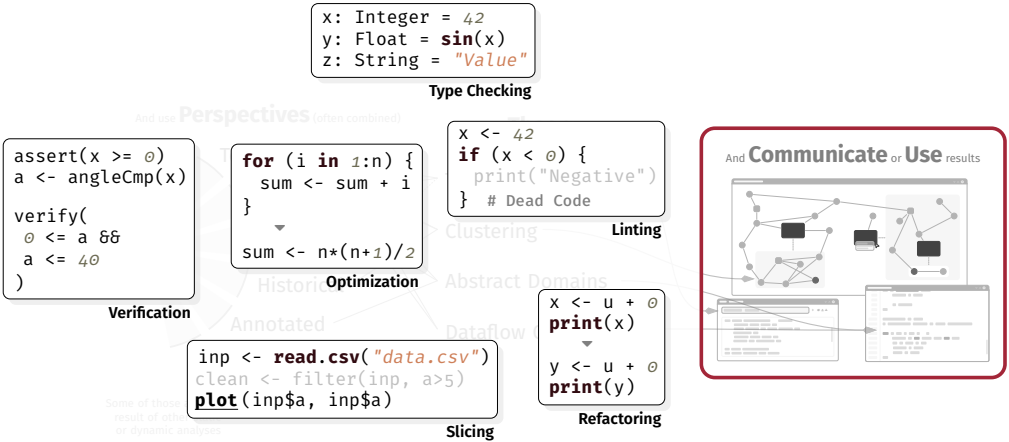


A Primer on Static Code Analysis

Software Quality Assurance — Static Code Analysis, I | Florian Sihler | December 3, 2025

Software Engineering
Programming Languages

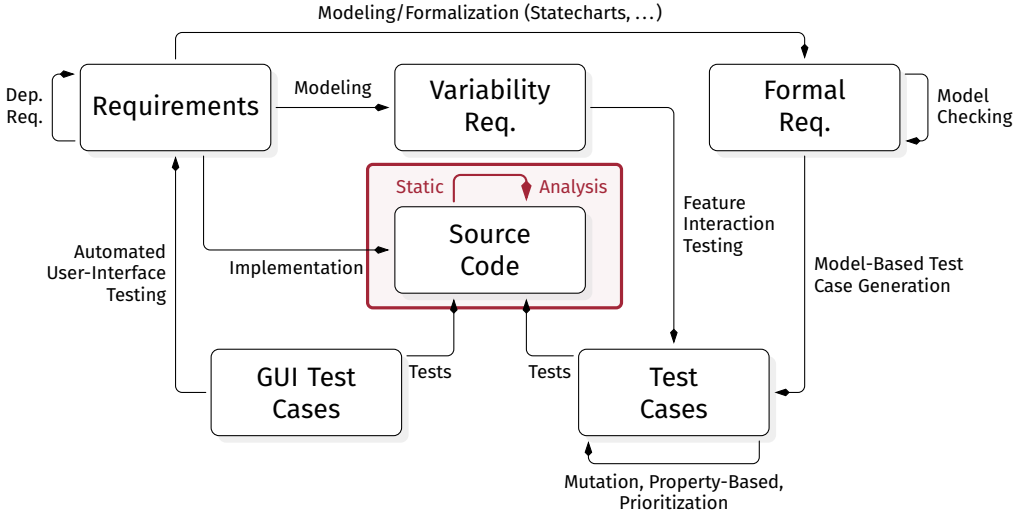
universität ulm

Outline

1. A first Overview
2. Linting Origins
3. More Sophisticated Fronts
4. Important Terminology
5. Outlook

1. A first Overview

Embedding a Landscape



What is static code analysis?

Derivable purely on program syntax.
e.g., Line-Counts, Indentation, Parameter-Counts, …

Derivable from program semantics (behavior).
e.g., Possible Variable Values, Termination, …

Discover *syntactic/semantic properties* of programs **without** running them.[RY20]

Reason on *all* possible executions.

Why do static analysis?

- Find bugs or vulnerabilities
- Prove correctness
- Find and apply optimizations
- Follow coding guidelines / style guides
- Refactoring support
- … (it is fun!)

Let’s find some bugs!

Edsger W. Dijkstra (1930–2002)
Communications of the ACM

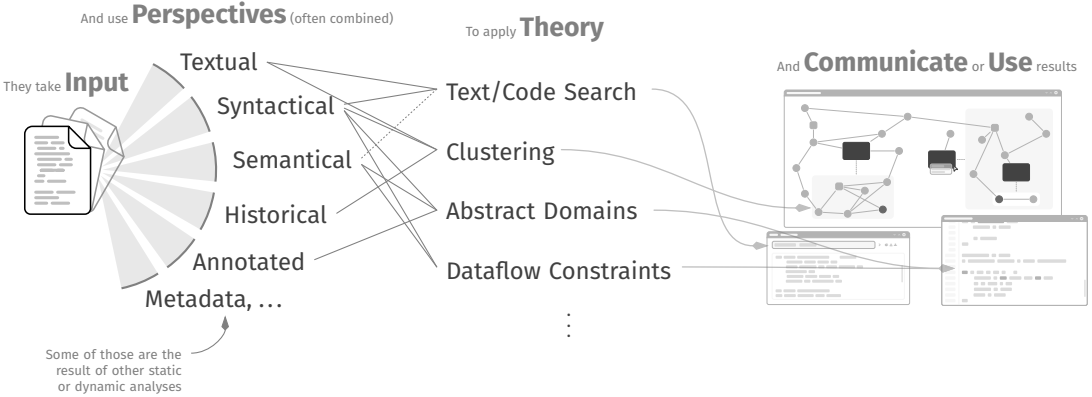
Easy
Dynamic
Practical
Quick
System.out.println?
Nope

But,
Tests are
Unsound!

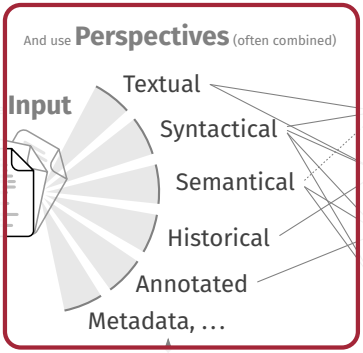
“Program testing can be used to show the presence of bugs, but never to show their absence!”

Dijkstra[Dij69]

A First Look

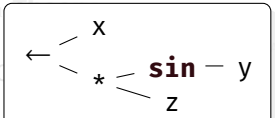

And use Perspectives (often combined)
To apply Theory
They take Input
Textual
Syntactical
Semantical
Historical
Annotated
Metadata, …
Some of those are the result of other static or dynamic analyses
Text/Code Search
Clustering
Abstract Domains
Dataflow Constraints
And Communicate or Use results

Perspectives

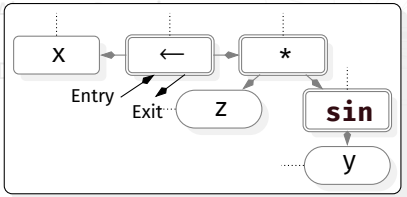



```
x ← sin(y) * z
```

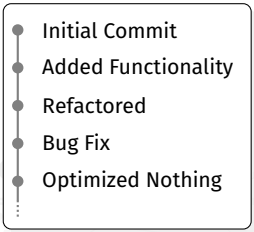
Textual Perspective



Syntactical Perspective (AST)



Semantical Perspective (CFG & DFG)



Historical Perspective

```
# Caused CVE 2025-12345
x ← sin(y)
```

Annotated Perspective

Author: Jane Doe
Code Review: Passed
…

Metadata Perspective

Theory

```
(assignment_expression
  left: (member_expression
    object: (call_expression)
))
```

Searching for Patterns

```
r <- read.csv("data.csv")          load
r <- filter(r, value > 10)         transf.
plot(r$time, r$value)              vis.
lines(r$time, r$score)
```

Clustering

To apply **Theory**

- Text/Code Search
- Clustering
- Abstract Domains
- Dataflow Constraints
- ⋮

```
x ← 2 * rnd(0,1)      x ∈ [0,2]
y ← x + 1             y ∈ [1,3]
z ← y + abs(u)        z ∈ [1,∞]
```

$$\prod_k [\ell_k .. h_k] = [\max(\ell_k) .. \min(h_k)]$$

Abstract Domains

$$\mathbf{In}_n = (\mathbf{Out}_n - \mathbf{Kill}_n) \cup \mathbf{Gen}_n$$

$$\mathbf{Out}_n = \begin{cases} \mathbf{BI} & n \text{ is End} \\ \bigcup_{m \in \mathrm{succ}(n)} \mathbf{In}_m & \text{else} \end{cases}$$

Dataflow Constraints

Application

```
x: Integer = 42
y: Float = sin(x)
z: String = "Value"
```

Type Checking

```
assert(x >= 0)
a ← angleCmp(x)

verify(
 0 <= a &&
 a <= 40
)
```

Verification

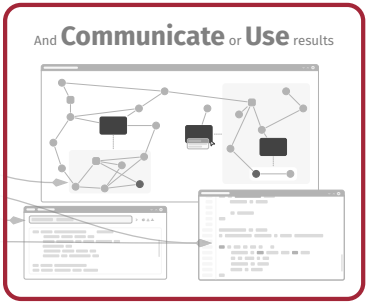
```
for (i in 1:n) {
  sum ← sum + i
}

sum ← n*(n+1)/2
```

Optimization

```
x ← 42
if (x < 0) {
  print("Negative")
} # Dead Code
```

Linting




```
x ← u + 0
print(x)

y ← u + 0
print(y)
```

Refactoring

```
inp ← read.csv("data.csv")
clean <- filter(inp, a>5)
plot(inp$a, inp$a)
```

Slicing

2. Linting Origins

A User-Driven History: Linting

- “Lint” by Stephen C. Johnson (1978, Bell Labs)
  - Lints (ger. “Fusseln”) as “small, unwanted bits”
  - Created to debug a *yacc* grammar for *C*
    - Also created by Johnson
- Linting is omni-present today
- Today, many aspects are integrated in compilers
  - Identification of dead code
  - Unused variables or functions
  - Possible null dereferences
  - …

Stephen C. Johnson (1944)

Which linters do you know?
⟹ Part 3

A Word on “Linting”

Linting is often used as a generic term!

Whether something is a linter, type checker, verifier, …
usually depends on the perspective of the creator.

In the following, we explore some common linting tasks.

Style Checks

Check compliance with coding style guidelines

Dennis M. Ritchie (1941)
National Inventors HoF

Brian W. Kernighan (1942)
Faces of OS

- For example, standardized with the C[KR88]
- Goal:
  - Improve code readability for maintenance
  - Naming conventions, indentation, brace positions, …
  - E.g., avoidance of “magic numbers”
- Style guides can catch “simple” bugs early:
  - Wrong nesting for conditionals (forcing braces)
  - Missing *switch* cases (forcing *default*)
  - Require non-empty *catch* blocks
  - …

Style guides can be, language, organization, domain, or even project specific!

Good Code, Bad Code

```java
int timeToReadBook = (220 * 10) / 60;
```

Apples? Oranges?

```java
final int PAGES = 220;
final int MINUTES_PER_PAGE = 10;
final int MINUTES_PER_HOUR = 60;

int timeToReadBook = (PAGES * MINUTES_PER_PAGE) /
                     MINUTES_PER_HOUR;
```

And Besides Synthetical Examples?

libsecurity_ssl/lib/sslKeyExchange.c

C

```
574 static OSStatus
575 SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
576                                  uint8_t *signature, UInt16 signatureLen)
577 {
    // ...
620     if ((err = SSLFreeBuffer(&hashCtx)) != 0)
621         goto fail;
622
623     if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
624         goto fail;
625     if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
626         goto fail;
627     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
628         goto fail;
629     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
630         goto fail;
631         goto fail;
632     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
633         goto fail;
    // ...
647 fail:
648     SSLFreeBuffer(&signedHashes);
649     SSLFreeBuffer(&hashCtx);
650     return err;
651
652 }
```

Wellp!

Now, *err* can be 0, signaling a valid signature!

http://support.apple.com/kb/HT6147

Style Checks: Summary

- How to do style checks?
  - Mostly through Regular Expressions or AST pattern matching
  - Many also offer fixes/suggestions
  - For example (with ast-grep, slightly simplified):

yaml

```yaml
rules:
  - id: add-if-braces
    pattern: if ($COND) $BODY
    constraints: $BODY: not:
          kind: block
    rewrite: if ($COND) { $BODY }
```

- Usually, they are heavily configurable to fit project needs
- Likewise, they are usually only syntactic, without semantic info
  (although many offer more features)
- Examples: checkstyle, pylint, ESLint Sylistic, …

Please consider using `.git-blame-ignore-revs`.

Unconventional Patterns and Bugs

```
for (int i=1; i<=3; i++);
  f(i);
```

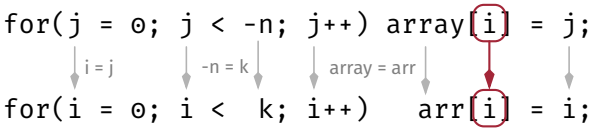
```
for(int i=0; j<n; i++) { ... }
```

```
int foo(int a, int b) { ... }
foo(b, a)
```

```
if (a == NULL) { compute(a) }
```

- Various Categories:
  - Single-Threaded corr.
  - Thread/Synch. corr.
  - Performance issues
  - Security vulnerabilities
  - …
- Various Techniques:
  - State Machine on bytecode
  - Data Flow analyses
  - Abstract interpretation
  - …

Error Mining



- We try to find code that has been *almost* written like this!
- If many do something similar, but one does it differently, it might be a bug
- Usually relies on AST patterns with minimal binding information

See also:

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy (Association for Computing Machinery, 2004)
[Bre12] Alexander Breckel. Error mining: Bug detection through comparison with large code databases (MSR, 2012)

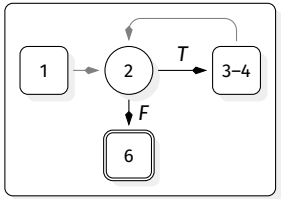
3. More Sophisticated Fronts

A Best Friend for Life: Control Flow Graph (CFG) [CT22]

Control Flow Graph (CFG)
Directed graph representing all possible execution paths of a program. Usually uses *basic blocks* as nodes.

Basic Block
A sequence of instructions with a single entry point (no jumps in) and a single exit point (no jumps out except at the end).

```java
int a = 42;
while(a > 0) {
  a = a + 1;
  a = a / 2;
}
System.out.println(a);
```



CFG

[CT22] K.D. Cooper and L. Torczon. Engineering a Compiler (Elsevier Science, 2022)

Exercise: Build a CFG

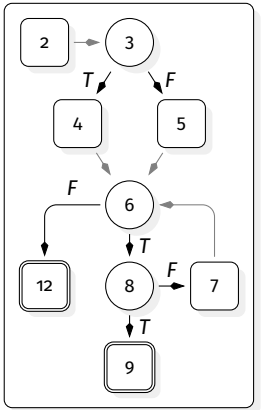
```java
static int gcdBad(int a, int b) {
  int i;
  if (a < b)
      { i = a; }
  else { i = b; }
  for (;i > 1;
         i--) {
    if (a % i == 0 && b % i == 0) {
      return i;
    }
  }
  return 1;
}
```

- How to find dead code?
- How to find infinite loops?

Exercise: Build a CFG

```java
static int gcdBad(int a, int b) {
  int i;
  if (a < b)
      { i = a; }
  else { i = b; }
  for (;i > 1;
         i--) {
    if (a % i == 0 && b % i == 0) {
      return i;
    }
  }
  return 1;
}
```

- How to find dead code?
  E.g., nodes not reachable from the start node.
- How to find infinite loops?
  E.g., nodes that cannot reach an exit node.



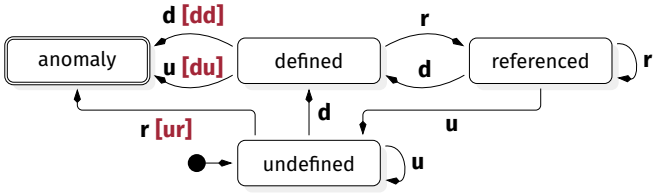
CFG

Using the CFG: Find Dataflow Anomalies

- For every variable, we record the following actions:
  - **d** Definition (write)
  - **r** Reference (read)
  - **u** Undefined (before write, end of program, …)

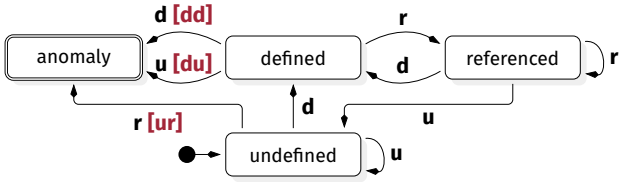
```
x = 42
print(x)
int x
```

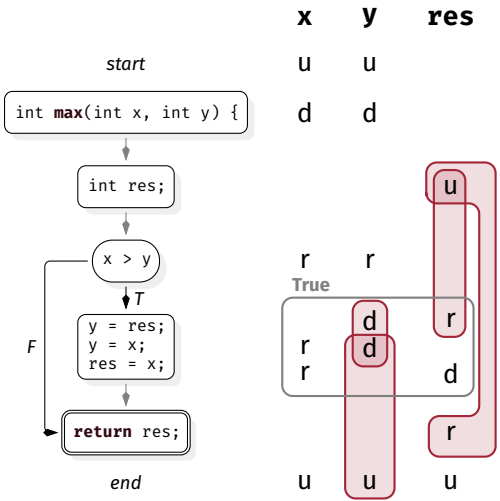
- Following the CFG
  - we track the state of each variable,
  - iterate on control-flow cycles until reaching a fixed point,
  - go statement-by-statement for basic blocks,
  - and report anomalies.



Anomalies? ?

Let's Run an Example!





- **[dd]**: Unnecessary definition
- **[du]**: Definition without ref.
- **[ur]**: Ref. without prior definition
- Consider all paths through the CFG!

Reality Is *more* Complex, Ain't It?

- Languages possess many features
  - Concurrency[GW21]
  - Reflection[AM21]
  - Dynamic loading[Thé+24]
  - Native calls[Rot+24]
  - Exceptions[Jo+04]
  - …
- These features usually intertwine with each other
- A plethora of techniques try to address these challenges from various perspectives
- One prominent technique is *abstract interpretation*[Cou21]
  We'll tackle this in the second lecture!

4. Important Terminology

Let's Return to the Roots

Discover *syntactic/semantic properties* of programs
without running them.

- Depending on our focus, we are interested in:
  - Satisfying these properties (**verification**)
    E.g., x is always non-negative
  - Violating these properties (**bug finding**)
    E.g., possible division by zero

Liveness and Safety Properties [RY20; Lam77]

Leslie B. Lamport (1941)
Heidelberg Laureate Forum

- **Liveness properties:** (“good things”, infinite time)
  Things that *must* happen during execution. *For example,*
  - Every request eventually receives a response.
  - Every thread that is started eventually terminates.
  - …

- **Safety properties:** (“bad things”, finite time)
  Things that *must not* happen during execution. *For example,*
  - Integer variables never overflow.
  - Array accesses are always within bounds.
  - …

Rice's Theorem

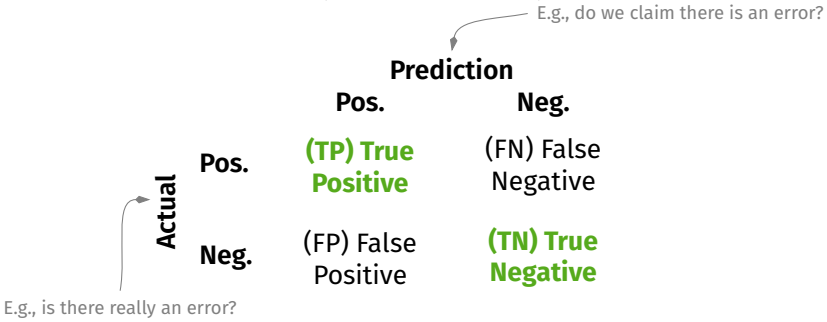
Henry Gordon Rice (1920–2003)
Medium

- We want to prove properties of programs (e.g., no overflow, shapes, …)
- However, thanks to Rice [Ric53] we know:

  Rice's theorem states that all nontrivial semantic properties of programs are undecidable. [Cou21, p. 100]

- We can not solve the halting problem
- We have to approximate the reality

The Confusion Matrix (Reminder)



- **Precision:** **TP**/(**TP** + FP) (“how many false alarms”)
- **Recall:** **TP**/(**TP** + FN) (“how many errors did we find”)

Soundness and Completeness

Soundness (“Correct Over-Approximation”)

- All properties we derive are true (but we may miss some)
- If we report bugs for violated properties, we produce no false negative

Completeness (“Correct Under-Approximation”)

- We are able to infer all interesting properties in the program
- If we report bugs for violated properties, we produce no false positive

What is Sound, What is Complete?

Testing

Finite Model Checking

Static Analysis

- What is more important? Depends on what you want!
  - Make sure the auto-pilot cannot steer directly down?
  - Need to show your website can be viewed?

What is Sound, What is Complete?

Testing

Complete, but unsound.

We find definitely reachable states, but (generally) not all!

Finite Model Checking

Complete and sound.

Explore all states in a finite model, but the model may wrongly represent reality.

Static Analysis

Sound, but incomplete.

We find all runtime states in an abstract model, but may over-approximate.

- What is more important? Depends on what you want!
  - Make sure the auto-pilot cannot steer directly down?
    Soundness is key.
  - Need to show your website can be viewed?
    You better want completeness.

5. Outlook

A Couple of Questions to Round It Of

1. How would you capture what a *property* is?
2. How would you phrase that one property is “better” than another?
3. For what operations would you *not* use a control-flow graph?
4. Why can’t there be a fully automatic, sound, and complete static analyzer for general programs?
5. What (big) additional challenges do you see in the real-world?

We will address these questions in the next lecture(s)!

References I

[AM21] Vincenzo Arceri and Isabella Mastroeni. “Analyzing Dynamic Code: A Sound Abstract Interpreter for Evil Eval”. In: *ACM Trans. Priv. Secur.* 24.2 (Jan. 2021). ISSN: 2471-2566. DOI: 10.1145/3426470. URL: https://doi.org/10.1145/3426470.

[Bre12] Alexander Breckel. “Error mining: Bug detection through comparison with large code databases”. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. MSR, 2012, pp. 175–178. DOI: 10.1109/MSR.2012.6224278.

[Cou21] Patrick Cousot. “Principles of Abstract Interpretation”. In: (2021).

[CT22] K.D. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2022. ISBN: 9780128154120.

[Dij69] EW Dijkstra. “Notes on Structured Programming (EWD249)”. In: *Technical V. Eindhoven, The Netherlands* (1969).

[GW21] Damian Giebas and Rafał Wojszczyk. “Detection of Concurrency Errors in Multithreaded Applications Based on Static Source Code Analysis”. In: *IEEE Access* 9 (2021), pp. 61298–61323. DOI: 10.1109/ACCESS.2021.3073859.

[HP04] David Hovemeyer and William Pugh. “Finding bugs is easy”. In: *SIGPLAN Not.* 39.12 (Dec. 2004), pp. 92–106. ISSN: 0362-1340. DOI: 10.1145/1052883.1052895. URL: https://doi.org/10.1145/1052883.1052895.

[Jo+04] Jang-Wu Jo et al. “An uncaught exception analysis for Java”. In: *Journal of Systems and Software* 72.1 (2004), pp. 59–69. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(03)00057-8. URL: https://www.sciencedirect.com/science/article/pii/S0164121203000578.

[KR88] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Educación, 1988.

[Lam77] Leslie Lamport. “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[Ric53] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[Rot+24] Tobias Roth et al. “AXA: Cross-Language Analysis through Integration of Single-Language Analyses”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1195–1205. ISBN: 9798400712487. DOI: 10.1145/3691620.3696193. URL: https://doi.org/10.1145/3691620.3696193.

[RY20] Xavier Rival and Kwangkeun Yi. “Introduction to Static Analysis: An Abstract Interpretation Perspective”. In: (2020).

[Thé+24] Gaspard Thévenon et al. “B-Side: Binary-Level Static System Call Identification”. In: *Proceedings of the 25th International Middleware Conference*. Middleware ’24. Hong Kong, Hong Kong: Association for Computing Machinery, 2024, pp. 225–237. ISBN: 9798400706233. DOI: 10.1145/3652892.3700761. URL: https://doi.org/10.1145/3652892.3700761.