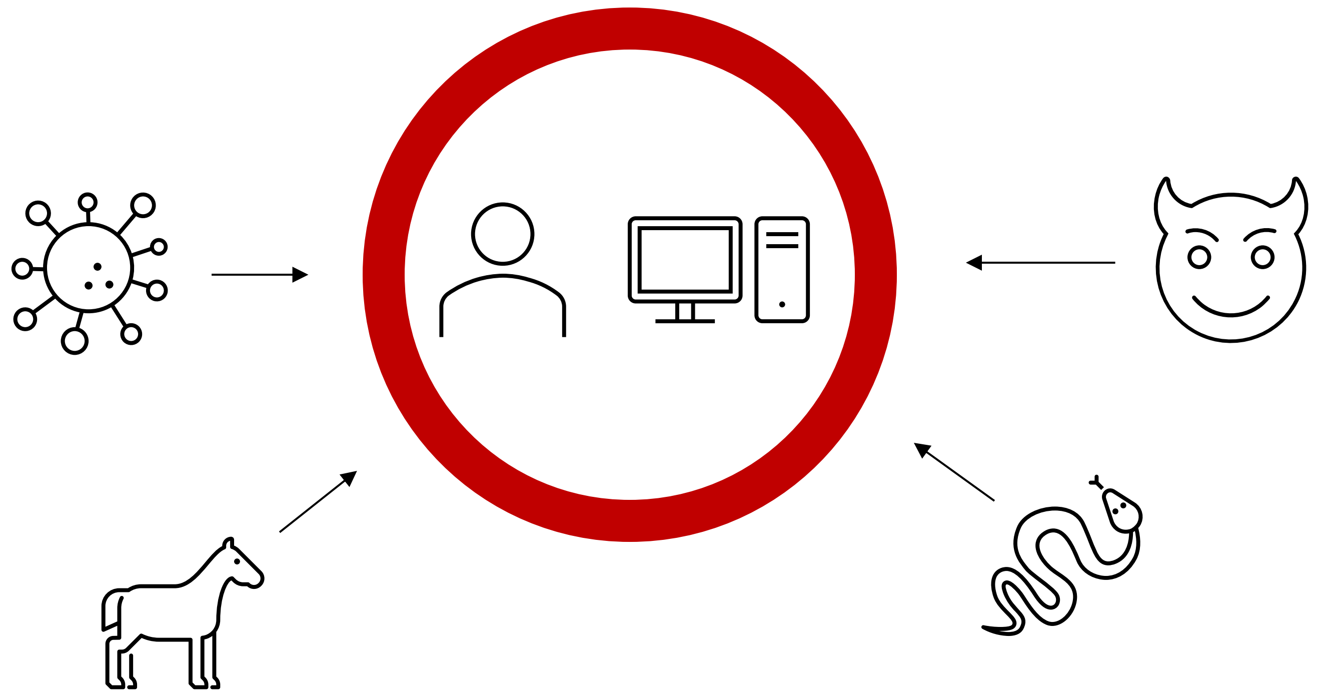
Protecting Software Against Man-At-The-End Attacks

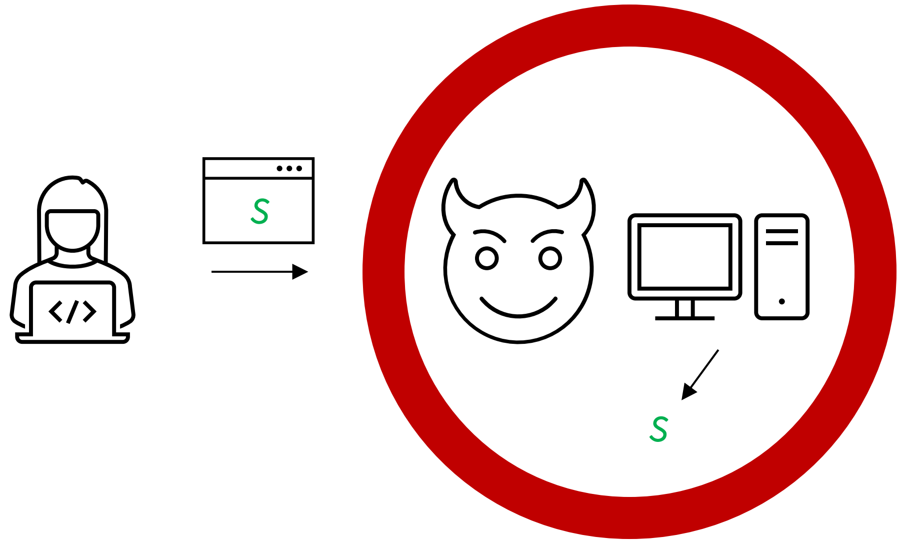
The Efficiency Challenge

Dr. Sebastian Schrittwieser · Christian Doppler Laboratory AsTra

Man-at-the-End Attack Scenario



Man-at-the-End Attack Scenario



Why software protections?

System architecture

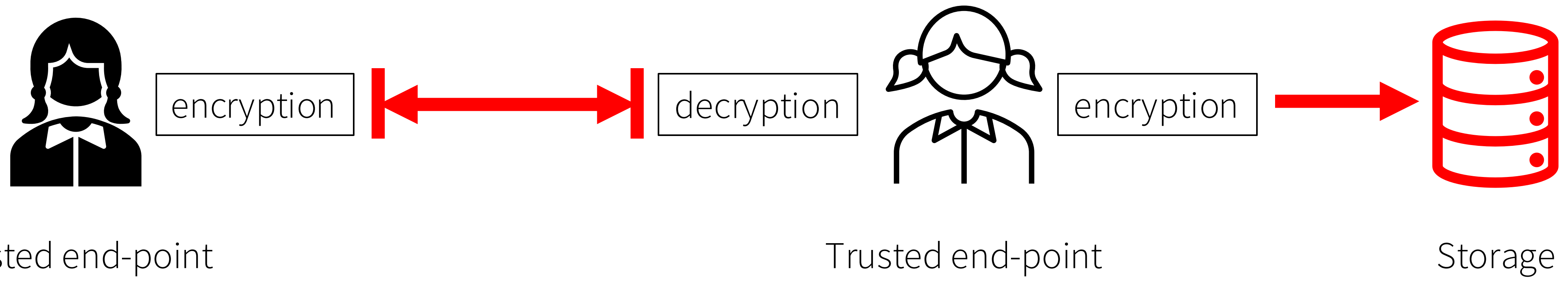
Cryptographic key

Protection of some secret in software!

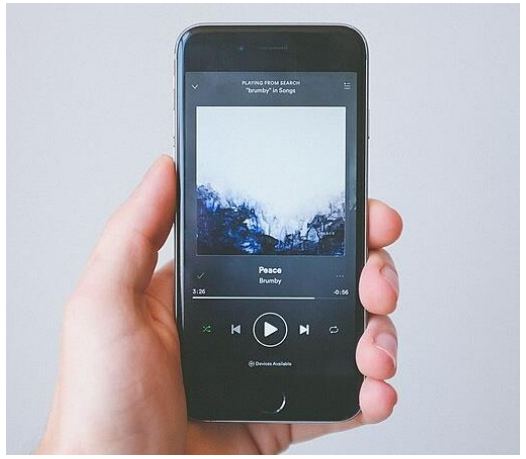
Superior algorithm

Copy-protection mechanism

Encrypting assets in software?



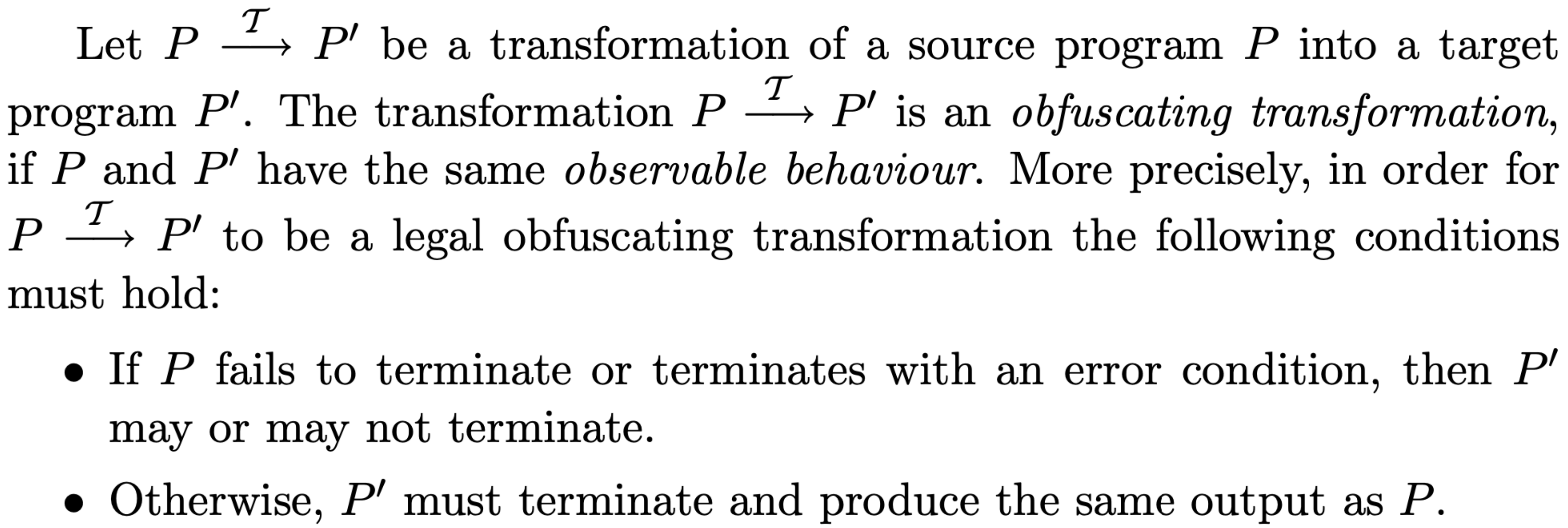
No trusted end-points and no data at rest!



Software Protection

Analysis	Modification	Theft
Code Obfuscation Increasing the complexity of program code	**Software Tamperproofing** Adding program logic that detects code modifications	**Watermarking** Making a program (uniquely) identifiable

Definitions



Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'. The transformation $P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behaviour*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

(Collberg et a. 1997)

Definitions

- More formal definition by Barak et al. [2001]
- An obfuscator *O* is a “compiler” which takes as input a program *P* and produces a new program *O(P)* such that for every *P*:
  - Functionality: *O(P)* computes the same function as *P*
  - Polynomial Slowdown: The description length and running time of *O(P)* are at most polynomially larger than that of *P*
  - “Virtual black box” property: “Anything that can be efficiently computed from *O(P)* can be efficiently computed given oracle access to *P*“

Classification of obfuscating algorithms

Data obfuscation

Static obfuscation

Dynamic obfuscation

Data obfuscation

- Make data look different
- Goal: An attacker is unable to locate data based on its known structure
- Example: AES key
    - 128/192/256 bit, high entropy
    - Splitting the key into multiple fragments makes pattern matching based identification more difficult

Static code rewriting

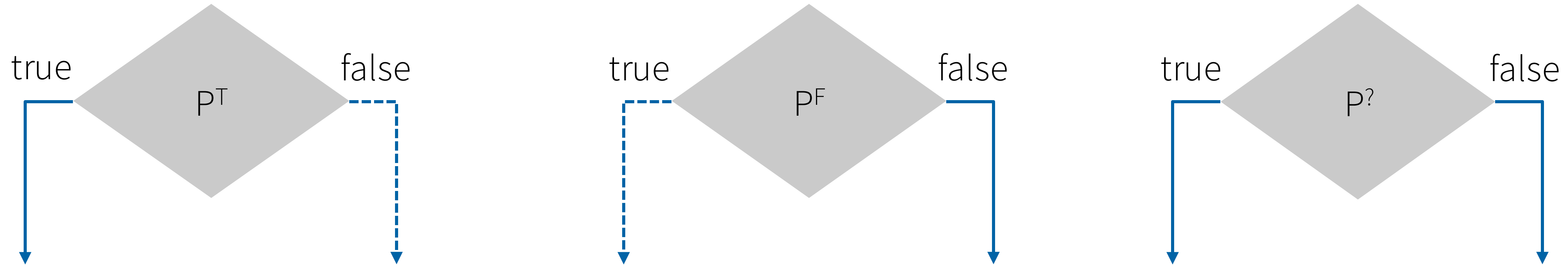
Opaque predicates	Inserting dead or irrelevant code	Replacing instructions	Reordering	Loop trans-formations	Function splitting/ recombination
Aliasing	Control flow obfuscation	Parallelized code	Name scrambling	Removing standard library calls	Breaking relations

Name scrambling

```
public long convert(float amountDollar, float rate)
   amountEuro = amountDollar * rate;
   return amountEuro;
}
```

Opaque predicates

- Opaque expression: Expression whose value is known at obfuscation time, but difficult for an attacker to figure out
- Most common are opaque predicates (boolean valued expressions)
- Mixed Boolean-Arithmetic (MBA)



Mixed Boolean-Arithmetic (MBA)

x+y $\longrightarrow$

```
4*(~x&y)-(x^y)-(x|y)
+4*~(x|y)-~(x^y)-~y-
(x|~y)+1+6*x+5*~z+
(~(x^z))-(x|z)-2*~x-
4*(~(x|z))-4*(x&~z)
+3*(~(x|~z))
```

Source: Liu, Binbin, et al. "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation." 30th USENIX Security Symposium. 2021.

Aliasing

```
int function1(int* a, int* b) {
    *a = 10;
    *b = 5;
    if ((*a - *b) == 0) function2();
}
```

```
function(&x, &x)
```

Dynamic code rewriting

Packing/ Encryption	Dynamic code modifications	Environmental requirements
Hardware-assisted obfuscation	Virtualization	Anti-debugging techniques

Virtualization

- One of the most advanced techniques for binary obfuscation
- Converting the program’s functionality into byte code for a custom virtual machine interpreter that is bundled with the program
- The virtual machine interpreter and payload can be different for each instance of the program (polymorphism)

Software Protection

Analysis	Modification	Theft
Code Obfuscation Increasing the complexity of program code	**Software Tamperproofing** Adding program logic that detects code modifications	**Watermarking** Making a program (uniquely) identifiable

Tampering with software

```
if (license_expired()) {
    printf(“expired!”);
    abort();
}
```

Tampering with software

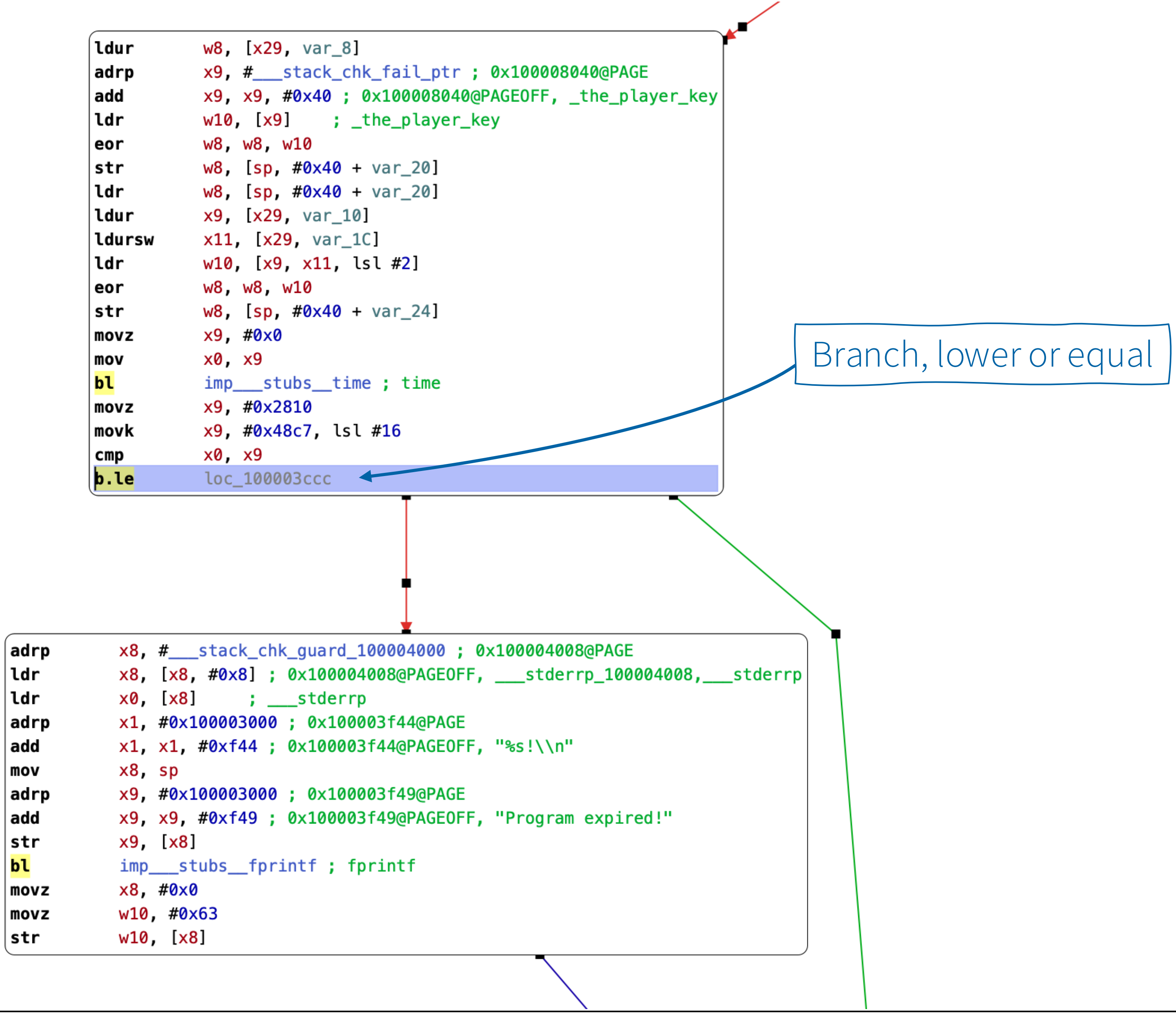
```
if (false) {
    printf(“expired!”);
    abort();
}
```

Tampering with software

```
if (dateToday > expirationDate) {
   printf(“expired!”);
   abort();
}
```

Tampering with software

```
if (dateToday < expirationDate) {
    printf("expired!");
    abort();
}
```



```
ldur      w8, [x29, var_8]
adrp      x9, #___stack_chk_fail_ptr ; 0x100008040@PAGE
add       x9, x9, #0x40 ; 0x100008040@PAGEOFF, _the_player_key
ldr       w10, [x9]     ; _the_player_key
eor       w8, w8, w10
str       w8, [sp, #0x40 + var_20]
ldr       w8, [sp, #0x40 + var_20]
ldur      x9, [x29, var_10]
ldursw    x11, [x29, var_1C]
ldr       w10, [x9, x11, lsl #2]
eor       w8, w8, w10
str       w8, [sp, #0x40 + var_24]
movz      x9, #0x0
mov       x0, x9
bl        imp___stubs__time ; time
movz      x9, #0x2810
movk      x9, #0x48c7, lsl #16
cmp       x0, x9
b.le      loc_100003ccc
```

Branch, lower or equal

```
adrp      x8, #___stack_chk_guard_100004000 ; 0x100004008@PAGE
ldr       x8, [x8, #0x8] ; 0x100004008@PAGEOFF, ___stderrp_100004008,___stderrp
ldr       x0, [x8]      ; ___stderrp
adrp      x1, #0x100003000 ; 0x100003f44@PAGE
add       x1, x1, #0xf44 ; 0x100003f44@PAGEOFF, "%s!\\n"
mov       x8, sp
adrp      x9, #0x100003000 ; 0x100003f49@PAGE
add       x9, x9, #0xf49 ; 0x100003f49@PAGEOFF, "Program expired!"
str       x9, [x8]
bl        imp___stubs__fprintf ; fprintf
movz      x8, #0x0
movz      w10, #0x63
str       w10, [x8]
```

Code checking

```
if (hash(P’s code) != 0xda6ba121)
   return false;
```

Code checking

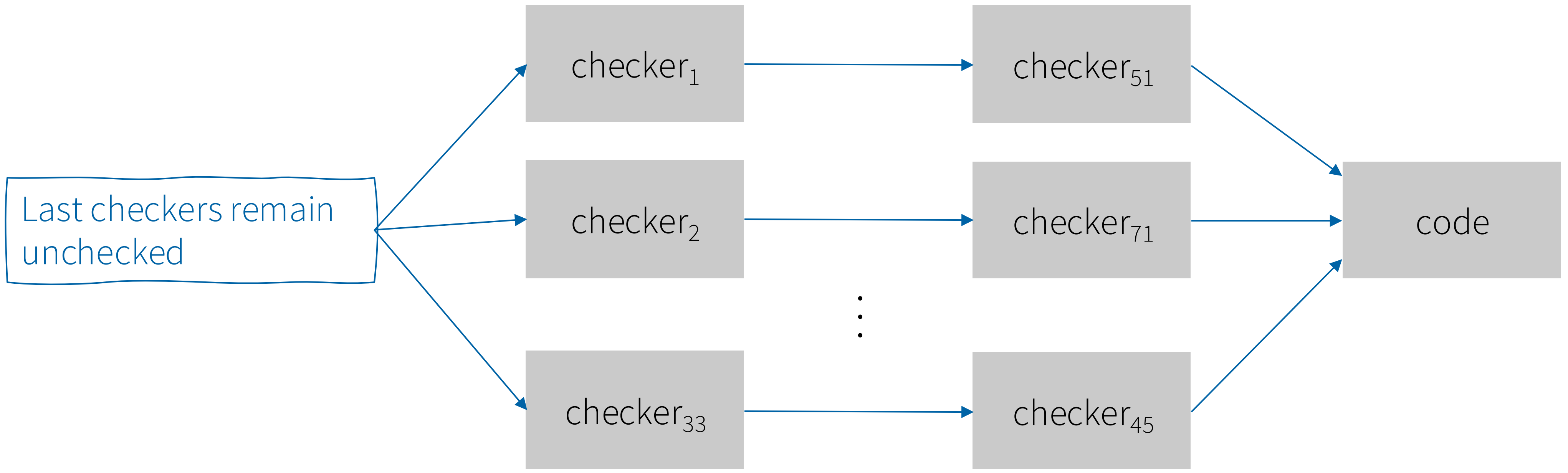
- Can be broken easily with simple pattern matching attacks
  - Identification of the hashing algorithm
  - Identification of hard-coded hash value

```
if (hash(P’s code) == 0xda6ba121)
   return false;
```

Protecting the hash value

- Hard-coded hash values are problematic
  - Collusive attacks: if each copy of your program looks different (e.g., because it is fingerprinted) an attacker can look for differences in literal data to identify the hash values
- Countermeasures
  - Adding a copy of the hashed region and then compare the hashes of the two regions
  - Construction of the hash function so that unless the code has been modified, the function hashes to zero [Horne et al. 2001]
    - Accomplished by using an invertible hash function and adding a semantically irrelevant data value (*slot*) that makes the region hash to zero

Code checker networks



Software Tamperproofing

- The response of a program to tampering attempts is crucial for its protection strength
- Famous example: Settlers III
  - Copy protection „Sysiphus“
  - A pirated copy of the game runs perfectly at first
  - After a few hours, iron melters produce only pigs, no new settlers are born, newly planted trees don‘t grow, goods placed at a harbour for transport disappear

Response mechanisms

- Spatial separation
  - Static separation: making sure that detection and response are far away from each other in the binary
  - Dynamic separation: making sure that the response function is not on the call stack when the detection takes place or that many functions are called between detection and response
- Temporal separation
  - Wait a significant amount of time, before responding to tampering
  - Do not wait for too long, otherwise the attacker can still cause damage until your program responds to the tampering

Software Protection

Analysis	Modification	Theft
Code Obfuscation Increasing the complexity of program code	**Software Tamperproofing** Adding program logic that detects code modifications	**Watermarking** Making a program (uniquely) identifiable

Instruction mapping watermark

- Idea: Replacing opcodes in a dummy function to encode the watermark

1. Add an unrelated dummy function *D* to the program
2. Modify *D*'s opcodes to embed the mark
3. Add a bogus call `if (`P^F`) `D`()`, protected by an opaque predicate, to tie *D* to the cover program

```
000 → iadd, 001 → isub, 010 → imul, 011 → idiv, 100
→ irem, 101 → ior, 110 → iand, 111 → ixor
```

Protection evaluation

Protection evaluation

- In contrast to cryptography, it is very difficult to make a statement about the strength of an obfuscation
  - Barak‘s impossibility result from 2002
    - A general obfuscator for any program does not exist
    - Practical applicability of indistinguishability obfuscation remains unclear
  - Protection strength depends on a variety of parameters, including the motivation and creativity of a human analyst
- Collberg et al. proposed a taxonomy for obfuscations in 1997
  - Potency, resilience, stealth, and costs
    - There is no universally agreed and applicable method to define, qualify, or quantify the former three

Potency

- Potency measures how effectively a protection increases the difficulty, time, or resources required for an analysis or reduces the precision or usefulness of its results
- Potency is achieved by increasing the apparent complexity of the object to be analyzed or lowering its suitability as input for analysis, relative to specific analysis methods
- While traditionally focused on human manual analysis, we argue that potency also includes automated analyses, such as thwarting advanced malware detection and classification techniques

Potency

- A potent obfuscating transformation makes at least one analysis method harder to perform and no analysis easier
- Mila Dalla Preda presented a potency framework based on abstract domains
  - Comparing the properties that are preserved by obfuscation transformations
  - A transformation that preserves more properties is weaker than one than preserves less
  - Often, domains of obfuscating transformations are not comparable
  - Example: (very simple) data obfuscation

Potency

```
T1(x) = 2 * x
T2(x) = 3 * x
```

```
            -1 if x < 0
Sign(x):     0 if x = 0
             1 if x > 0

Parity(x):   0 if even(x)
             1 if odd(x)
```

Both obfuscations preserve the *Sign* property

Only T1 preserves the *Parity* property

Two simple data obfuscations

Properties *Sign* and *Parity*

T1 preserves less, thus is more potent

Resilience

- Resilience measures how difficult it is to weaken or bypass a protection, specifically by adapting or developing new analyses or transformations to counteract its potency
- Counter-attacks may involve modifying existing analyses or designing new approaches that reduce the protection's effectiveness with respect to specific analyses
- A protection is considered resilient if efforts to mitigate its potency (e.g., for malware detection) are challenging or only partially successful, even with advanced or alternative techniques

Stealth

- Stealth of protected code
  - Can obfuscated code be distinguished from untransformed code?
- Local stealth
  - An attacker cannot determine a particular instruction as being affected by an obfuscating transformation
- Steganographic stealth
  - An attacker can not determine if a program has been transformed with a certain transformation or not

Costs

- Computational overhead (runtime, memory consumption, etc.) of an obfuscating transformation
- Measuring costs is easy compared to potency and resilience
- However, meaningless without potency, resilience, and stealth measurements
- What are acceptable costs?
  - Highly depends on the concrete use case
  - Typically, software protections add a significant computational overhead to a program

universität wien

Literature survey

- In-depth analysis of 571 publications on software protections
- For each paper, we collected 113 aspects
  - Contribution area (obfuscation, deobfuscation, analysis, etc.)
  - Types of performed measurements
  - Used sample sets
  - etc.
- 64.523 individual data points
- Published in ACM Computing Surveys

Evaluation Methodologies in Software Protection Research

BJORN DE SUTTER*, Electronics and Information Systems, Ghent University, Gent, Belgium
SEBASTIAN SCHRITTWIESER*, Christian Doppler Laboratory for Assurance and Transparency in Software
Protection, University of Vienna, Vienna, Austria
BART COPPENS, Electronics and Information Systems, Ghent University, Gent, Belgium
PATRICK KOCHBERGER, Institute of IT Security Research, St. Pölten University of Applied Sciences, St.
Pölten, Austria

Man-at-the-end (MATE) attackers have full control over the system on which the attacked software runs, and try to break the
confidentiality or integrity of assets embedded in the software. Both companies and malware authors want to prevent such
attacks. This has driven an arms race between attackers and defenders, resulting in a plethora of different protection and
analysis methods. However, it remains difficult to measure the strength of protections because MATE attackers can reach their
goals in many different ways and a universally accepted evaluation methodology does not exist. This survey systematically
reviews the evaluation methodologies of papers on obfuscation, a major class of protections against MATE attacks. For 571
papers, we collected 113 aspects of their evaluation methodologies, ranging from sample set types and sizes, over sample
treatment, to performed measurements. We provide detailed insights into how the academic state of the art evaluates both
the protections and analyses thereon. In summary, there is a clear need for better evaluation methodologies. We identify nine
challenges for software protection evaluations, which represent threats to the validity, reproducibility, and interpretation of
research results in the context of MATE attacks and formulate a number of concrete recommendations for improving the
evaluations reported in future research papers.

CCS Concepts: • **Security and privacy → Software reverse engineering.**

Additional Key Words and Phrases: survey, software protection, obfuscation, deobfuscation, diversification

1 Introduction

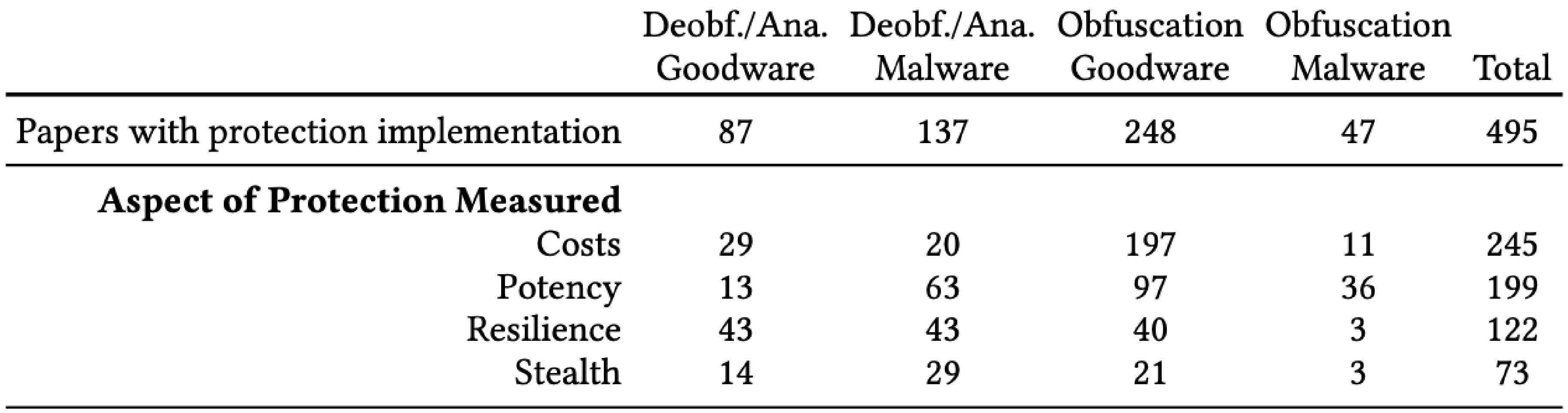
The desire and need to protect software from analysis, reverse engineering, and tampering have always existed.
Benign software vendors want to keep the exact implementation of their product and its embedded data, such
as cryptographic keys, secret. They want to prevent the removal of copy protections, cheating in multi-player
games, etc. Malware authors also often employ *software protection* (SP) mechanisms to prevent detection and
analysis of the malicious nature of their code.

Over the past three decades, a vast number of obfuscation, tamperproofing, and anti-analysis techniques for
both goodware and malware have been introduced in the literature [73, 98, 148] and in industry to meet the
described needs. The deployment of such protections has in turn triggered an arms race in which many new
deobfuscation, tampering, and analysis techniques have been proposed. These are collectively called *man-at-the-
end* (MATE) attacks, because the *human attackers* using the deobfuscation, tampering, and analysis techniques

*B. De Sutter and S. Schrittwieser share dual first authorship.

Authors' Contact Information: Bjorn De Sutter, bjorn.desutter@ugent.be, and Bart Coppens, bart.coppens@ugent.be, Computing Systems
Lab, Ghent University, Ghent, Belgium; Sebastian Schrittwieser, sebastian.schrittwieser@univie.ac.at, Christian
Assurance and Transparency in Software Protection, Faculty of Computer
patrick.kochberger@fhstp.ac.at, St. Pölten

Measured aspects



	Deobf./Ana. Goodware	Deobf./Ana. Malware	Obfuscation Goodware	Obfuscation Malware	Total
Papers with protection implementation	87	137	248	47	495
Aspect of Protection Measured					
Costs	29	20	197	11	245
Potency	13	63	97	36	199
Resilience	43	43	40	3	122
Stealth	14	29	21	3	73

Lack of knowledge on protection strength

- Focus on cost measurements, lack of strength measurements
  - 25% of all analyzed papers do not report strength measurements
    - 38% of the obfuscation goodware papers
  - Not particularly surprising, as cost measurements (incl. interpretation) are rather straightforward
- Layering of protections is not well researched
- The more the merrier?
  - Most software protections have a significant impact on code efficiency
  - Goal: Implement protections that are sufficiently strong, avoiding unnecessary inefficiencies

Efficient software protections

Targeting specific protection scenarios*

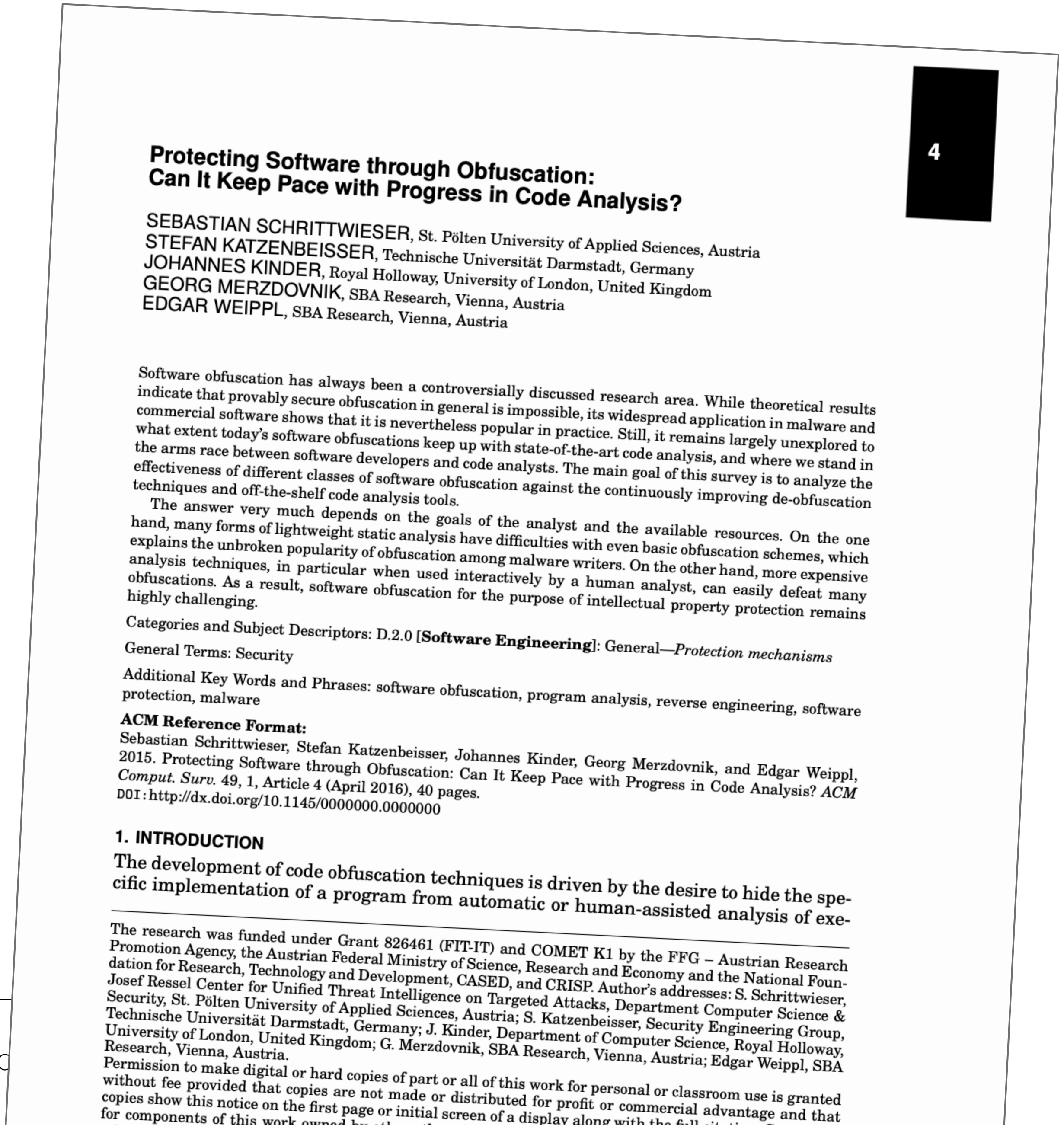
**Objectives and capabilities of the attacker*

Quantifying protection strength*

**Potency, resilience, and stealth*

Software protection scenarios

- Generation of 22 software protection scenarios
- Evaluation of the effectiveness of different classes of obfuscation against code analysis and de-obfuscation techniques in each of the 22 scenarios



4

Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?

SEBASTIAN SCHRITTWIESER, St. Pölten University of Applied Sciences, Austria
STEFAN KATZENBEISSER, Technische Universität Darmstadt, Germany
JOHANNES KINDER, Royal Holloway, University of London, United Kingdom
GEORG MERZDOVNIK, SBA Research, Vienna, Austria
EDGAR WEIPPL, SBA Research, Vienna, Austria

Software obfuscation has always been a controversially discussed research area. While theoretical results indicate that provably secure obfuscation in general is impossible, its widespread application in malware and commercial software shows that it is nevertheless popular in practice. Still, it remains largely unexplored to what extent today's software obfuscations keep up with state-of-the-art code analysis, and where we stand in the arms race between software developers and code analysts. The main goal of this survey is to analyze the effectiveness of different classes of software obfuscation against the continuously improving de-obfuscation techniques and off-the-shelf code analysis tools.

The answer very much depends on the goals of the analyst and the available resources. On the one hand, many forms of lightweight static analysis have difficulties with even basic obfuscation schemes, which explains the unbroken popularity of obfuscation among malware writers. On the other hand, more expensive analysis techniques, in particular when used interactively by a human analyst, can easily defeat many obfuscations. As a result, software obfuscation for the purpose of intellectual property protection remains highly challenging.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General—*Protection mechanisms*

General Terms: Security

Additional Key Words and Phrases: software obfuscation, program analysis, reverse engineering, software protection, malware

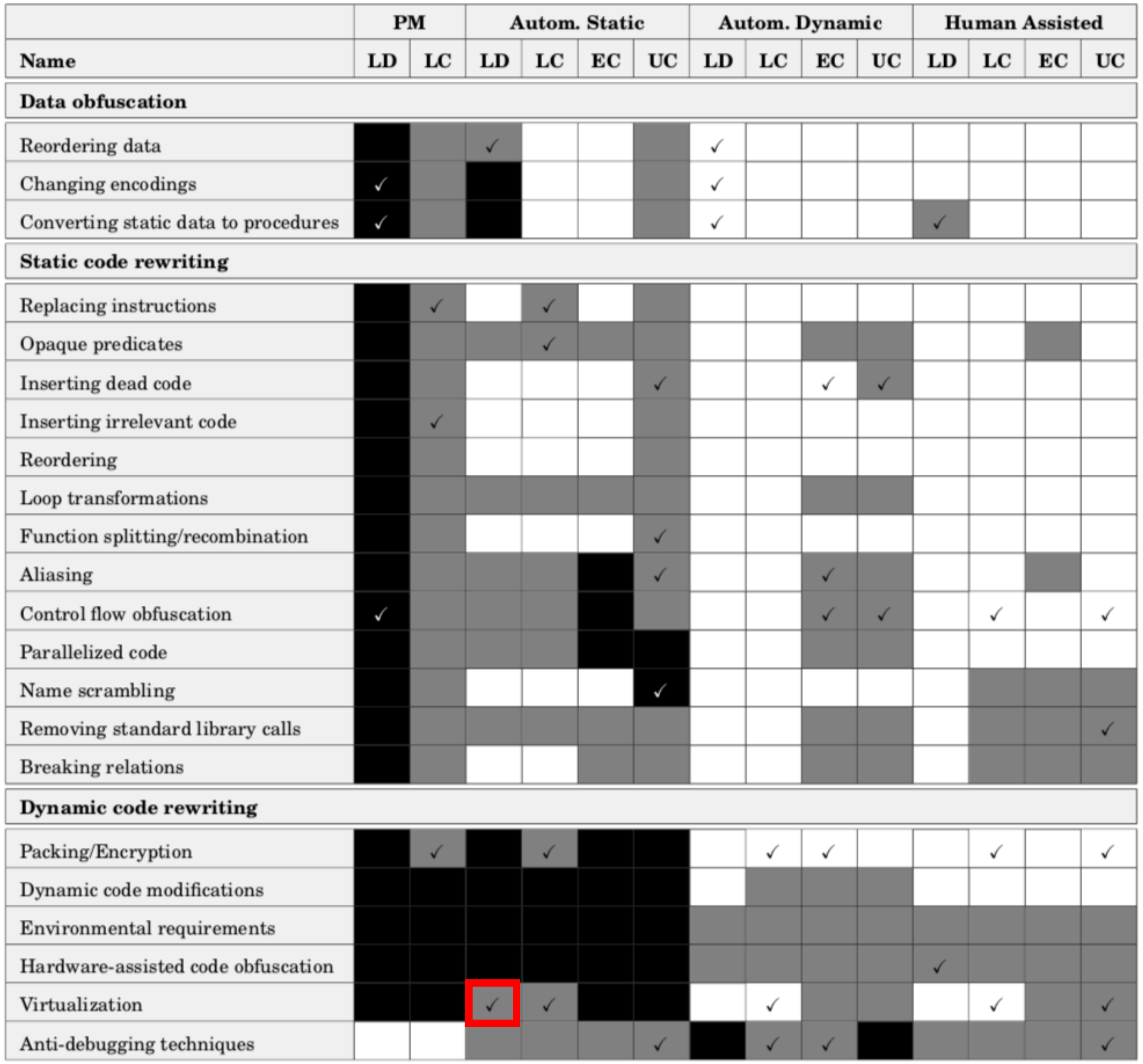
ACM Reference Format:
Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl, 2015. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (April 2016), 40 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The development of code obfuscation techniques is driven by the desire to hide the specific implementation of a program from automatic or human-assisted analysis of exe-

The research was funded under Grant 826461 (FIT-IT) and COMET K1 by the FFG – Austrian Research Promotion Agency, the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development, CASED, and CRISP. Author's addresses: S. Schrittwieser, Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks, Department Computer Science & Security, St. Pölten University of Applied Sciences, Austria; S. Katzenbeisser, Security Engineering Group, Technische Universität Darmstadt, Germany; J. Kinder, Department of Computer Science, Royal Holloway, University of London, United Kingdom; G. Merzdovnik, SBA Research, Vienna, Austria; Edgar Weippl, SBA Research, Vienna, Austria.

	Pattern Matching	Automated Static Analysis	Automated Dynamic Analysis	Human-assisted Analysis
Finding the location of data				
Finding the location of program functionality				
Extraction of code fragments				
Understanding the program				



	PM		Autom. Static				Autom. 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	■	▨	▨	▨	▨	▨			▨	▨				
Function splitting/recombination	■	▨				▨ ✓								
Aliasing	■	▨	▨	▨	■	▨ ✓			▨ ✓	▨			▨	
Control flow obfuscation	■ ✓	▨	▨	▨	■	▨			▨ ✓	▨ ✓		✓		✓
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			▨	▨	▨	▨ ✓	■	▨ ✓	▨ ✓	■	▨	▨	▨	▨ ✓

Example

- Kinder, J. Towards Static Analysis of Virtualization-Obfuscated Binaries. In Proceedings of the 19th Working Conf. Reverse Engineering (2012). IEEE, 61– 70.
- Lifting static analysis to a second dimension of location (“virtual program counter”)

Same analysis precision as on unobfuscated code is achievable

Evaluation on a toy examples only

Protection strength quantification

- Goal: Quantifying the strength of different obfuscations in a particular protection scenario to generate an efficient protection strategy
- Initial focus on potency and stealth
- Code complexity metrices
  - Originally created to help building reliable, readable, and maintainable software constructs
    - Attention: Less complex code is not necessarily more efficient
  - Often utilized for measuring obfuscation potency, i.e., how well humans can comprehend the code
    - e.g., Collberg et al. 1997

Halstead difficulty

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

```
main() {
  int a, b, sum;
  scanf("%d %d", &a, &b);
  sum = a+b;
  printf("sum: %d", sum);
}
```

Distinct operators (η1):

```
main
()
{}
int
scanf
&
=
+
printf
,
;
```

$\eta_1=11$

Distinct operands (η2):

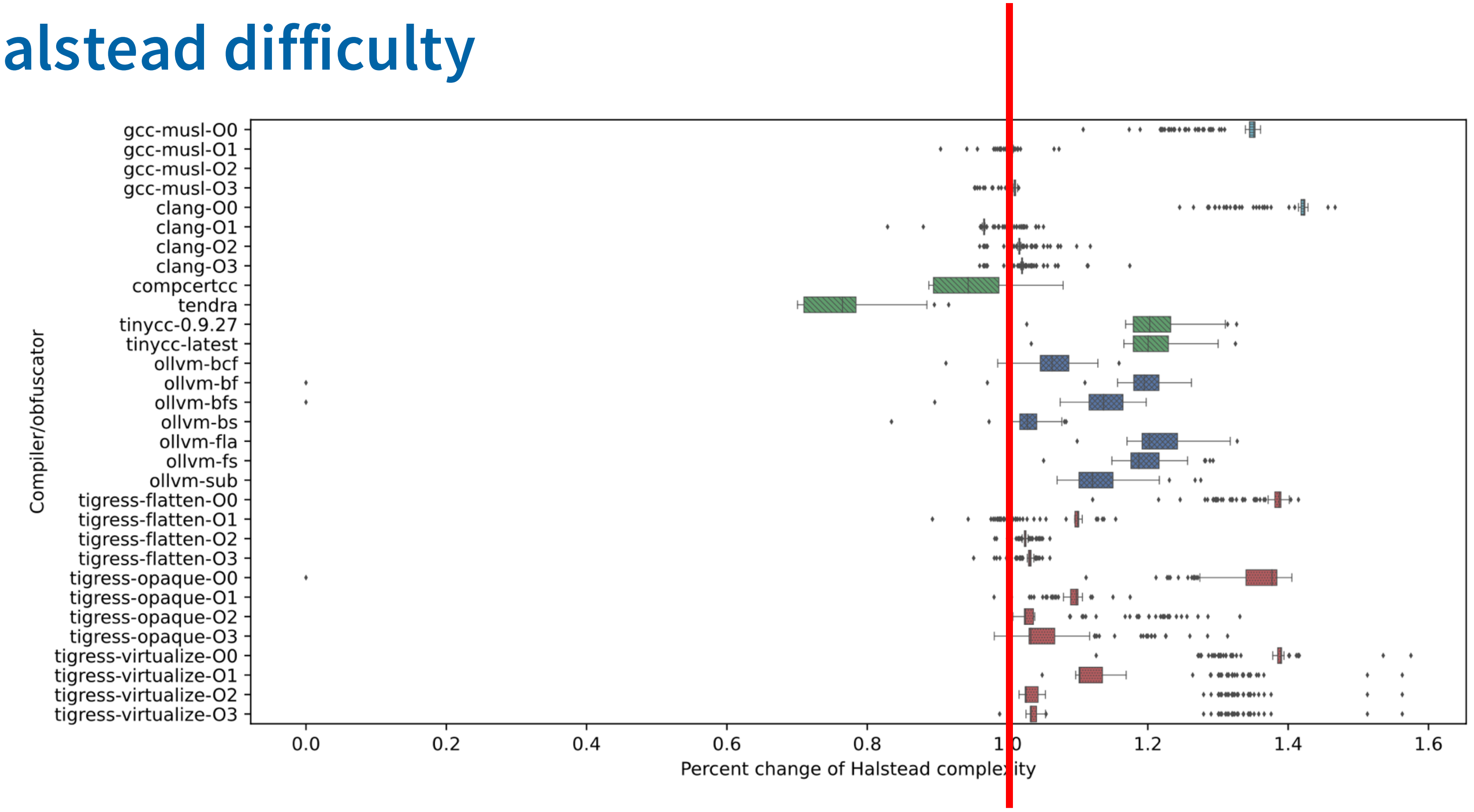
```
a
b
sum
"%d %d"
"sum: %d"
```

$\eta_2=5$

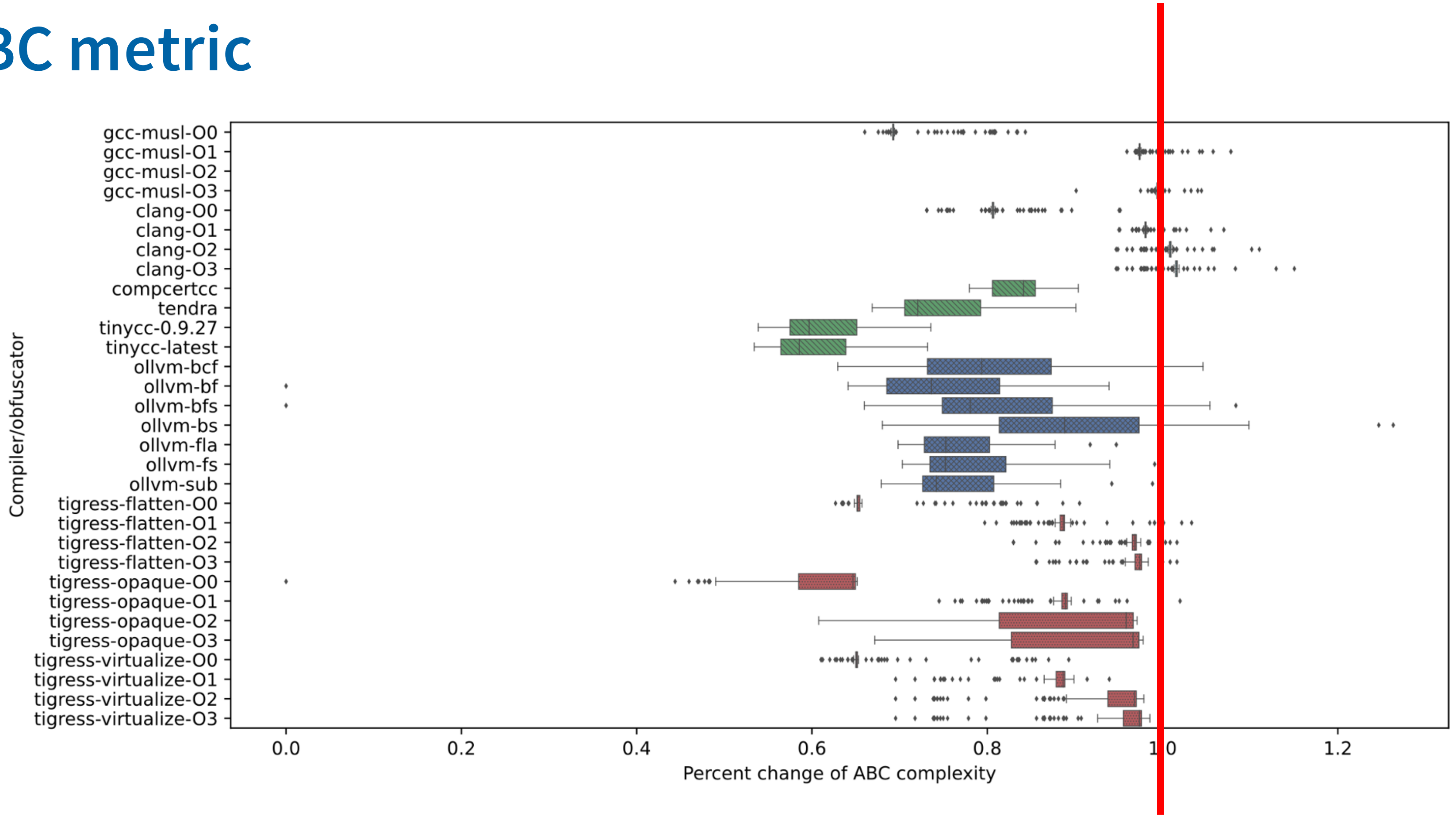
$N_2=11$

$D = 11/2 * 11/5 = 12{,}1$

Halstead difficulty

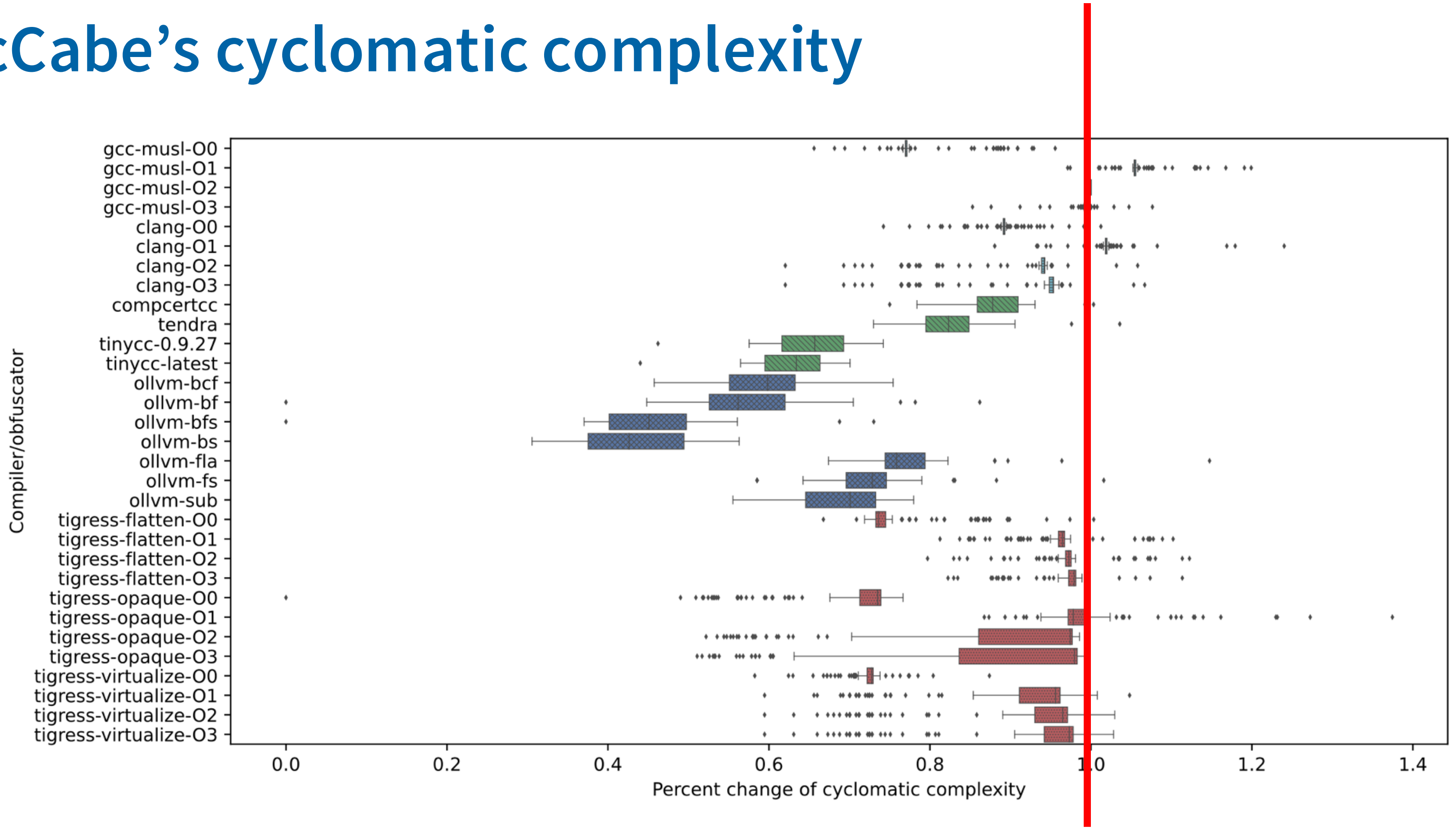


ABC metric

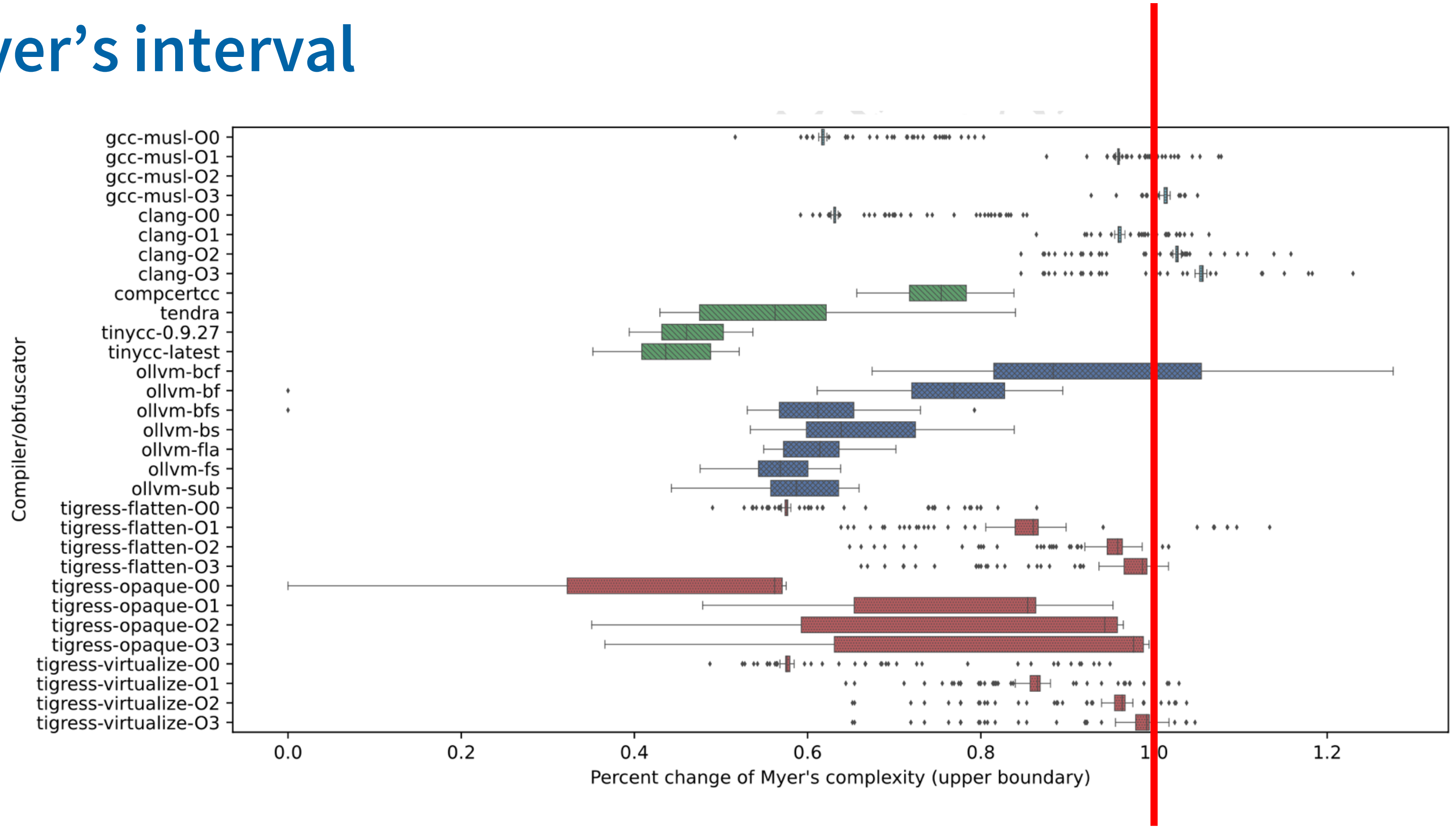


Compiler/obfuscator
gcc-musl-O0
gcc-musl-O1
gcc-musl-O2
gcc-musl-O3
clang-O0
clang-O1
clang-O2
clang-O3
compcertcc
tendra
tinycc-0.9.27
tinycc-latest
ollvm-bcf
ollvm-bf
ollvm-bfs
ollvm-bs
ollvm-fla
ollvm-fs
ollvm-sub
tigress-flatten-O0
tigress-flatten-O1
tigress-flatten-O2
tigress-flatten-O3
tigress-opaque-O0
tigress-opaque-O1
tigress-opaque-O2
tigress-opaque-O3
tigress-virtualize-O0
tigress-virtualize-O1
tigress-virtualize-O2
tigress-virtualize-O3
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Percent change of ABC complexity

McCabe’s cyclomatic complexity



Myer’s interval

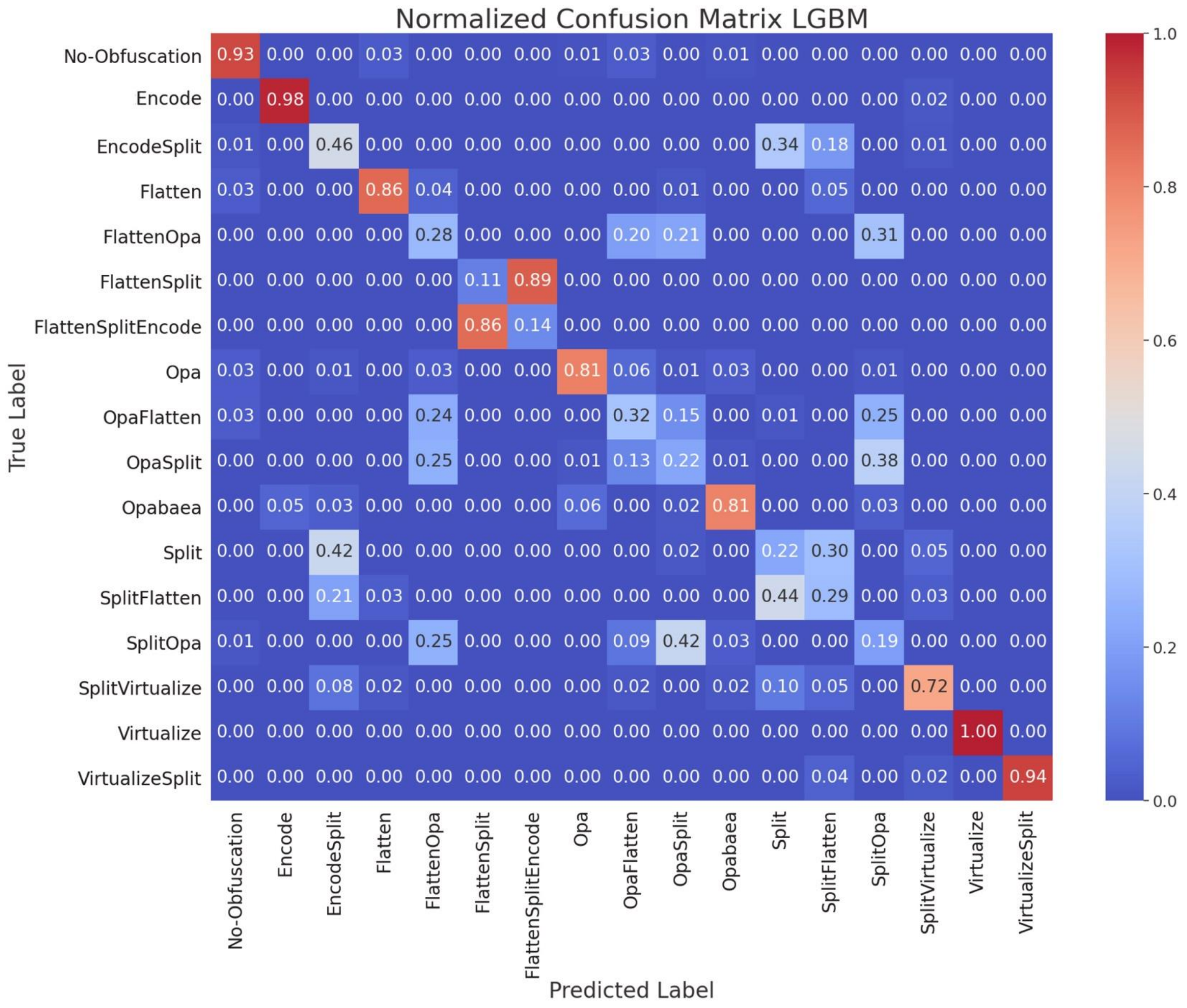


Structural patterns in (protected) binary code

- Obfuscations create characteristic patterns in a set of complexity measures
  - Example: CFG flattening
    - Reduced cyclomatic complexity and Myer's interval
    - Increased Halstead difficulty
- Obfuscation layering
  - Which obfuscations and combinations of obfuscations generate similar patterns?
    - CFG flattening and virtualization
  - Does the last obfuscation applied to a program have the biggest impact on the observed code complexity pattern?

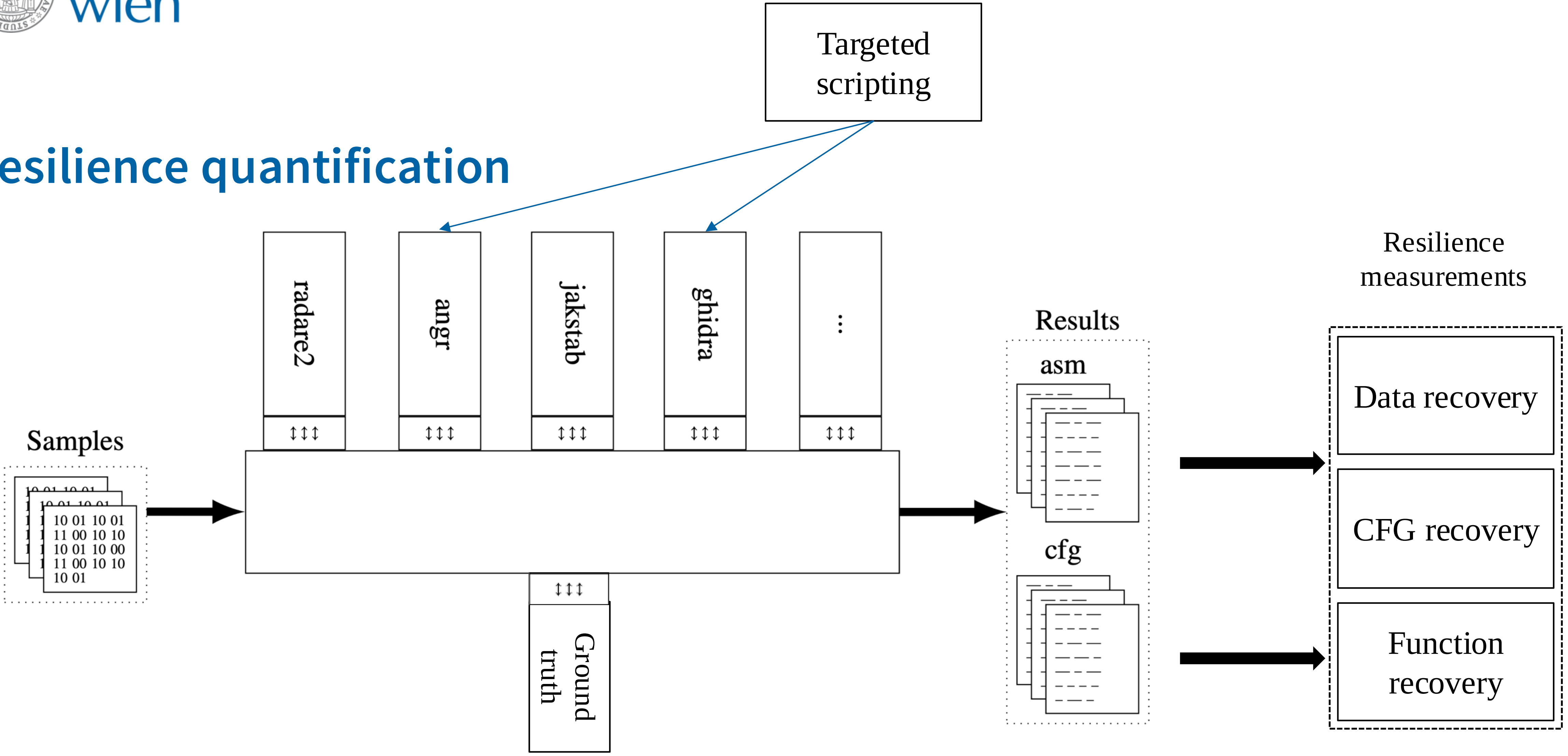
Obfuscation stealth model

- We measured the effects of obfuscation layering on the structure of binary code.
- We evaluated if individual obfuscations cover the structural patterns of others
  - 85 C programs
  - 80 different build configurations
  - A total of 6211 samples



	No-Obfuscation	Encode	EncodeSplit	Flatten	FlattenOpa	FlattenSplit	FlattenSplitEncode	Opa	OpaFlatten	OpaSplit	Opabaea	Split	SplitFlatten	SplitOpa	SplitVirtualize	Virtualize	VirtualizeSplit
No-Obfuscation	0.93	0.00	0.00	0.03	0.00	0.00	0.00	0.01	0.03	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00
Encode	0.00	0.98	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00
EncodeSplit	0.01	0.00	0.46	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.34	0.18	0.00	0.01	0.00	0.00
Flatten	0.03	0.00	0.00	0.86	0.04	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.05	0.00	0.00	0.00	0.00
FlattenOpa	0.00	0.00	0.00	0.00	0.28	0.00	0.00	0.00	0.20	0.21	0.00	0.00	0.00	0.31	0.00	0.00	0.00
FlattenSplit	0.00	0.00	0.00	0.00	0.00	0.11	0.89	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
FlattenSplitEncode	0.00	0.00	0.00	0.00	0.00	0.86	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Opa	0.03	0.00	0.01	0.00	0.03	0.00	0.00	0.81	0.06	0.01	0.03	0.00	0.00	0.01	0.00	0.00	0.00
OpaFlatten	0.03	0.00	0.00	0.00	0.24	0.00	0.00	0.00	0.32	0.15	0.00	0.01	0.00	0.25	0.00	0.00	0.00
OpaSplit	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.01	0.13	0.22	0.01	0.00	0.00	0.38	0.00	0.00	0.00
Opabaea	0.00	0.05	0.03	0.00	0.00	0.00	0.00	0.06	0.00	0.02	0.81	0.00	0.00	0.03	0.00	0.00	0.00
Split	0.00	0.00	0.42	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.22	0.30	0.00	0.05	0.00	0.00
SplitFlatten	0.00	0.00	0.21	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.44	0.29	0.00	0.03	0.00	0.00
SplitOpa	0.01	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.09	0.42	0.03	0.00	0.00	0.19	0.00	0.00	0.00
SplitVirtualize	0.00	0.00	0.08	0.02	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.10	0.05	0.00	0.72	0.00	0.00
Virtualize	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
VirtualizeSplit	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.02	0.00	0.94

Resilience quantification



Conclusions

- Software protections are heavily used in the industry
  - Gaming, commercial software, malware
- They typically add significant overheads to a program (runtime performance, memory usage, etc.)
- Still, software protections are often applied with little assessment of their effectiveness
- Research towards more sustainable use of software protections
  - Quantification of the protection strength
    - Potency, resilience, and stealth
  - Targeting concrete protection scenarios

Thank you for your attention!

sebastian.schrittwieser@univie.ac.at