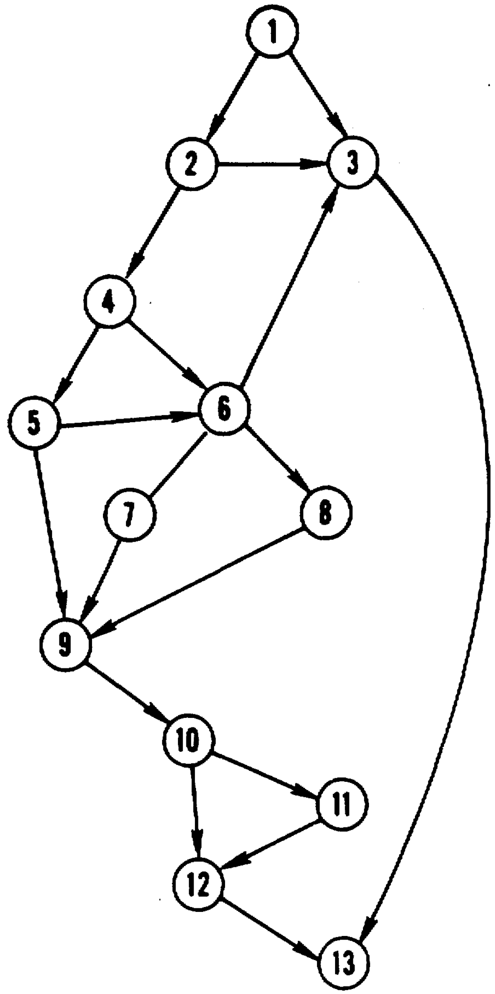


V(G) =6

Cyclomatic number (McCabe, 1976)



V(G) =8



V(G) =8

- 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

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
- So is the brain
- Measure misses with small sizes of memory

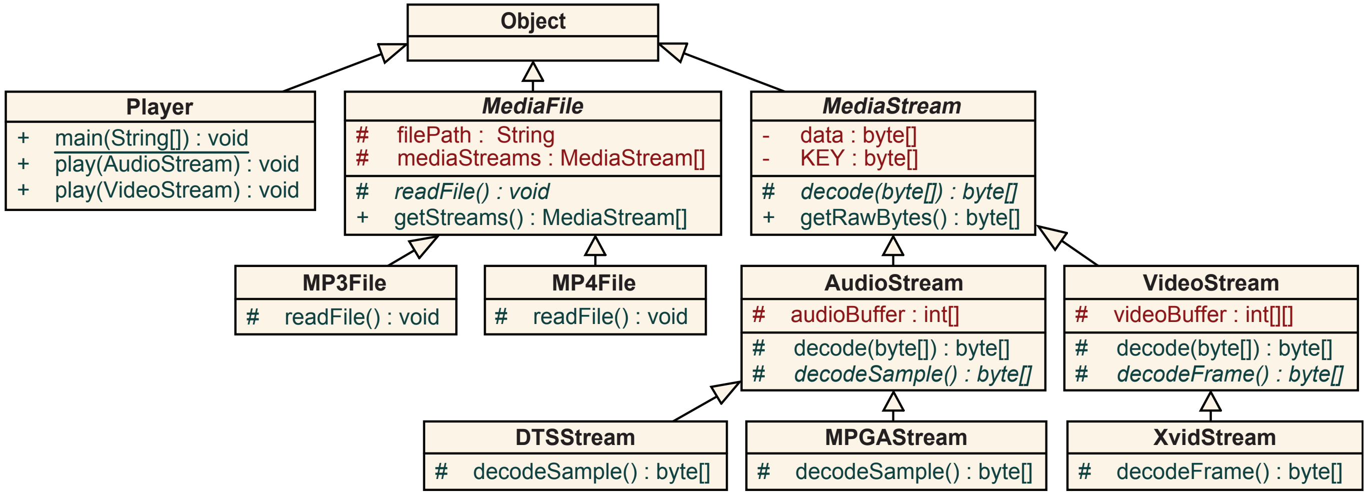
Combine all of them (Anckaert et al, 2007)

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
   - data structure complexities
4. data
   - application-specific

static -> graphs

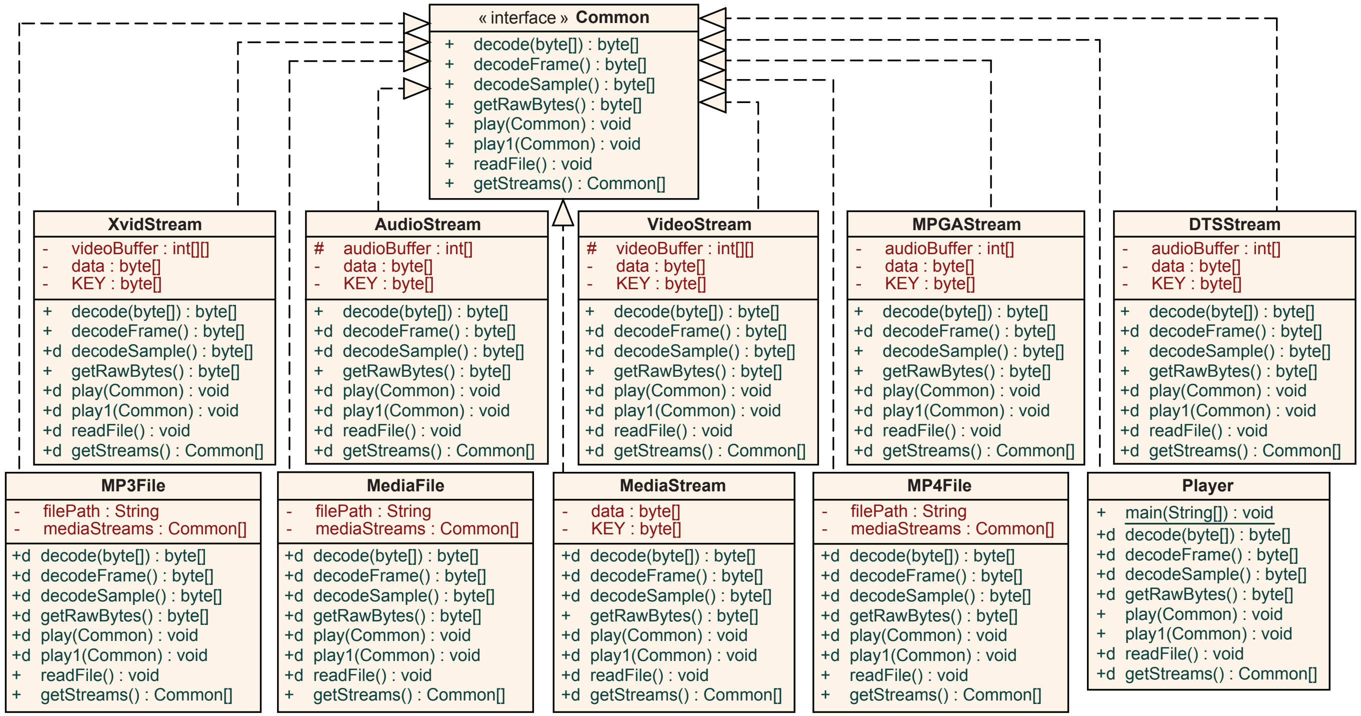
dynamic -> traces

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



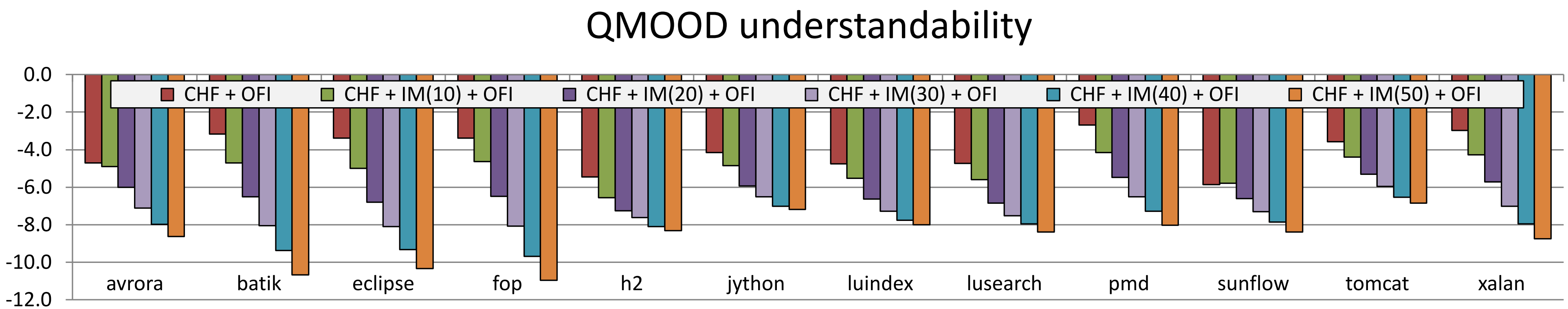

```
public class Player {
  public void play(AudioStream as) {
    /* send as.getRawBytes() to audio device */
  }
  public void play(VideoStream vs) {
    /* send vs.getRawBytes() to video device */
  }
  public static void main(String[] args) {
    Player player = new Player();
    MediaFile[] mediaFiles = ...;
    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)




```java
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){ ... }
}
```

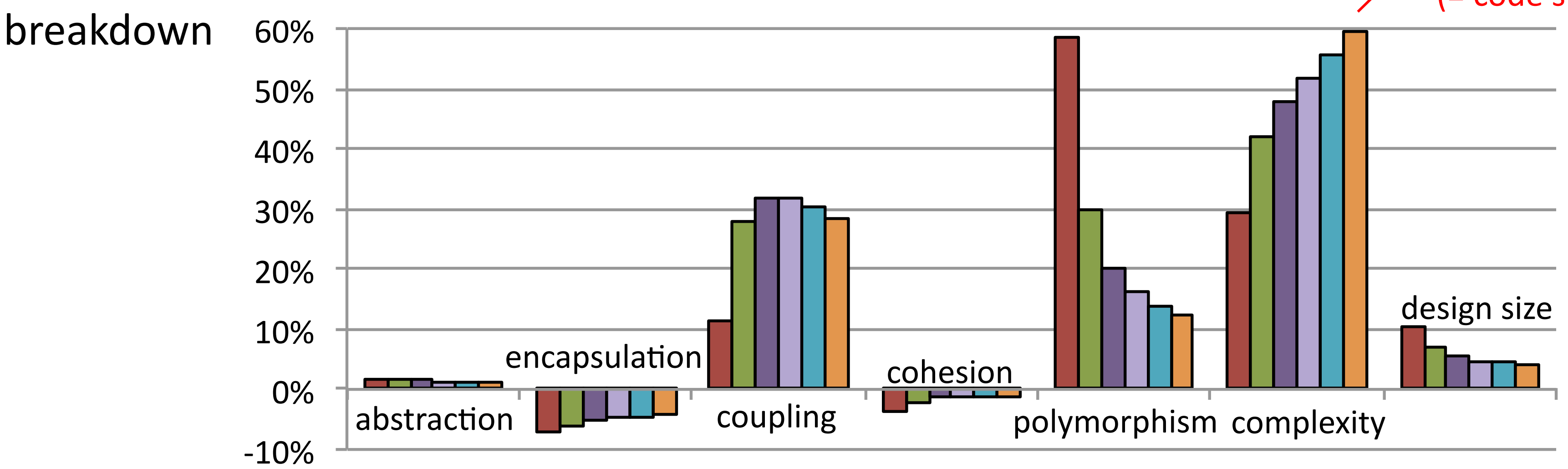
Object-Oriented Quality Metrics (Bansiya & Davis, 2002)



90% of classes transformed

25% of classes transformed

dominating term
(= code size)



Tool-based metrics:
Example 1: Disassembly Thwarting (Linn & Debray, 2003)

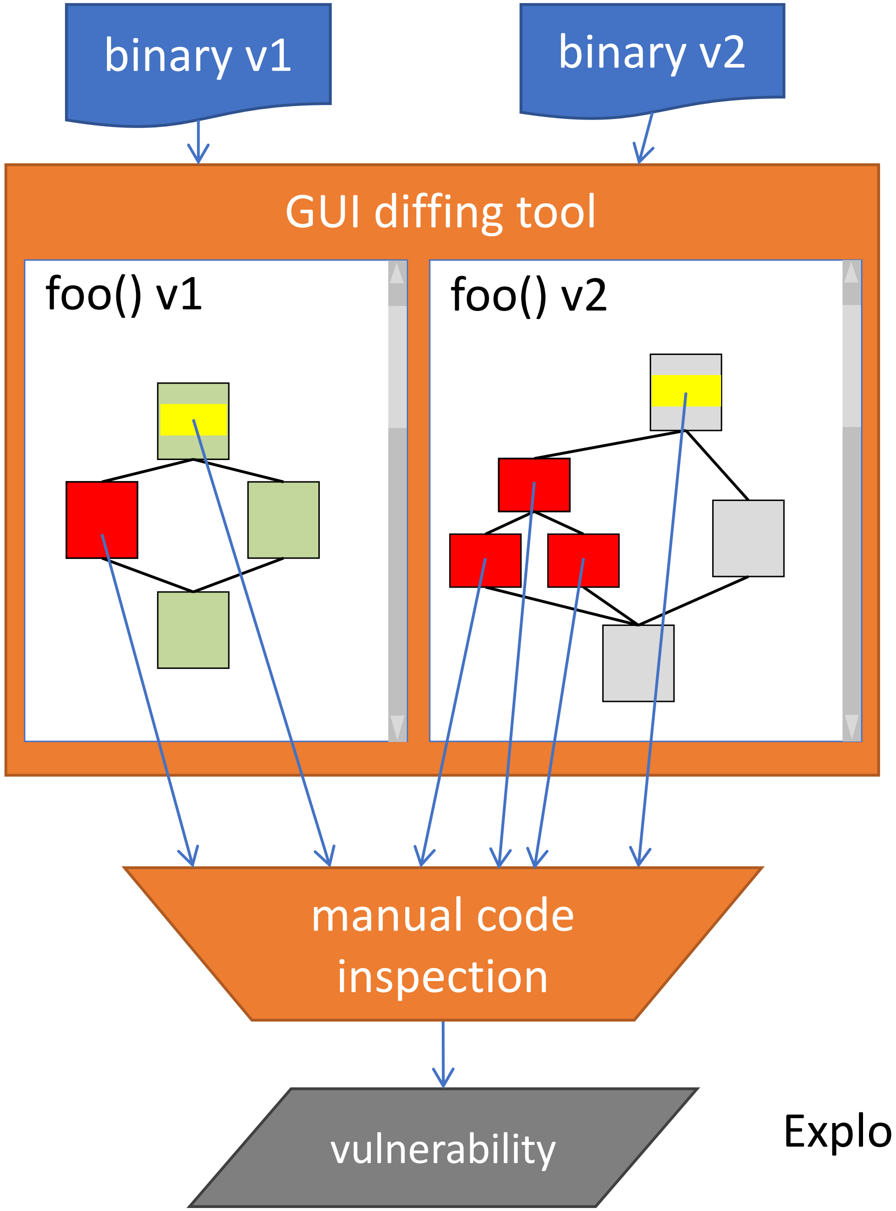
- Confusion factor

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

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

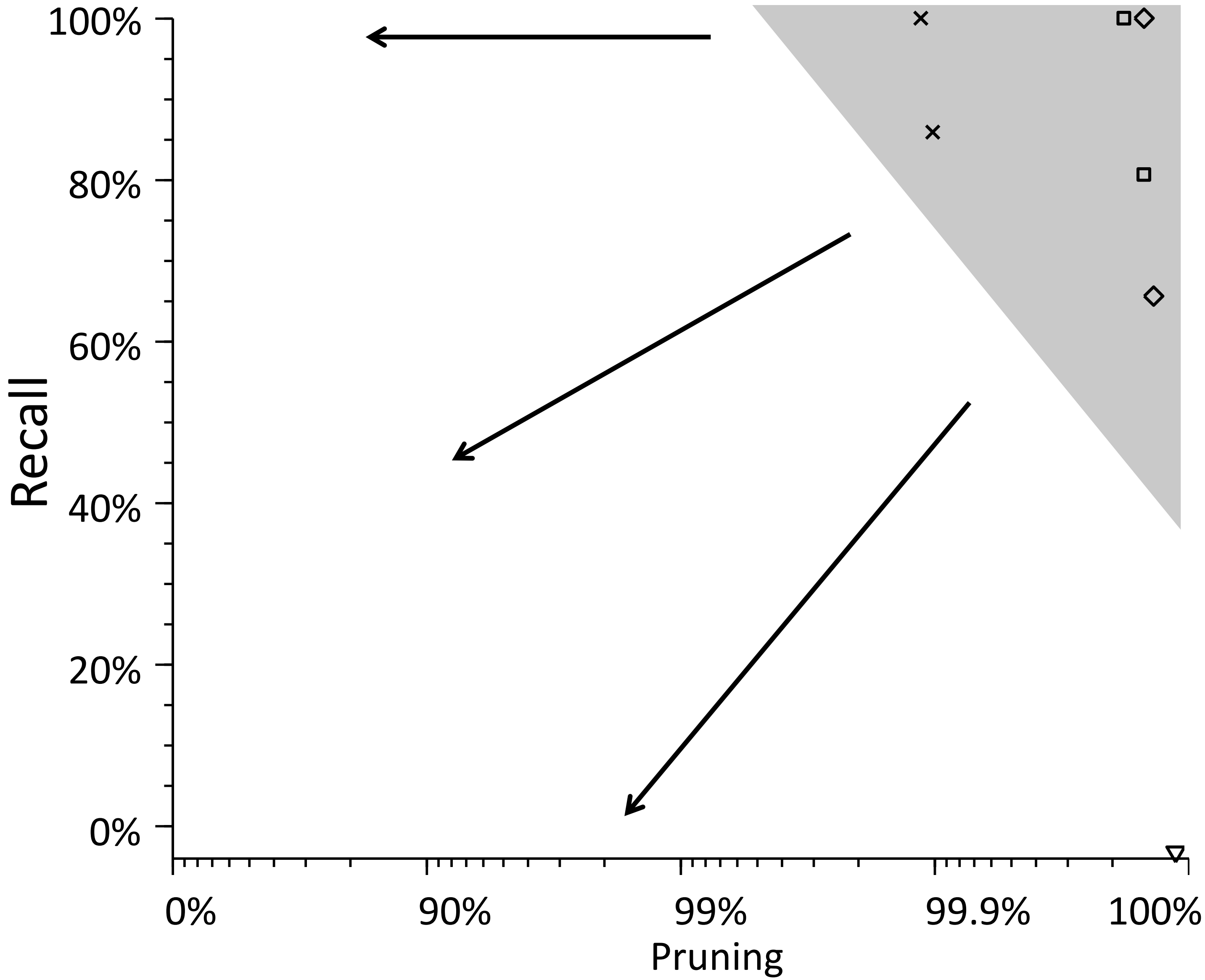
PROGRAM	Confusion factor (%)								
	LINEAR SWEEP (OBJDUMP)			RECURSIVE TRAVERSAL			COMMERCIAL (IDA PRO)		
	Instructions	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*	*Instructions*	*Basic blocks*	*Functions*
compress95	43.93	63.68	100.00	30.04	40.42	75.98	75.81	91.53	87.37
gcc	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)

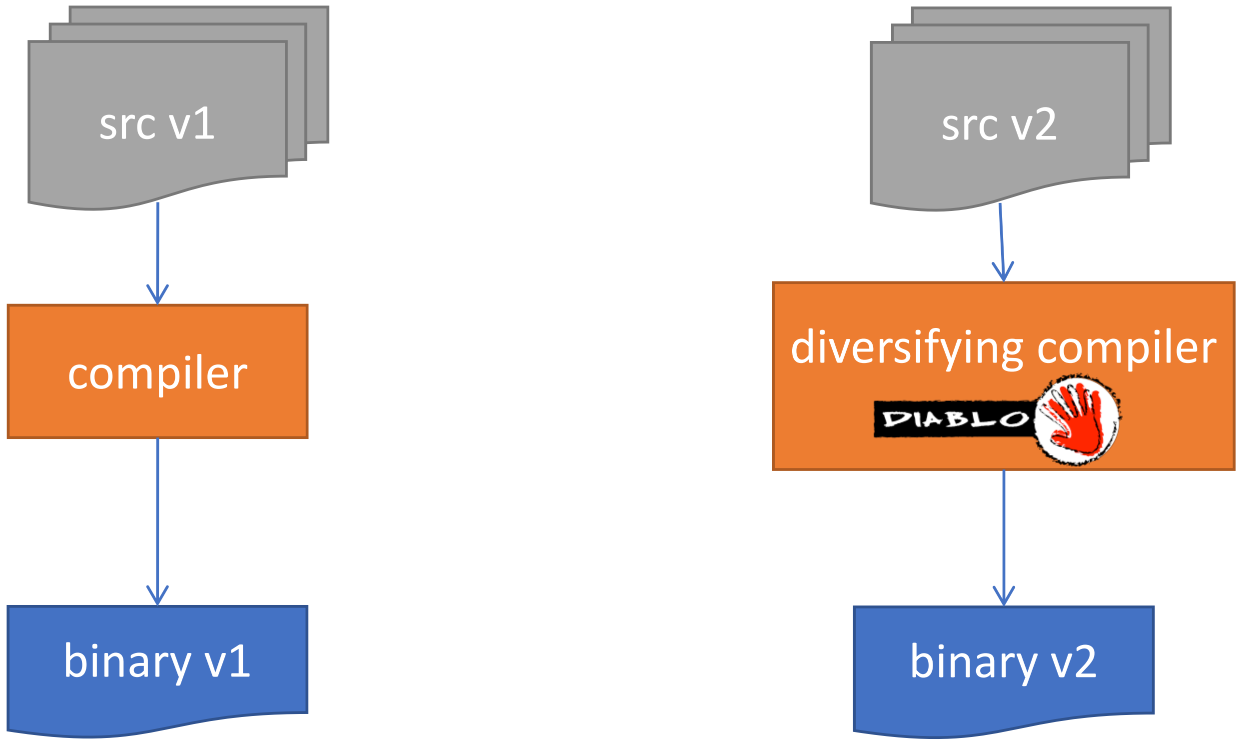


Exploit Wednesday

BinDiff on Patch Tuesday



Software Diversification



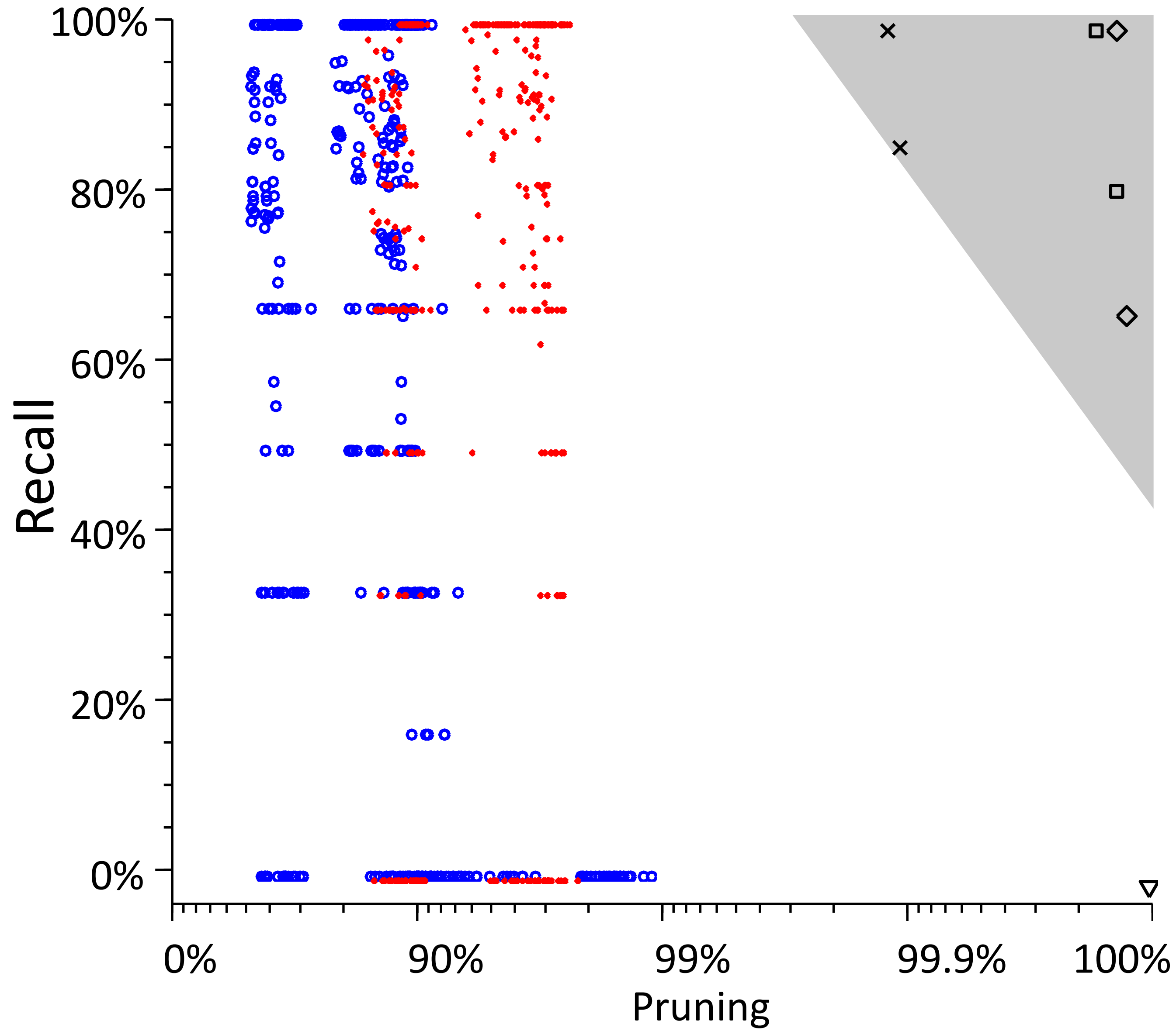
Bindiff on Patch Tuesday

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

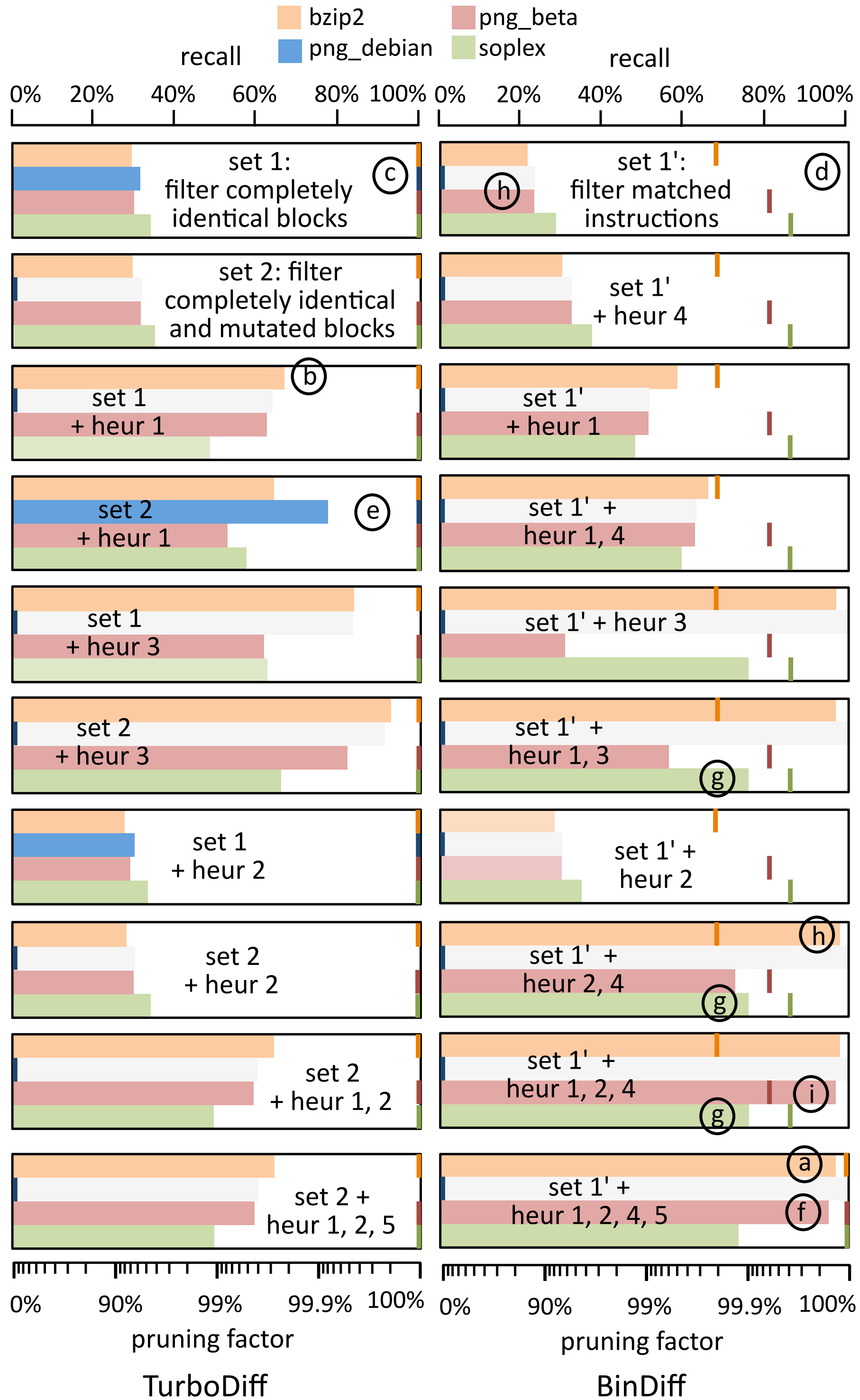
similarit	confide	change	EA primary	name primary	EA secondary	name secondary	con	algorithm
0.24	0.44	GI--E--	08076887	sub_8076887_384	0808D8C1	sub_808D8C1_1458		call reference matching
0.25	0.40	GI--E--	08063B6D	sub_8063B6D_265	0804F0A3	sub_804F0A3_701		call sequence matching(sequence)
0.25	0.83	GI-J-L-	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



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
0% 90% 99% 99.9% 100%
pruning factor
TurboDiff
BinDiff

Lecture Overview

1. Protection vis-à-vis attacks
   - attacks on what?
   - attack and protection models
2. Qualitative Evaluation
3. Quantitative Evaluation
   - complexity metrics
   - tools
4. Human Experiments

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)

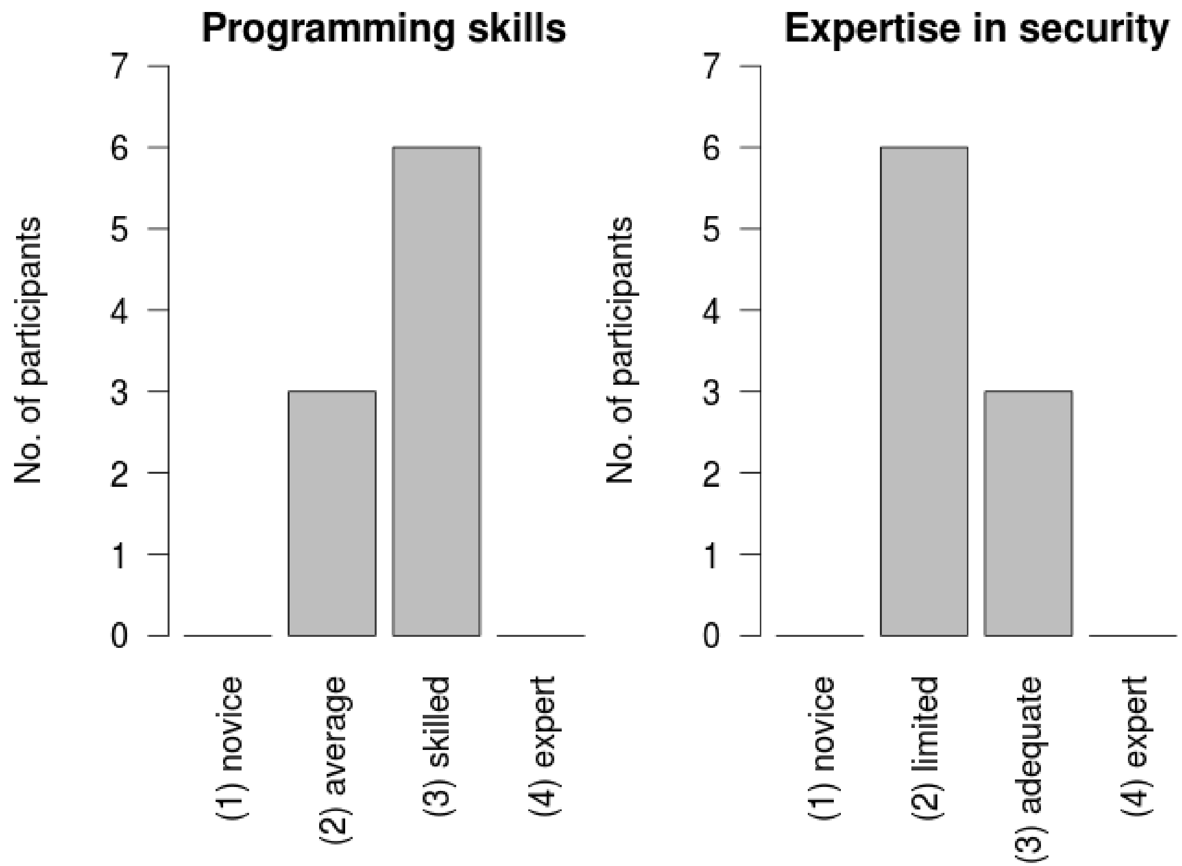
Table 4 Source code metrics debug disabled

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.

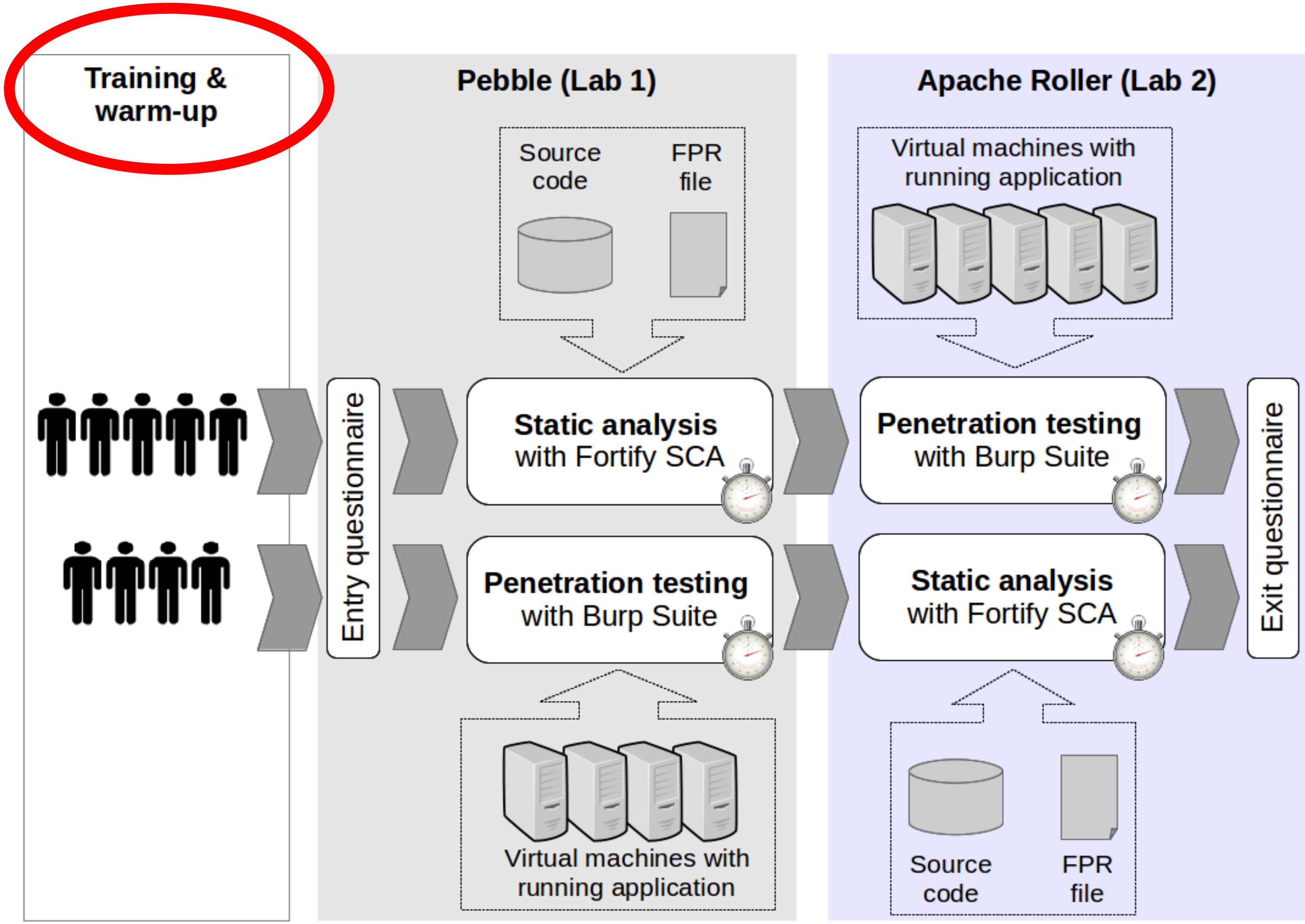
Static Analysis vs. Penetration Testing (Scandariato, 2013)

• 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

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

Static Analysis vs. Penetration Testing (Scandariato, 2013)

- Rigorous statistical analysis of the results

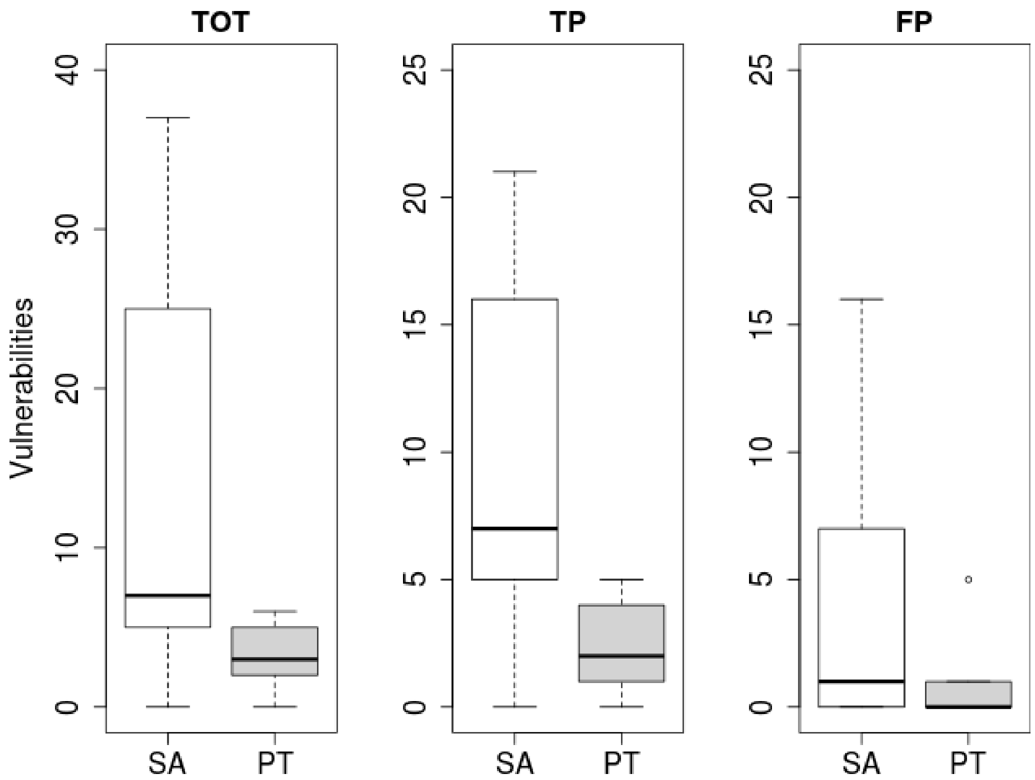


Fig. 5. Boxplot of reported results (TOT), correct results (TP) and false alarms (FP)

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 discusse
  - 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

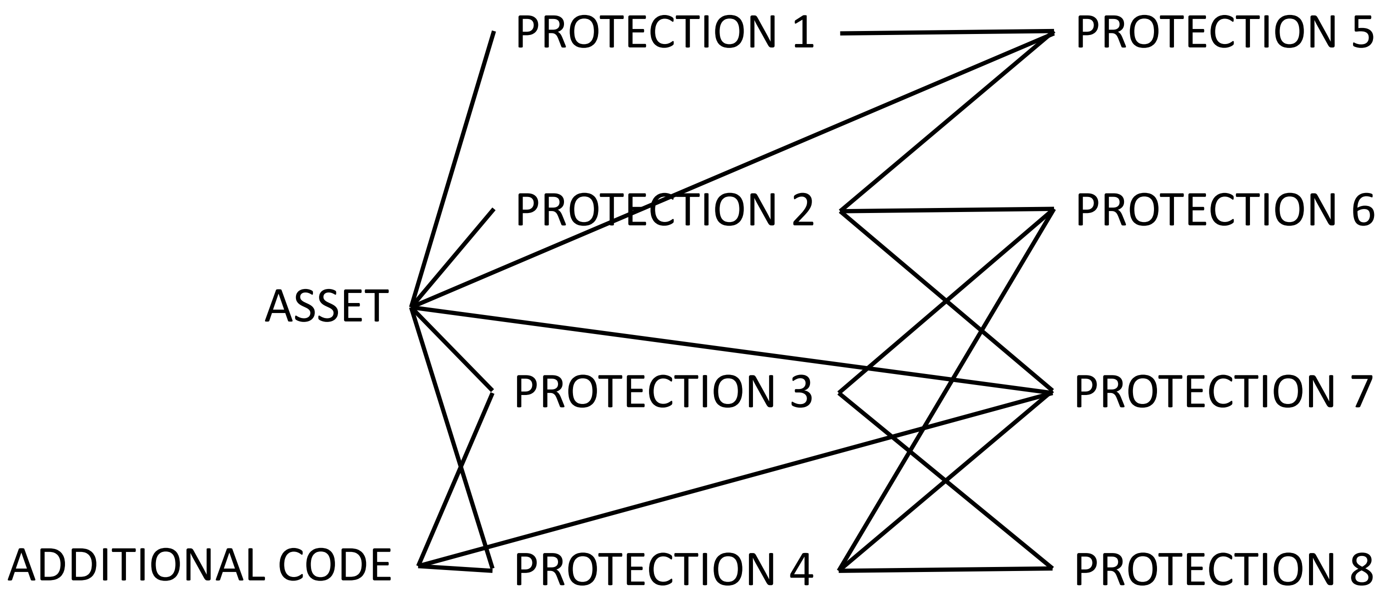
```
public void k(String s, String s1)
{
    h h1 = h(s);
    if(h1 != null)
        h1.k(s1);
}

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

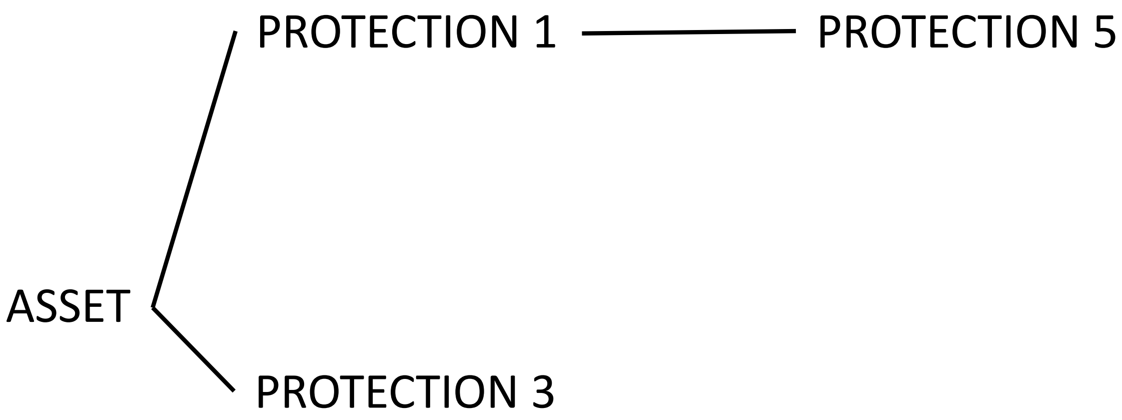
Fragment with opaque predicates

```
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());
   }
}
```

Pitfalls of small controlled experiments



Pitfalls of small controlled experiments



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**

Alternative: professional pen-tests

- How do <u>professional hackers</u> understand <u>protected</u> code when they are <u>attacking</u> it?

Participants

- Professional penetration testers working for security companies
- Routinely involved in security assessment of company's products
- Profiles:
  - Hackers with substantial experience in the field
  - Fluent with state of the art tools (reverse engineering, static analysis, debugging, profiling, tracing, …)
  - Able to customize existing tools, to develop plug-ins for them, and to develop their own custom tools
- Minimal intrusion (hacker activities can not be traced)

Experimental procedure

- Attack task definition
  - Description of the program to attack, attack scope, attack goal(s) and report structure
- Monitoring (long running experiment: 30 days)
  - Minimal intrusion into the daily activities
    - Could not be traced automatically or through questionnaires
  - Weekly conf call to monitor the progress and provide support for clarifying goals and tasks
- Attack reports
  - Final (narrative) report of the attack activities and results
  - Qualitative analysis

Objects	C	H	Java	C++	Total
DRMMediaPlayer	2,595	644	1,859	1,389	6,487
LicenseManager	53,065	6,748	819	-	58,283
OTP	284,319	44,152	7,892	2,694	338,103

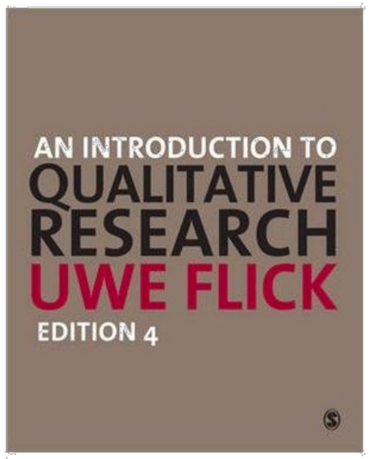
Data collection

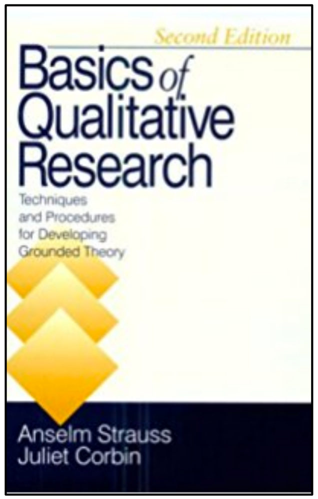
- Report in free format
- Professional hackers were asked to cover these topics:
  1. type of activities carried out during the attack;
  2. level of expertise required for each activity;
  3. encountered obstacles;
  4. decision made, assumptions, and attack strategies;
  5. exploitation on a large scale in the real world.
  6. return / remuneration of the attack effort;

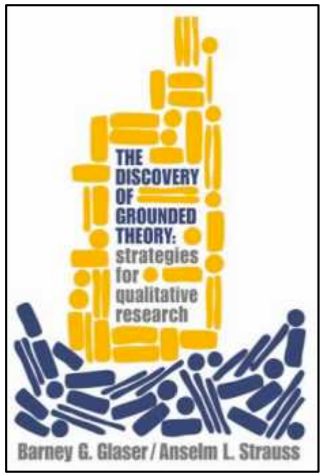
Data analysis

- Qualitative data analysis method from Grounded Theory
  - Data collection
  - Open coding
  - Conceptualization
  - Model analysis

- Not applicable to our study:
  - Immediate and continuous data analysis
  - Theoretical sampling
  - Theoretical saturation







Open coding

- Performed by 7 coders from 4 academic project partners
  - Autonomously & independently
  - High level instructions
    - Maximum freedom to coders, to minimize bias
- Annotated reports have been merged
- No unification of annotations, to preserve viewpoint diversity

	Annotator							
Case study	A	B	C	D	E	F	G	Total
P	52	34	48	53	43	49	-	279
L	20	10	6	12	7	18	9	82
O	12	22	-	29	24	11	-	98
Total	**84**	**66**	**54**	**94**	**74**	**78**	**9**	**459**

Conceptualization

1. **Concept identification**
   - Identify key concepts used by coders
   - Organize key concepts into a common hierarchy
2. **Model inference**
   - Temporal relations (e.g., *before*)
   - Causal relations (e.g., *cause*)
   - Conditional relations (e.g., *condition for*)
   - Instrumental relations (e.g., *used to*)

Conceptualization results: taxonomy of concepts

- Obstacle
  - Protection
    - Obfuscation
      - Control flow flattening
      - Opaque predicates
    - Anti debugging
    - White box cryptography
  - Execution environment
    - Limitations from operating system
  - Tool limitations
- Analysis / reverse engineering
  - String / name analysis
  - Symbolic execution / SMT solving
  - Crypto analysis
  - Pattern matching
  - Static analysis
  - Dynamic analysis
    - Dependency analysis
      - Data flow analysis
    - Memory dump
    - Monitor public interfaces
    - Debugging
    - Profiling
    - Tracing
    - Statistical analysis
      - Differential data analysis
      - Correlation analysis
  - Black-box analysis
    - File format analysis

- Attack strategy
- Attack step
  - Prepare the environment
  - Reverse engineer app and protections
    - Understand the app
      - Preliminary understanding of the app
        - Identify input / data format
      - Recognize anomalous/unexpected behaviour
      - Identify API calls
      - Understand persistent storage / file / socket
      - Understand code logic
    - Identify sensitive asset
      - Identify code containing sensitive asset
        - Identify assets by static meta info
          - Identify assets by naming scheme
      - Identify thread/process containing sensitive asset
    - Identify points of attack
      - Identify output generation
    - Identify protection
    - Run analysis
    - Reverse engineer the code
      - Disassemble the code
      - Deobfuscate the code*
  - Build the attack strategy
    - Evaluate and select alternative step / revise attack strategy
      - Choose path of least resistance
    - Limit scope of attack
      - Limit scope of attack by static meta info

- Attack step
  - Prepare attack
    - Choose/evaluate alternative tool
    - Customize/extend tool
      - Port tool to target execution environment
    - Create new tool for the attack
    - Customize execution environment
    - Build a workaround
    - Recreate protection in the small
    - Assess effort
  - Tamper with code and execution
    - Tamper with execution environment
    - Run app in emulator
    - Undo protection
      - Deobfuscate the code*
      - Convert code to standard format
      - Disable anti-debugging
      - Obtain clear code after code decryption at runtime
    - Tamper with execution
      - Replace API functions with reimplementation
      - Tamper with data
    - Tamper with code statically
    - Out of context execution
    - Brute force attack
  - Analyze attack result
    - Make hypothesis
      - Make hypothesis on protection
      - Make hypothesis on reasons for attack failure
    - Confirm hypothesis

- Workaround
- Weakness
  - Global function pointer table
  - Recognizable library
    - Shared library
    - Java library
  - Decrypt code before executing it
  - Clear key
  - Clues available in plain text
  - Clear data in memory
- Asset
- Background knowledge
  - Knowledge on execution environment framework
- Tool
  - Debugger
  - Profiler
  - Tracer
  - Emulator

- Obstacle
  - Protection
    - Obfuscation
      - Control flow flattening
      - Opaque predicates
    - Anti debugging
    - White box cryptography
  - Execution environment
    - Limitations from operating system
  - Tool limitations
- Analysis / reverse engineering
  - String / name analysis
  - Symbolic execution / SMT solving
  - Crypto analysis
  - Pattern matching
  - Static analysis
  - Dynamic analysis
    - Dependency analysis
      - Data flow analysis
    - Memory dump
    - Monitor public interfaces
    - Debugging
    - Profiling
    - Tracing
    - Statistical analysis
      - Differential data analysis
      - Correlation analysis
  - Black-box analysis
    - File format analysis

[P:F:7] General obstacle to understanding [by dynamic analysis]: execution environment (Android: limitations on network access and maximum file size)

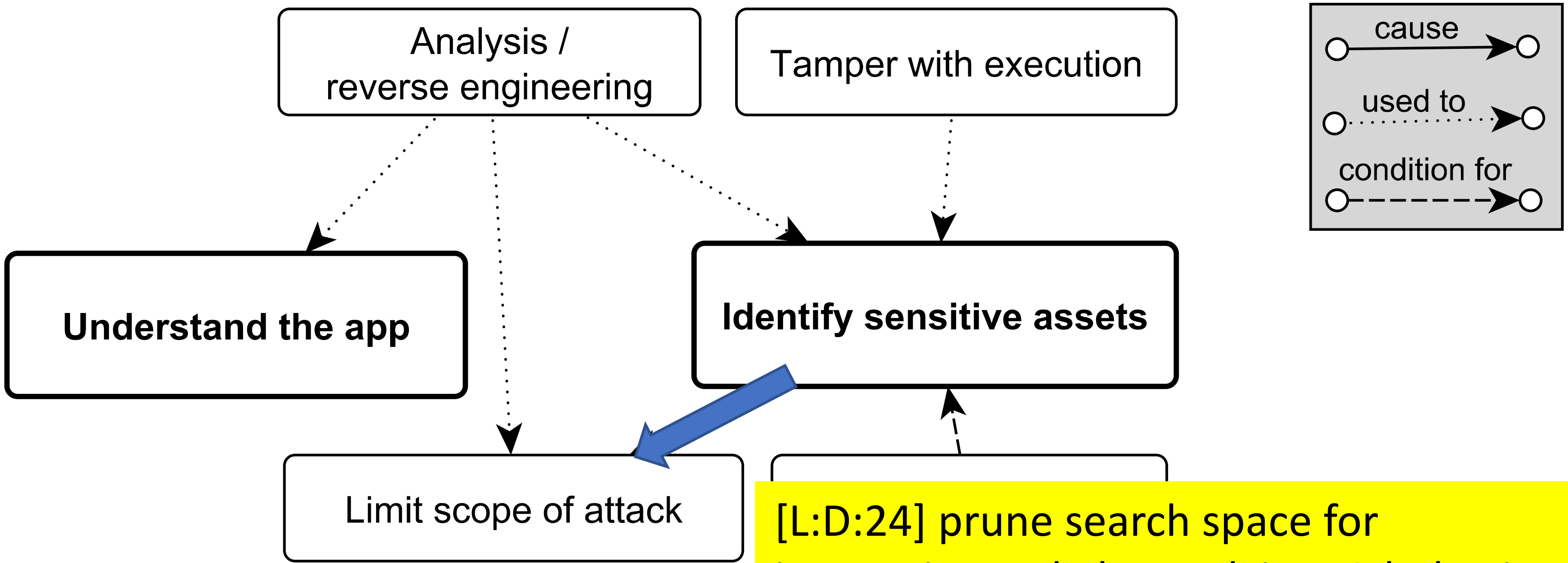
"Aside from the [omissis] added inconveniences [due to protections], execution environment requirements can also make an attacker's task much more difficult. [omissis] Things such as limitations on network access and maximum file size limitations caused problems during this exercise"

- Obstacle
  - Protection
    - Obfuscation
      - Control flow flattening
      - Opaque predicates
    - Anti debugging
    - White box cryptography
  - Execution environment
    - Limitations from operating system
  - Tool limitations
- Analysis / reverse engineering
  - String / name analysis
  - Symbolic execution / SMT solving
  - Crypto analysis
  - Pattern matching
  - Static analysis
  - Dynamic analysis
    - Dependency analysis
      - Data flow analysis
    - Memory dump
    - Monitor public interfaces
    - Debugging
    - Profiling
    - Tracing
    - Statistical analysis
      - Differential data analysis
      - Correlation analysis
  - Black-box analysis
    - File format analysis

- Attack strategy
- Attack step
  - Prepare the environment
  - Reverse engineer app and protections
    - Understand the app
      - Preliminary understanding of the app
        - Identify input / data format
      - Recognize anomalous/unexpected behaviour
      - Identify API calls
      - Understand persistent storage / file / socket
      - Understand code logic
    - Identify sensitive asset
      - Identify code containing sensitive asset
        - Identify assets by static meta info
          - Identify assets by naming scheme
      - Identify thread/process containing sensitive asset
    - Identify points of attack
      - Identify output generation
    - Identify protection
    - Run analysis
    - Reverse engineer the code
      - Disassemble the code
      - Deobfuscate the code*
  - Build the attack strategy
    - Evaluate and select alternative step / revise attack strategy
      - Choose path of least resistance
    - Limit scope of attack
      - Limit scope of attack by static meta info

- Attack step
  - Prepare attack
    - Choose/evaluate alternative tool
    - Customize/extend tool
      - Port tool to target execution environment
    - Create new tool for the attack
    - Customize execution environment
    - Build a workaround
    - Recreate protection in the small
    - Assess effort
  - Tamper with code and execution
    - Tamper with execution environment
    - Run app in emulator
    - Undo protection
      - Deobfuscate the code*
      - Convert code to standard format
      - Disable anti-debugging
      - Obtain clear code after code decryption at runtime
    - Tamper with execution
      - Replace API functions with reimplementation
      - Tamper with data
    - Tamper with code statically
    - Out of context execution
    - Brute force attack
  - Analyze attack result
    - Make hypothesis
      - Make hypothesis on protection
      - Make hypothesis on reasons for attack failure
    - Confirm hypothesis

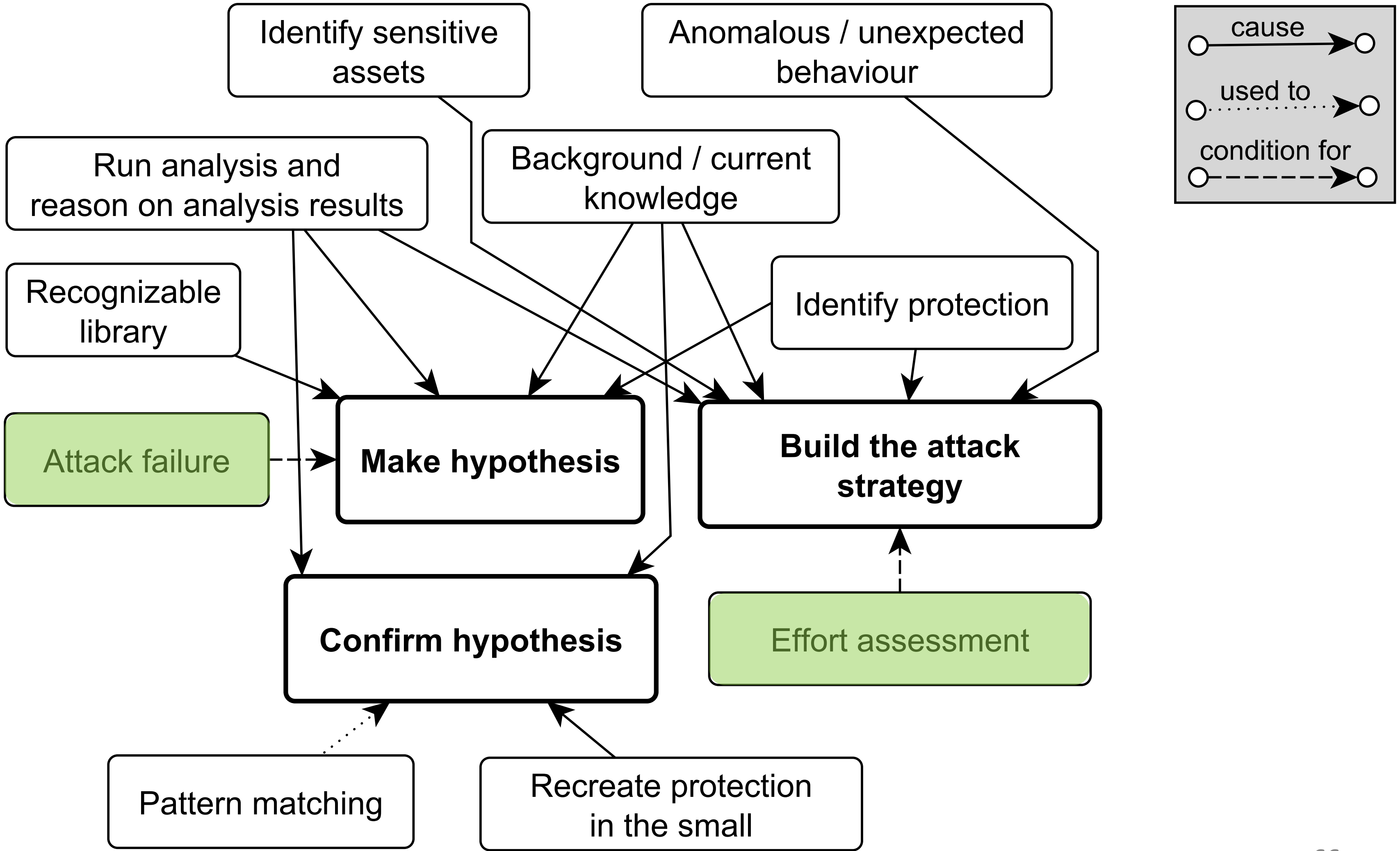
How hackers understand protected software



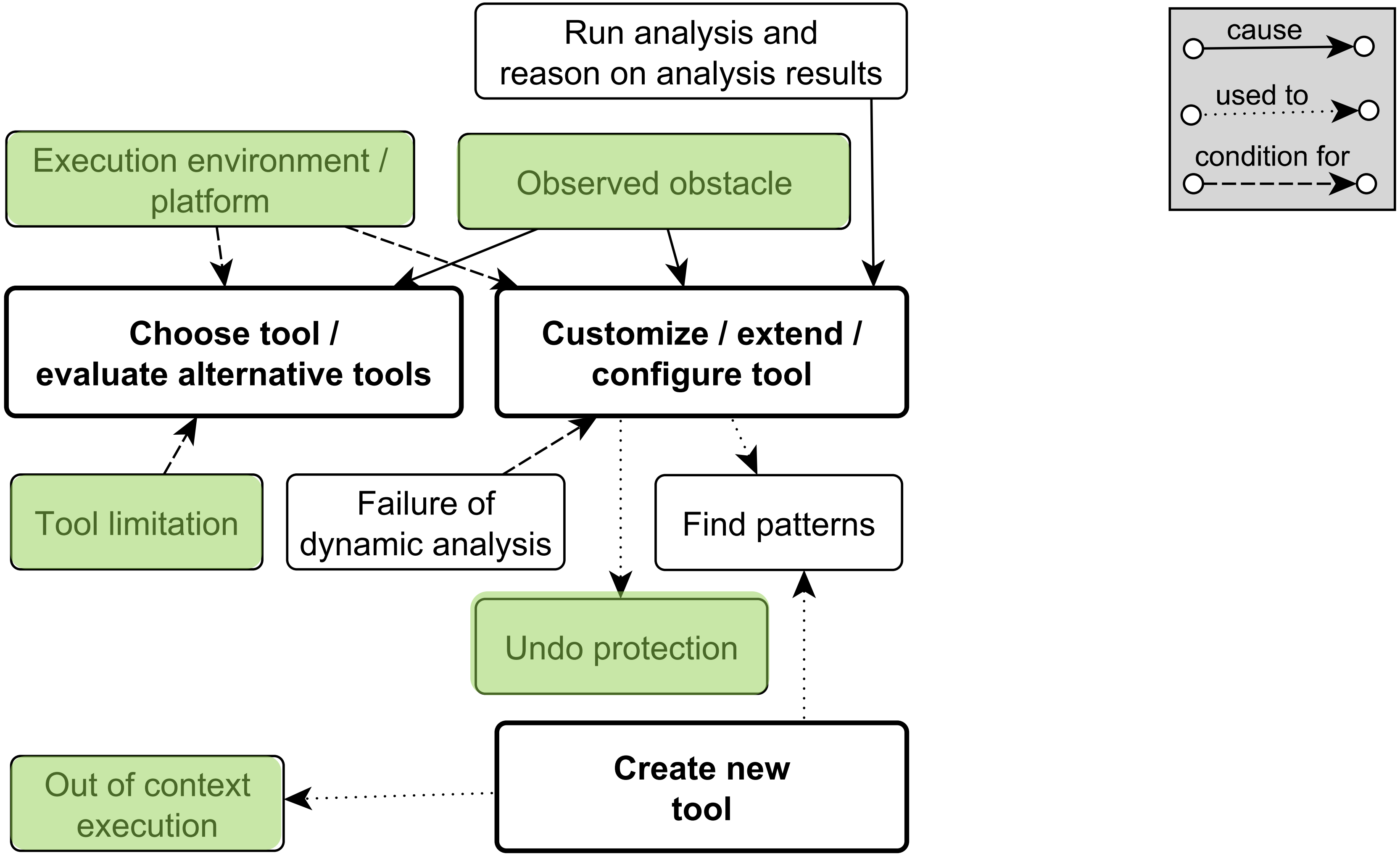
[L:D:24] prune search space for interesting code by studying IO behavior, in this case system calls

[L:D:26] prune search space for interesting code by studying static symbolic data, in this case string references in the code

How hackers build attack strategies



How attackers chose & customize tools



How hackers workaround & defeat protections