Dynamic Symbolic Execution:
Between Testing and Verification

Cristian Cadar

SOFTWARE RELIABILITY GROUP

Imperial College London

Funded by Engineering and Physical Sciences Research Council; erc European Research Council

Image credits: Powerpoint, Bing, jackmac34, Gam-Ol, Walkerssk @ Pixabay

Keynote @ VSTTE'24
Prague, Czechia, 14 October 2024

VERIFIED SOFTWARE

seL4: Formal Verification of an OS Kernel

Gerwin Klein[1,2], Kevin Elphinstone[1,2], Gernot Heiser[1,2,3]
June Andronick[1,2], David Cock[1], Philip Derrin[1*], Dhammika Elkaduwe[1,2‡] Kai Engelhardt[1,2]
Rafal Kolanski[1,2], Michael Norrish[1,4], Thomas Sewell[1], Harvey Tuch[1,2†], Simon Winwood[1,2]
[1] NICTA, [2] UNSW, [3] Open Kernel Labs, [4] ANU
ertos@nicta.com.au

Formal verification of a realistic compiler

Xavier Leroy
INRIA Paris-Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
xavier.leroy@inria.fr

Establishing Browser Security Guarantees through Formal Shim Verification

Dongseok Jang, *UC San Diego*
Zachary Tatlock, *UC San Diego*
Sorin Lerner, *UC San Diego*

Industrial hardware and software verification with ACL2

Warren A. Hunt Jr[1], Matt Kaufmann[1], J Strother Moore[1] and Anna Slobodova[2]

Implementing TLS with Verified Cryptographic Security

Karthikeyan Bhargavan*, Cédric Fournet†, Markulf Kohlweiss†, Alfredo Pironti*, Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt, {karthikeyan.bhargavan,alfredo.pironti}@inria.fr
†Microsoft Research, {fournet,markulf}@microsoft.com
‡IMDEA Software, pierre-yves@strub.nu

Using Crash Hoare Logic for Certifying the FSCQ File System

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

ORIENTAIS: Formal Verified OSEK/VDX Real-Time Operating System

Jianqi Shi, Jifeng He, Huibiao Zhu, Huixing Fang, Yanhong Huang
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, P. R. China
Email: {jqshi,jifeng,hbzhu,wxfang,yhhuang}@sei.ecnu.edu.cn

Xiaoxian Zhang
iSoft Infrastructure Software CO., LTD.
Shanghai, P. R. China
Email: alex.zhang@i-soft.com.cn

Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System

Jean Yang
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory

Chris Hawblitzel
Microsoft Research

VERIFIED SOFTWARE

LLVM
COMPILER INFRASTRUCTURE

SQLite

OpenSSL
Cryptography and SSL/TLS Toolkit

git

APACHE
HTTP SERVER PROJECT

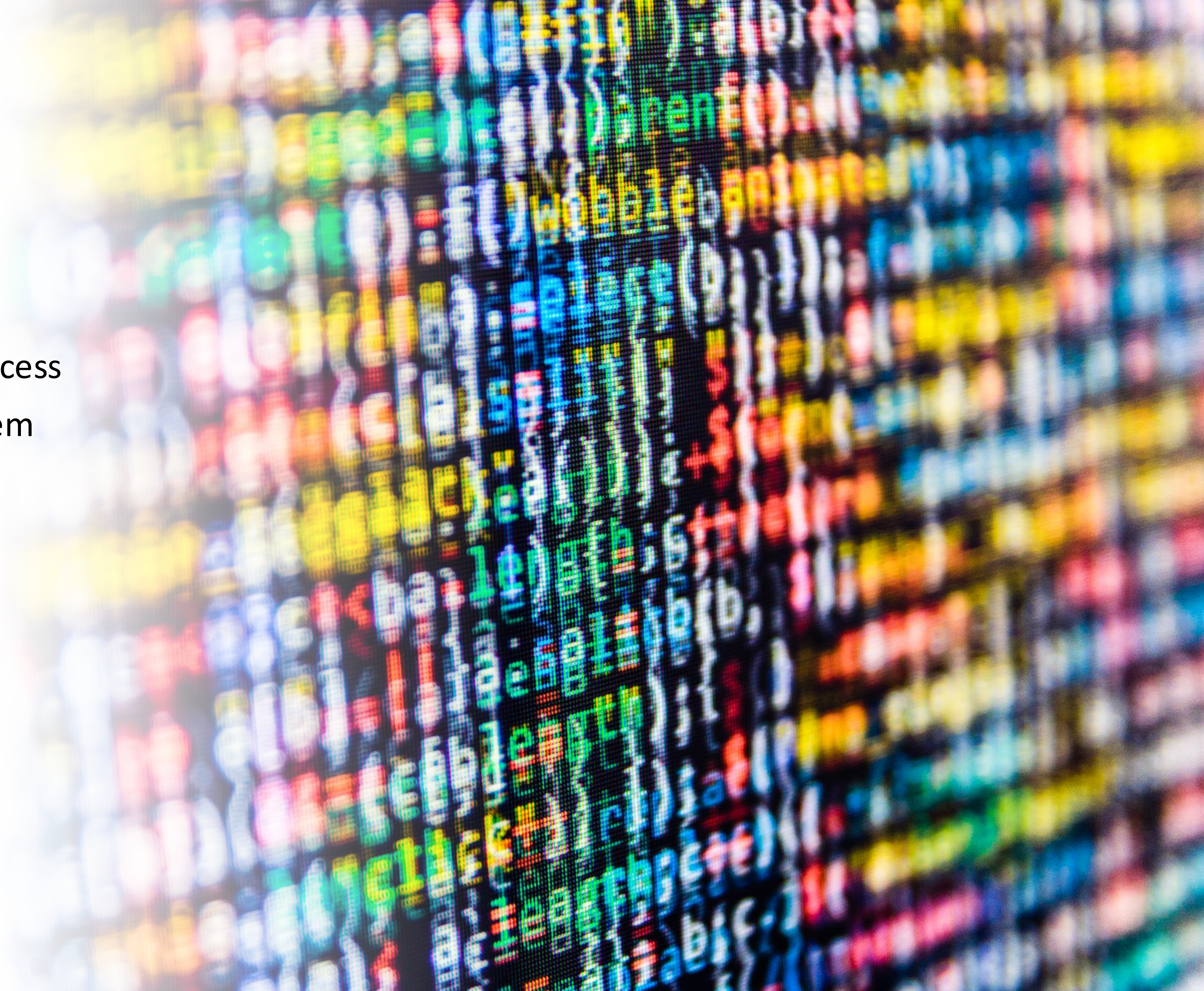
Office

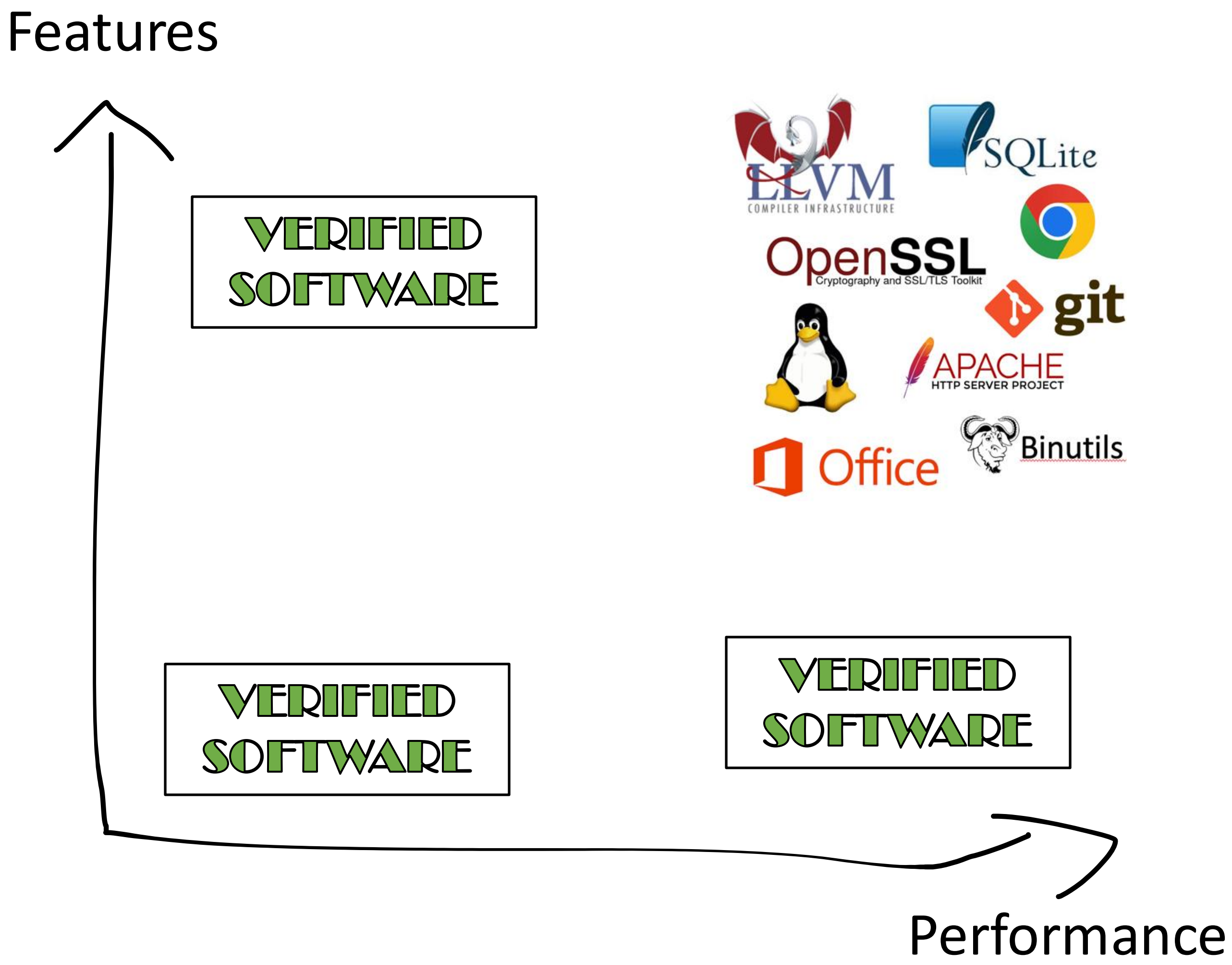
Binutils


MIND THE GAP

Complexity

- Complexity of code
- Complexity of specification
- Complexity of verification process
- Difficulty of evolving the system




Features
VERIFIED SOFTWARE
VERIFIED SOFTWARE
VERIFIED SOFTWARE
LLVM
COMPILER INFRASTRUCTURE
SQLite
OpenSSL
Cryptography and SSL/TLS Toolkit
git
APACHE
HTTP SERVER PROJECT
Office
Binutils
Performance

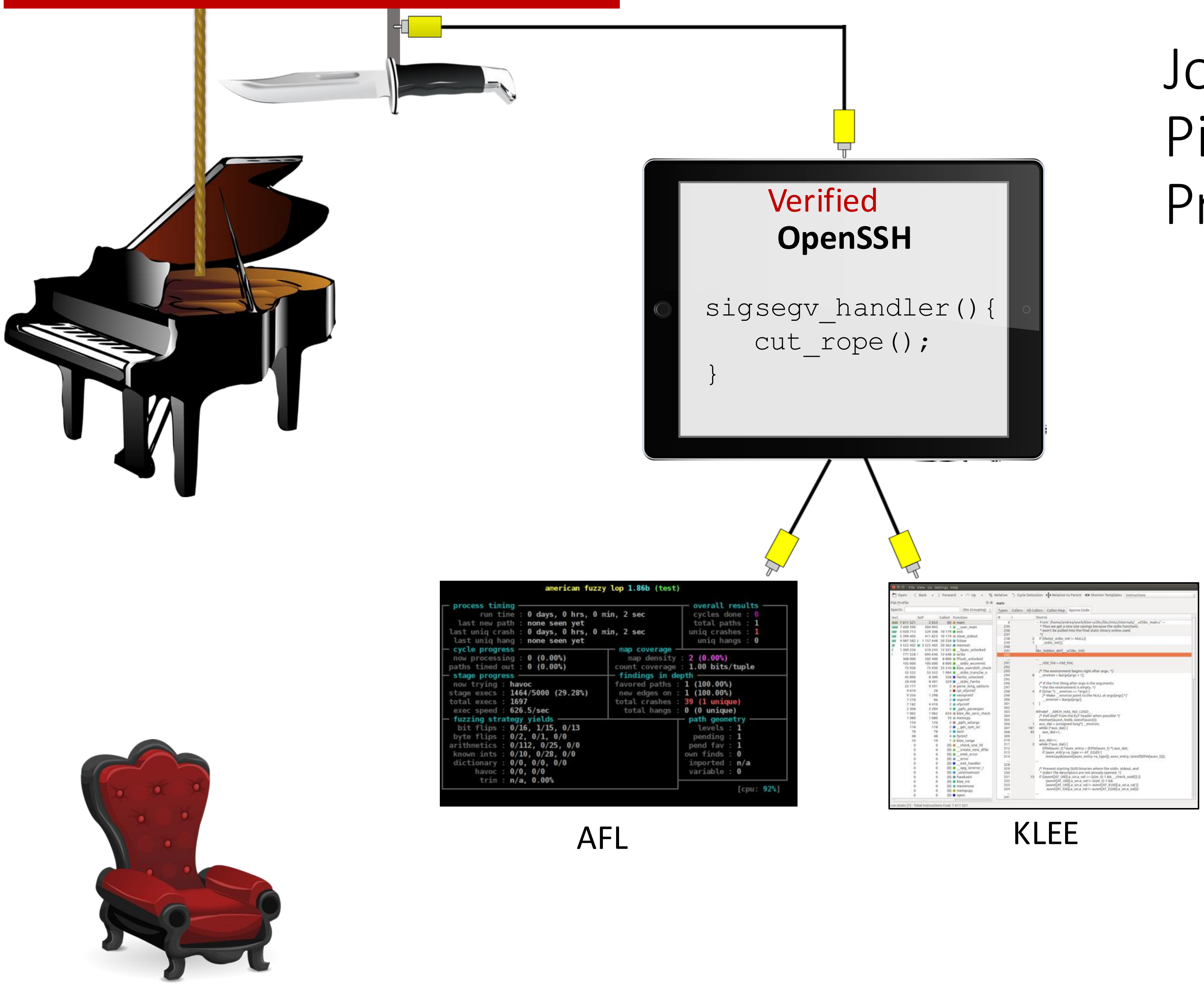
Donald Knuth -- Notes on Priority Deques, 1977

```
procedure insert2 (integer x, ℓ)
begin B[ℓ] ← B[ℓ] ∨ (2 ↑ (x mod 16));
    size[ℓ] ← size[ℓ]+1;
    if x < least[ℓ] then least[ℓ] ← x
    else if x > greatest[ℓ] then greatest[ℓ] ← x;
    end;
```

The implementation of deletion would be similar. It is safe to use 0 and $2^{16}-1$ for $-\infty$ and $+\infty$.

Beware of bugs in the above code; I have only proved it correct, not tried it.

John Regehr's Piano Test for Program Verification



Assumptions

- Formalisation/model of code is correct
  - Model-based verification, incorrect specifications
- Programming language semantics are correctly encoded
  - Including subtle issues such as undefined, unspecified and implementation-defined behaviour
- Compiler, linker, operating system etc. are correct
  - Source-level verification
- Environment behaves in a certain way
  - E.g., input format, reliable network, unlimited resources
- Software obeys mathematical rules
  - E.g., **n + 1 > n** or n + x ≠ n, for x ≠ 0
- Verification tools are correct
  - Large complex systems, sometimes even closed-source
  - Machine-checked proofs not always available
- etc.

An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy
University of Washington
{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

. . .

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: IronFleet, Verdi, and Chapar. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our

. . .

Assumptions

- Every method, formal or informal, makes assumptions
- We should do a better job documenting them
- Could take some inspiration from threat models of security research

When the Software is Correct…

VERIFICATION >> TESTING

When the Software is Buggy…

VERIFICATION ≈ TESTING

"Software is likely correct" **VS** "Software is likely buggy"

Testing ⟷ **Verification**

Manual Testing | Blackbox Fuzzing | Greybox Fuzzing | . . . | **Dynamic Symbolic Execution** | . . . | Sound Static Analysis | Model Checking | Formal Verification

Presence of Bugs

Low(er) Effort

Absence of Bugs

High(er) Effort

Dynamic Symbolic Execution (DSE)

Program analysis technique for ***automatically exploring paths*** through a program

Applications in:
- Bug finding
- Test generation
- Vulnerability detection and exploitation
- Equivalence checking
- Debugging
- Program repair
- Bounded verification
- etc. etc.

Dynamic Symbolic Execution in Practice

- Introduced in the 70s, revived mid-2000 by the DART and EGT projects
- Significant interest in the last few years
- Many dynamic symbolic execution/concolic tools available as open-source:
  - KLEE, CREST, SPF, FuzzBall, Angr, SymCC, etc.
- Started to be explored and adopted by industry:
  - Microsoft, Fujitsu, Hitachi, Bloomberg, Intel, Google, NASA, Samsung, Baidu, etc.
  - SAGE from Microsoft found 1/3 of file fuzzing bugs during development of Win 7
  - KLEE widely used in both academia and industry

KLEE

https://klee-se.org/
https://github.com/klee/

Popular dynamic symbolic executor primarily developed and maintained at Imperial

Academic impact:

- ACM SIGOPS Hall of Fame Award and ACM CCS Test of Time Award
- 3.5K+ citations to original KLEE paper (OSDI 2008)
- From many different research communities: testing, verification, systems, software engineering, programming languages, security, etc.
- Many different systems using KLEE: AEG, Angelix , BugRedux , Cloud9, GKLEE, KleeNet, KLEE-UC, S2E, SemFix, etc.

Growing impact in industry:

- **Baidu**: [KLEE 2018], **Fujitsu**: [PPoPP 2012], [CAV 2013], [ICST 2015], [IEEE Software 2017], [KLEE 2018], **Google**: [2x KLEE 2021], **Hitachi**: [CPSNA 2014], [ISPA 2015], [EUC 2016], [KLEE 2021], **Intel**: [WOOT 2015], **NASA Ames**: [NFM 2014], **Samsung**: [2x KLEE 2018], **Trail of Bits** [https://blog.trailofbits.com/], **etc.**

Active user and developer base with 100+ contributors listed on GitHub, 500+ forks, 2500+ stars, 400+ mailing list subscribers, 400+ participants to KLEE Workshops, etc.

KLEE Workshop 2024

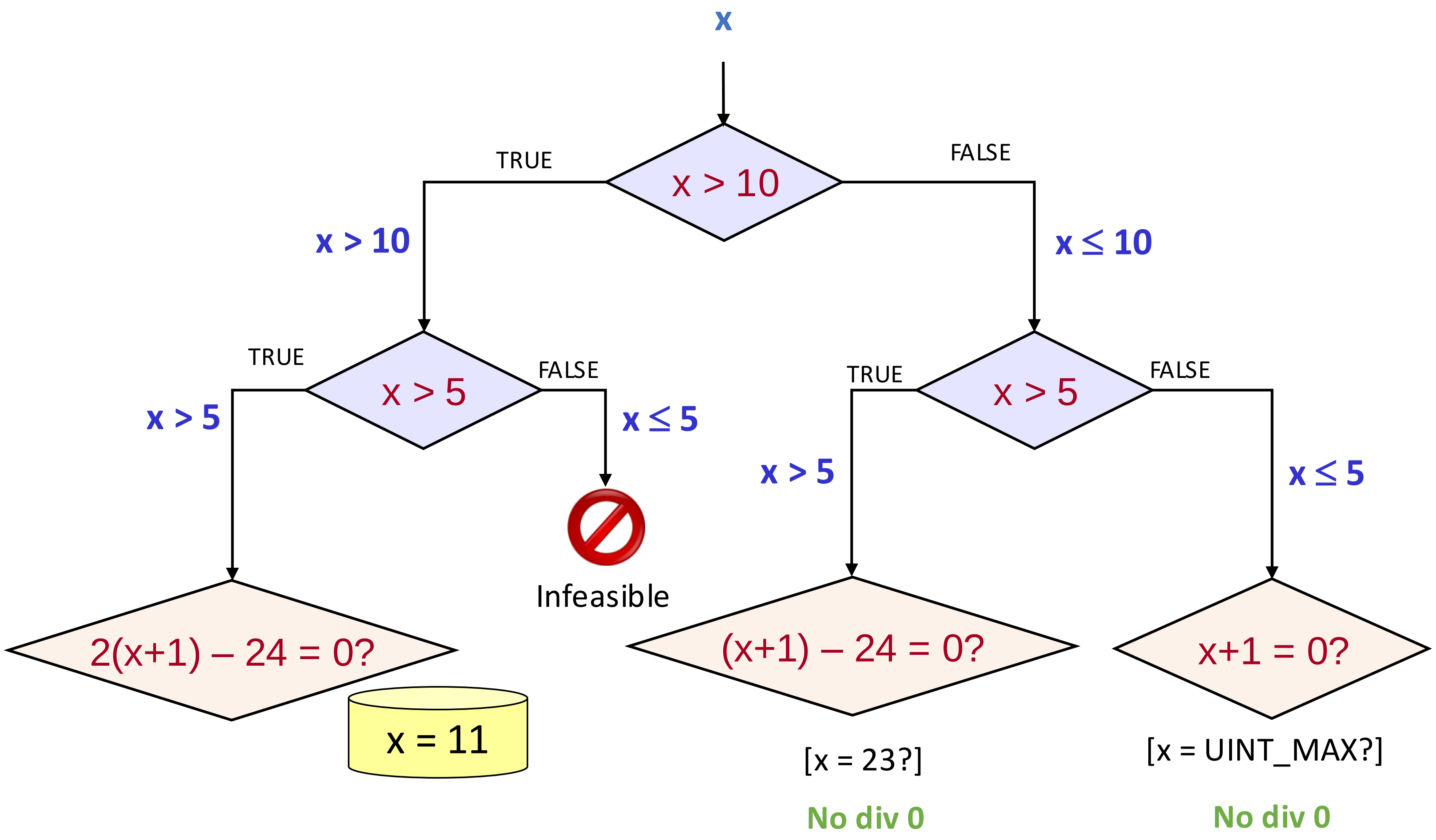
Program ▾ Attending Call for Contributions ▾ Organisation Sponsors Series ▾

4th International KLEE Workshop on Symbolic Execution

15–16 April 2024 • Lisbon, Portugal • Co-located with ICSE 2024

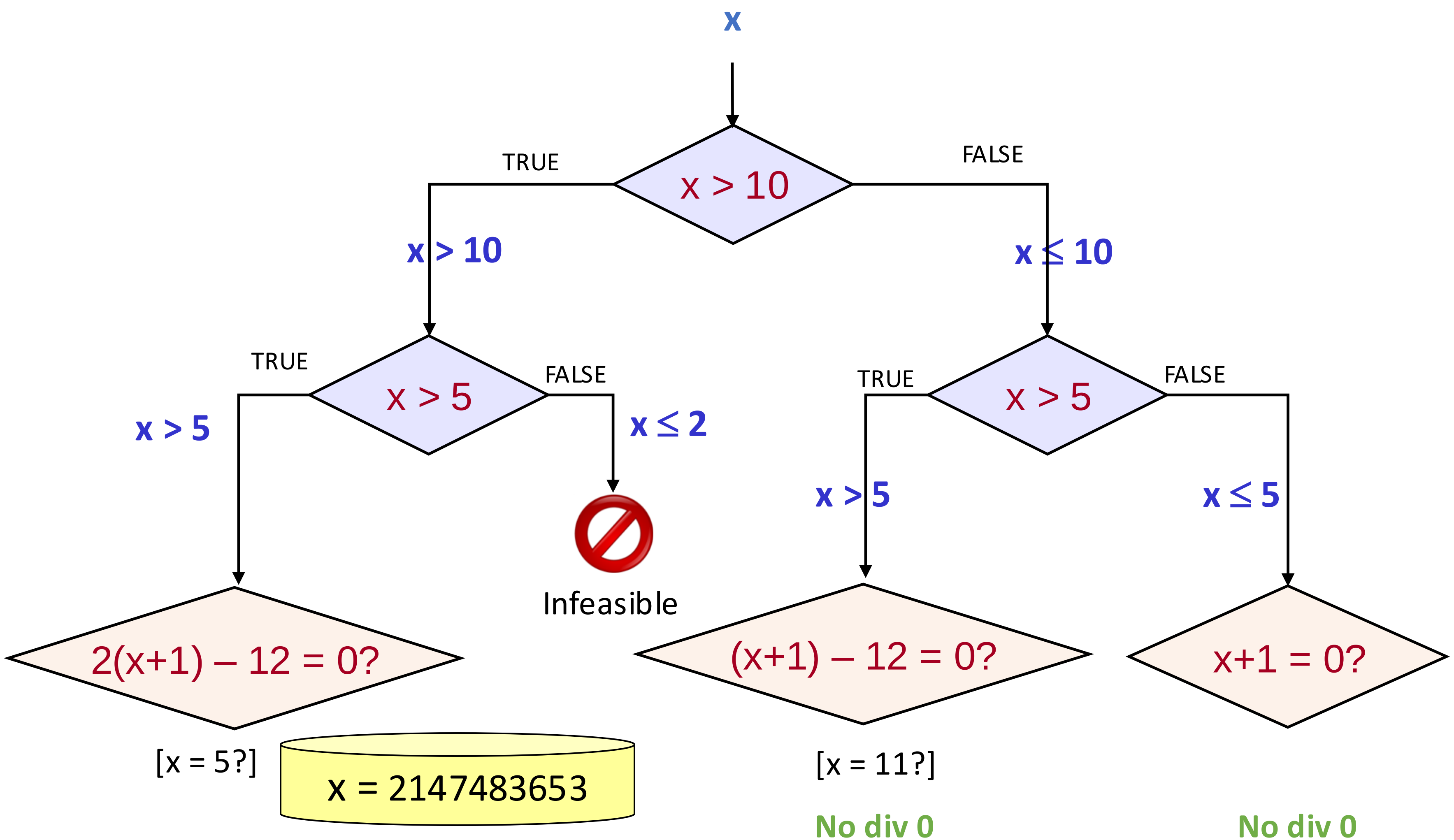
Dynamic Symbolic Execution

```
int foo(unsigned x) {
  int r = x + 1;

  if (x > 10)
    r = 2 * r;

  if (x > 5)
    r = r - 24;

  return x / r;
}
```



Dynamic Symbolic Execution

```
int foo(unsigned x) {
  int r = x + 1;

  if (x > 10)
     r = 2 * r;

  if (x > 5)
     r = r - 12;

  return x / r;
}
```



Dynamic Symbolic Execution

Key advantages:

- Systematically explores unique control-flow paths
- No control-flow abstraction
- No false positives
  - theory and practice!

- Reasons about all possible values on each explored path
- Per-path verification

Key challenges:

- Efficiently solving lots of constraints
- Path explosion, particularly in the presence of loops

A path with 1 iteration through the loop

$\neq$

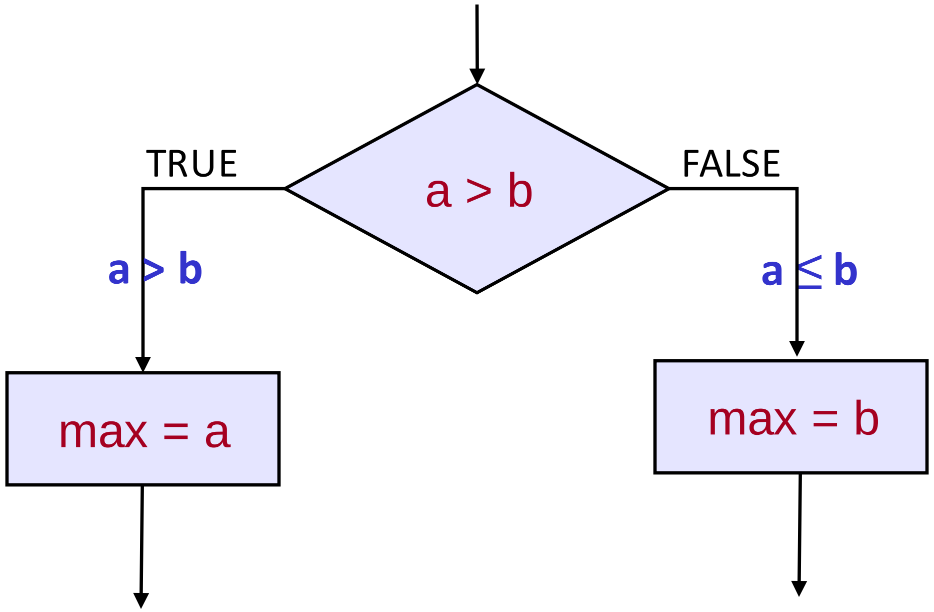
A path with 2 iteration through the loop

Merging Paths

[with P. Collingbourne and P. Kelly]

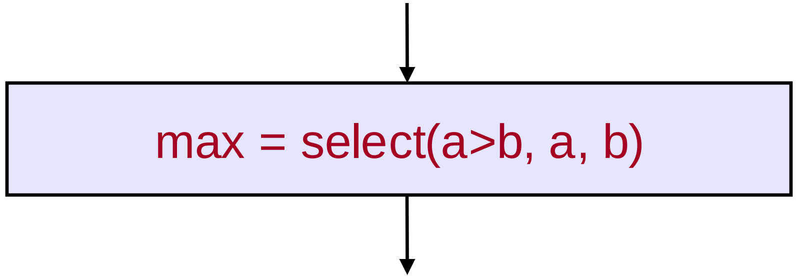
Default behaviour

```
if (a > b)
    max = a;
else max = b;
```



Path merging (via phi-node folding, when no side effects)

```
if (a > b)
    max = a;
else max = b;
```



Merging Paths

```
for (i=0; i < N; i++) {
    if (a[i] > b[i])
            max[i] = a[i];
    else max[i] = b[i];
}
```

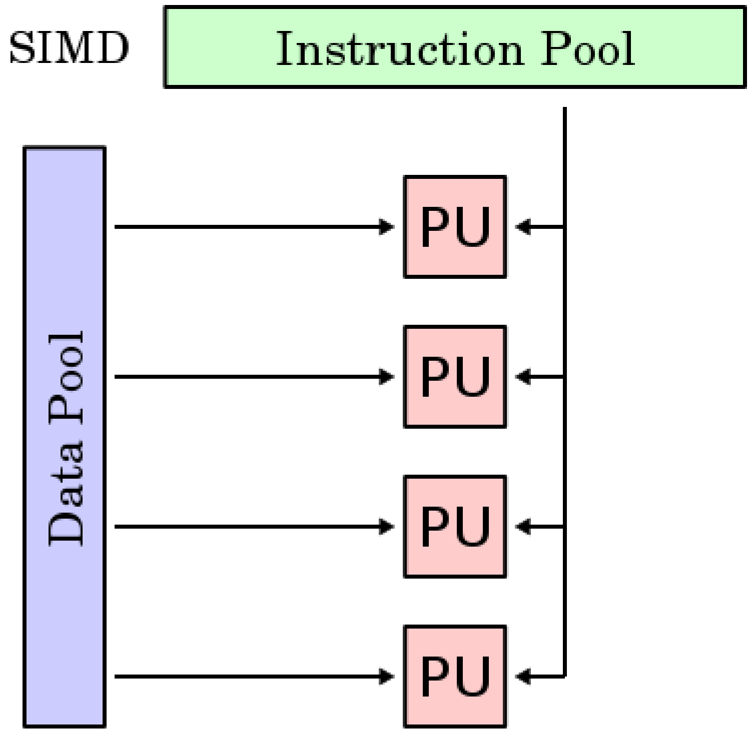
- Default: $\mathbf{2^N}$ paths
- Path merging: 1 path

Path merging $\equiv$ Outsourcing problem to constraint solver

SIMD Optimizations

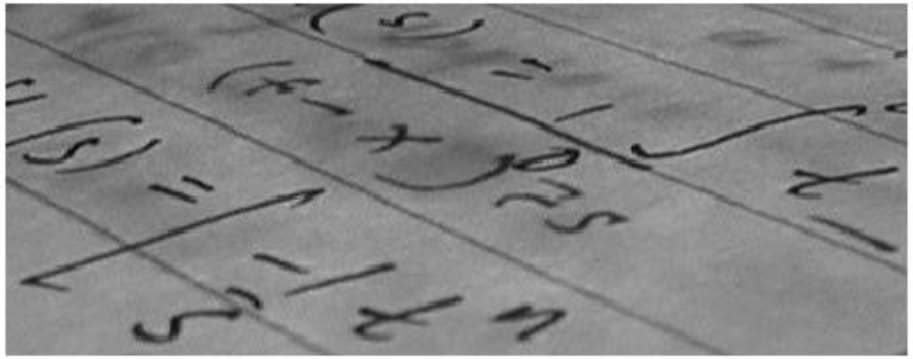
Most processors offer support for SIMD instructions

- Can operate on multiple data concurrently
- Many algorithms can make use of them (e.g., computer vision algorithms)

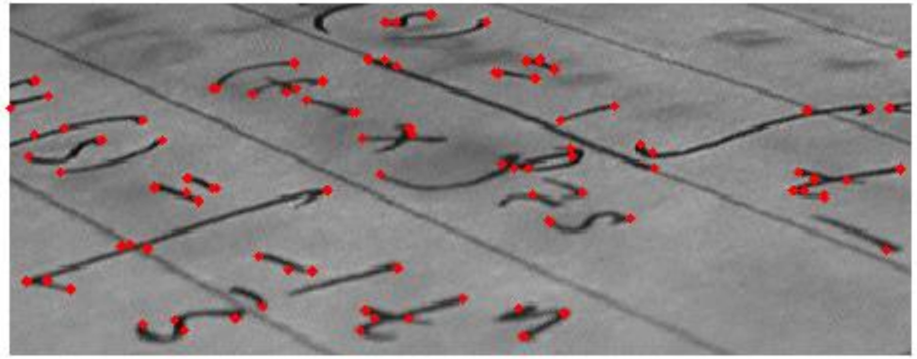


OpenCV

Popular computer vision library from Intel and Willow Garage



[Corner detection algorithm]



Computer vision algorithms were optimized to make use of SIMD

OpenCV: Correctness of SIMD Optimisations

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - DSE with aggressive path merging
- Verified the correctness of 41 of them up to a certain image size
  - Bounded verification
- Found mismatches in the other 10
  - Most mismatches due to tricky FP-related issues:
    precision, rounding, associativity, distributivity, NaN values

OpenCV: Correctness of SIMD Optimisations

Surprising find: min/max not commutative nor associative!

```
min(a,b) = a < b ? a : b

a < b (ordered) → always returns false if one
                  of the operands is NaN

min(NaN, 5) = 5
min(5, NaN) = NaN

min(min(5, NaN),  100) = min(NaN, 100) = 100
min(5, min(NaN, 100))  = min(5, 100) = 5
```

Loop Summaries
[with T. Kapus, O. Ish-Shalom, S. Itzhaky, N. Rinetzky]

- Strings are everywhere
- String operations usually involve loops

- Lots of work from SMT community on building string solvers
  - E.g., Z3, CVC4, HAMPI
- Can we use them for dynamic symbolic execution?

Problem

Developers often use custom loops instead of string functions

```
while (*s != '\n’)
    s++;
```

```
#define whitespace(c) (((c) == '␣') || ((c) == '\t'))
char *p;
for (p = line; p && *p && whitespace (*p); p++)
  ;
```

```
char *p = path + strlen (path);
for (; *p != '/' && p != path; p--)
  ;
```

```
while (('␣' == *pbeg) || ('\r' == *pbeg)
   || ('\n' == *pbeg) || ('\t' == *pbeg))
   pbeg++;
```

Solution

Replace custom loops with sequence of primitive pointer operations and calls to standard string functions

```
s = rawmemchr(s, '\n');
```

```
#define whitespace(c) (((c) == '␣') || ((c) == '\t'))
char *p = line + strspn(line, "␣\t")
```

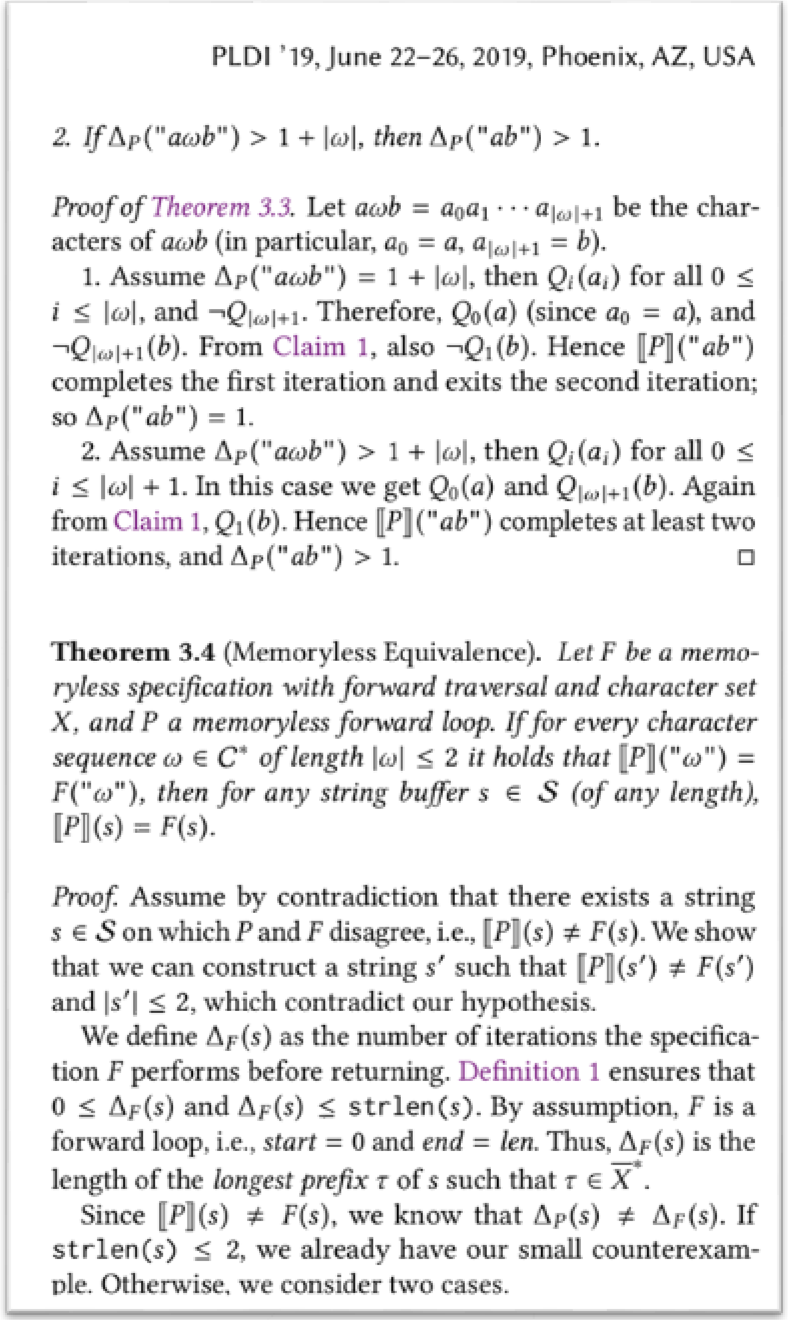
```
pbeg += strspn(pbeg, "␣\r\n\t");
```

```
p = strrchr(path, '/');
p = p == NULL ? path : p;
```

Scope: Memoryless Loops

- Loops conforming to an interface:
  - Argument: single pointer to a string
  - Returns: pointer to an offset in the string
- Only reads the character under current pointer

- For memoryless loops:
  - Equivalence for lengths ≤ 3 implies equivalence for any length
  - Intuitively the proof depends on the fact that each iteration is independent from previous ones



2. If $\Delta_P(\text{"}a\omega b\text{"}) > 1 + |\omega|$, then $\Delta_P(\text{"}ab\text{"}) > 1$.

Proof of Theorem 3.3. Let $a\omega b = a_0 a_1 \cdots a_{|\omega|+1}$ be the characters of $a\omega b$ (in particular, $a_0 = a$, $a_{|\omega|+1} = b$).

1. Assume $\Delta_P(\text{"}a\omega b\text{"}) = 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \le i \le |\omega|$, and $\neg Q_{|\omega|+1}$. Therefore, $Q_0(a)$ (since $a_0 = a$), and $\neg Q_{|\omega|+1}(b)$. From Claim 1, also $\neg Q_1(b)$. Hence $[\![P]\!](\text{"}ab\text{"})$ completes the first iteration and exits the second iteration; so $\Delta_P(\text{"}ab\text{"}) = 1$.
2. Assume $\Delta_P(\text{"}a\omega b\text{"}) > 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \le i \le |\omega| + 1$. In this case we get $Q_0(a)$ and $Q_{|\omega|+1}(b)$. Again from Claim 1, $Q_1(b)$. Hence $[\![P]\!](\text{"}ab\text{"})$ completes at least two iterations, and $\Delta_P(\text{"}ab\text{"}) > 1$. □

Theorem 3.4 (Memoryless Equivalence). *Let F be a memoryless specification with forward traversal and character set X, and P a memoryless forward loop. If for every character sequence $\omega \in C^*$ of length $|\omega| \le 2$ it holds that $[\![P]\!](\text{"}\omega\text{"}) = F(\text{"}\omega\text{"})$, then for any string buffer $s \in \mathcal{S}$ (of any length), $[\![P]\!](s) = F(s)$.*

Proof. Assume by contradiction that there exists a string $s \in \mathcal{S}$ on which P and F disagree, i.e., $[\![P]\!](s) \ne F(s)$. We show that we can construct a string s' such that $[\![P]\!](s') \ne F(s')$ and $|s'| \le 2$, which contradict our hypothesis.

We define $\Delta_F(s)$ as the number of iterations the specification F performs before returning. Definition 1 ensures that $0 \le \Delta_F(s)$ and $\Delta_F(s) \le \texttt{strlen}(s)$. By assumption, F is a forward loop, i.e., *start* = 0 and *end* = *len*. Thus, $\Delta_F(s)$ is the length of the *longest prefix* τ of s such that $\tau \in \overline{X}^*$.

Since $[\![P]\!](s) \ne F(s)$, we know that $\Delta_P(s) \ne \Delta_F(s)$. If $\texttt{strlen}(s) \le 2$, we already have our small counterexample. Otherwise, we consider two cases.

Vocabulary for Summarising String Loops

string.h functions

- `strspn`
- `strcspn`
- `memchr`
- `strchr`
- `strrchr`
- `strpbrk`

pointer manipulation

- `increment`
- `set to start`
- `set to end`

special

- `backward traverse`
- `return`

conditionals

- `is null`
- `is start`

```
for (char* p = line;
      *p && (*p == '␣' ||  *p == '\t');
      p++) ;
```

size_t **strspn**(const char *s, const char *charset);

"computes the string array index of the first character of s which is not in charset"

```
char *p = line + strspn(line, "␣\t")
```

STRSPN	␣\t	\0	RETURN

Loop summary

Interpreter for Loop Summaries

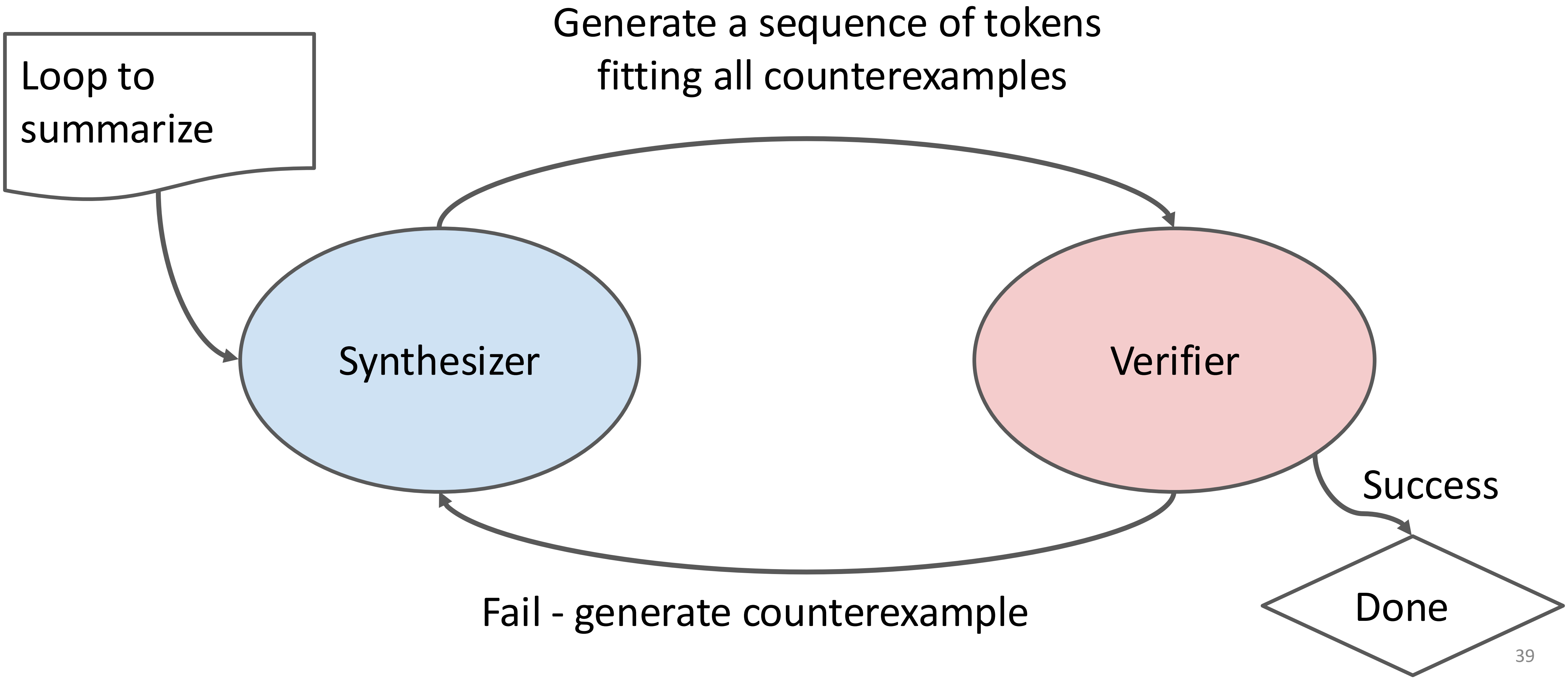
- Loop summary has meaning in an `interpreter()`
- Adding a new vocabulary item as simple as adding a new **`case`**

```
#define STRSPN ‘P’
#define RETUNR ‘F’

char* interpreter(char* input) {
  char *result = input;

  while(token = nextToken())
    switch(token)
      case STRSPN:
        result += strspn(result,
                     nextData());
      case RETURN:
        return result;
}
```

Loop summarization:

Find sequences of character tokens that when executed by our interpreter have the same behaviour as the original loop

Counterexample Guided Synthesis



Synthesizer

- Dynamic symbolic execution
- Symbolic input: sequence of tokens
- Constrain it to be equivalent on current (counter)examples
- Ask an SMT solver for a solution

Verifier

- Dynamic symbolic execution
- Symbolic input: strings of length ≤ 3
- Exhaustively check that the original loop is equivalent to the interpreted loop summary

Synthesis Evaluation

- 13 open source programs
- Extracted 115 memoryless loops
- 88/115 successfully synthesized within 2h*
- 81 within 5 minutes

*Gaussian process optimization to optimize the vocabulary

patch

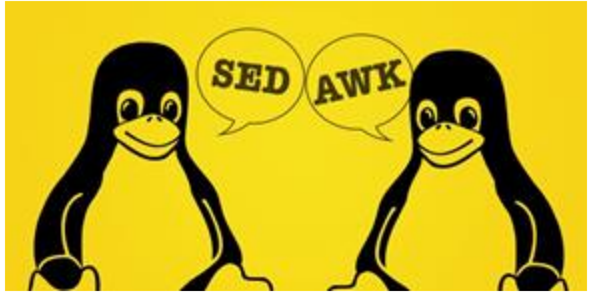
libosip

diff

m4

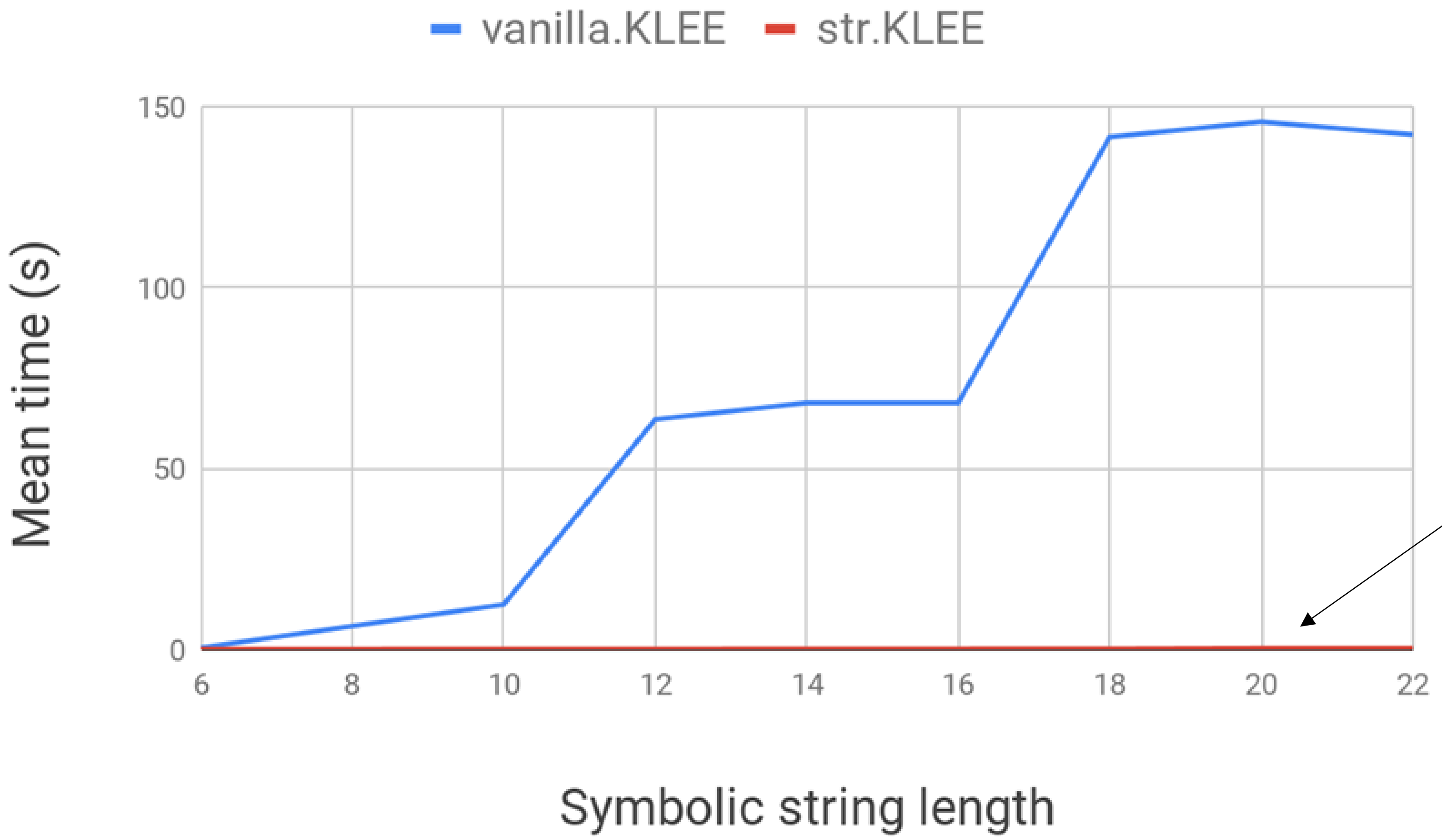
$ wget

make



Impact of string solvers (KLEE+Z3str) on DSE

Average across loops, 2min timeout



Can reason about unbounded string lengths

Refactoring

- Used summaries to create patches and send them to developers
- Submitted patches to 5 applications
- Patches accepted in `libosip`, `patch` and `wget`

```
- for(; *tmp == ' ' || *tmp == '\t'; tmp++){
- }
- for(; *tmp == '\n' || *tmp == '\r'; tmp++){
- }                                   /* skip LWS */
+ tmp += strspn(tmp, " \t");
+ tmp += strspn(tmp, "\n\r");
```

Dynamic Symbolic Execution

- DSE offers a middle ground b/w testing and verification

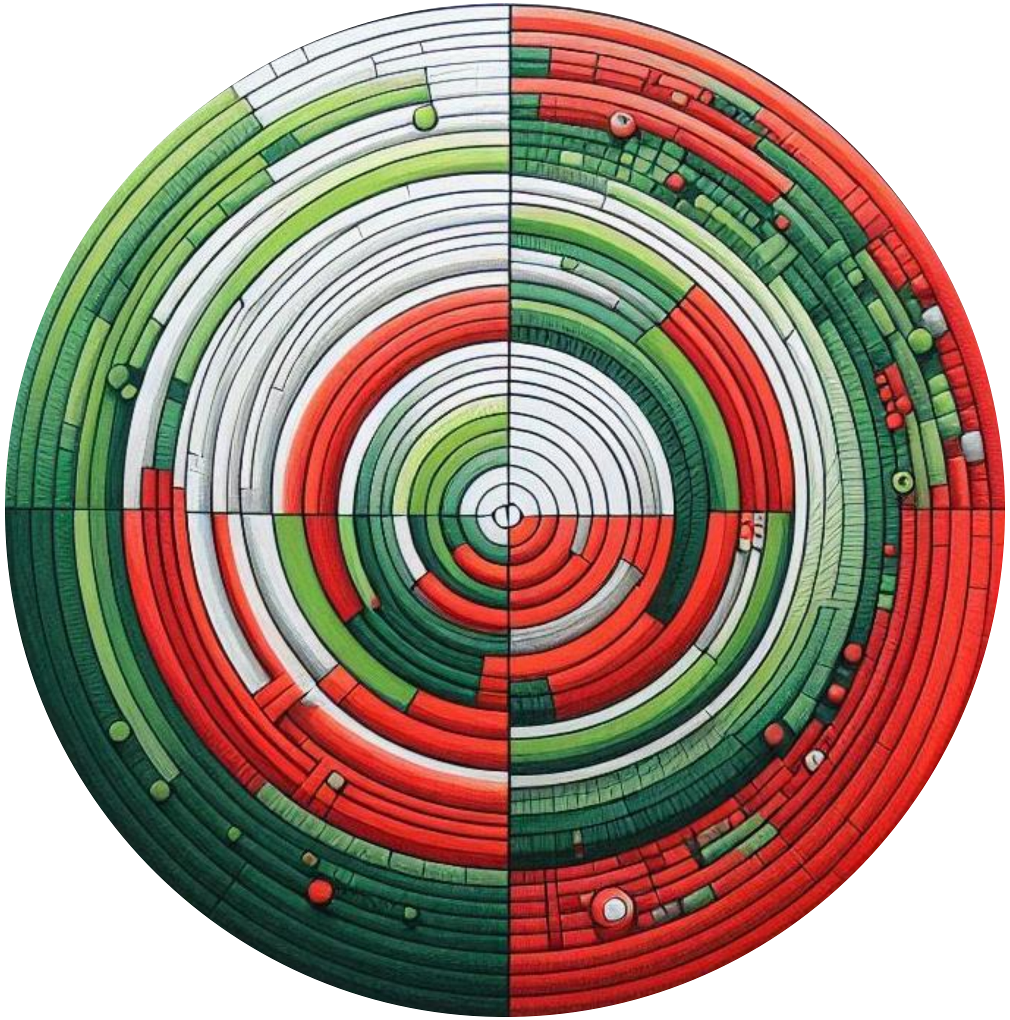
- DSE systematically explores paths through the code
- As in testing, no false positives, but only some paths are explored
- Exhaustive path exploration → verification

- As in testing, concrete inputs (best bug reports!) can be produced
- But unlike testing, DSE reasons about all possible values on a path: *per-path verification*

- DSE has already been successfully used for bounded verification in combination with path merging/code summarisation

- Open challenges include:
  - the right trade-off b/w individual path exploration and summarization
  - reasoning about unbounded inputs
  - combining DSE with other testing and verification techniques
  - applying DSE to new types of verification scenarios (particularly interested in patch verification!)

Testing and Verification



- ***What parts of the software should be verified and what parts tested?***
  - What are the partial guarantees in each case?
  - Under what assumptions?
  - Can one control the FP/FN ratio?
- ***Can testing/verif. handle fast evolving software?***
  - Can I test/verify software changes quickly?
- ***Does the testing/verification approach integrate well with existing development practices?***
  - How hard is to use the testing/verif. system?
  - What is the annotation/specif. writing effort?
  - Does it enhance/complement/hinder the existing development practices?