28th European Conference on Artificial Intelligence (ECAI 2025)
Generative Code Intelligence Workshop (GeCoIn 2025)

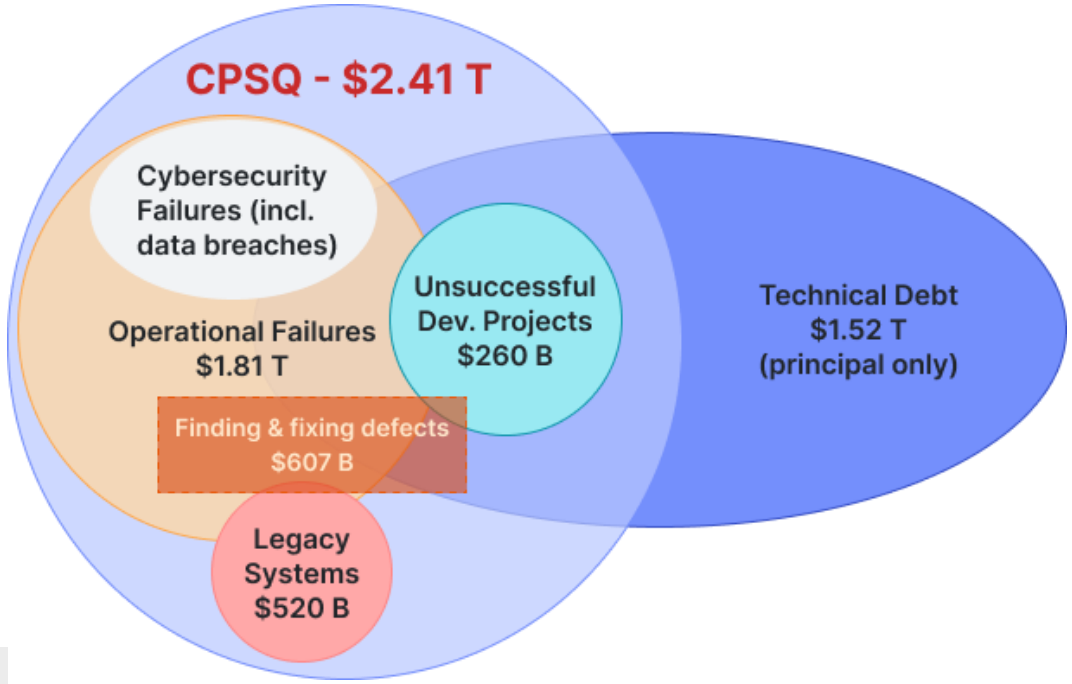
The University of Manchester

# AI-Assisted Formal Verification: Towards Fast, Accessible, and Rigorous Software Verification

**Lucas C. Cordeiro**
**Department of Computer Science**
lucas.cordeiro@manchester.ac.uk
https://ssvlab.github.io/lucasccordeiro/

# The $2.41 Trillion Problem

**Poor software quality cost** US companies **$2.41 trillion in 2022**, while the **accumulated software Technical Debt** (TD) has grown to **~$1.52 trillion**




→ 50%+ of project costs go to debugging, not building

→ The stakes are even higher for AI-generated code

→ AI makes us faster, but are we safer?

CISQ Consortium for Information & Software Quality™

2025 Quality Transformation Report: Closing the gap between speed, quality, and trust in the AI era

Tricentis

# How Secure is AI-Generated Code?

| Category | Avg Prop. Viol. per Line | Rank | $\mathcal{VS}$ | Rank | $\mathcal{VF}$ | $\mathcal{VU}$ (Timeout) | Avg Prop. Viol. per File |
|---|---|---|---|---|---|---|---|
| GPT-4o-mini | **0.0165** | 3 | 4.23% | 2 | 57.14% | 36.77% | 3.40 |
| Llama2-13B | 0.0234 | 2 | 12.36% | **1** | **51.30%** | 31.78% | 3.62 |
| Mistral-7B | 0.0254 | 7 | 8.36% | 4 | 62.08% | 25.88% | **3.07** |
| CodeLlama-13B | 0.0260 | **1** | **15.48%** | 3 | 52.71% | 29.52% | 4.13 |
| Falcon-180B | 0.0291 | 8 | 6.48% | 5 | 62.07% | 28.67% | 3.38 |
| GPT-3.5-turbo | 0.0295 | 6 | 7.29% | 7 | 65.07% | 26.09% | 4.42 |
| Gemini Pro 1.0 | 0.0305 | 5 | 9.49% | 6 | 63.91% | 24.13% | 4.70 |
| Gemma-7B | 0.0437 | 4 | 11.62% | 8 | 67.01% | 16.30% | 4.20 |

**Legend:**

$\mathcal{VS}$ :0.0234 Verification Success; $\mathcal{VF}$ : Verification Failed; $\mathcal{VU}$ : Verification Unknown (Timeout).

Best performance in a category is highlighted with bold and/or Rank.

Even the best LLMs fail on more than half of the verification tasks

Tihanyi, N., Bisztray, T., Ferrag, M., Jain, R., Cordeiro, L.: *How secure is AI-generated code: a large-scale comparison of large language models*. Empir. Softw. Eng. 30(2): 47 (2025)

# Why Formal Methods?

The White House Makes It Clear (Feb 2024):

***"While formal methods have been studied for decades, their deployment remains limited; further innovation in approaches to make formal methods widely accessible is vital to accelerate broad adoption."***

**Formal methods can:**

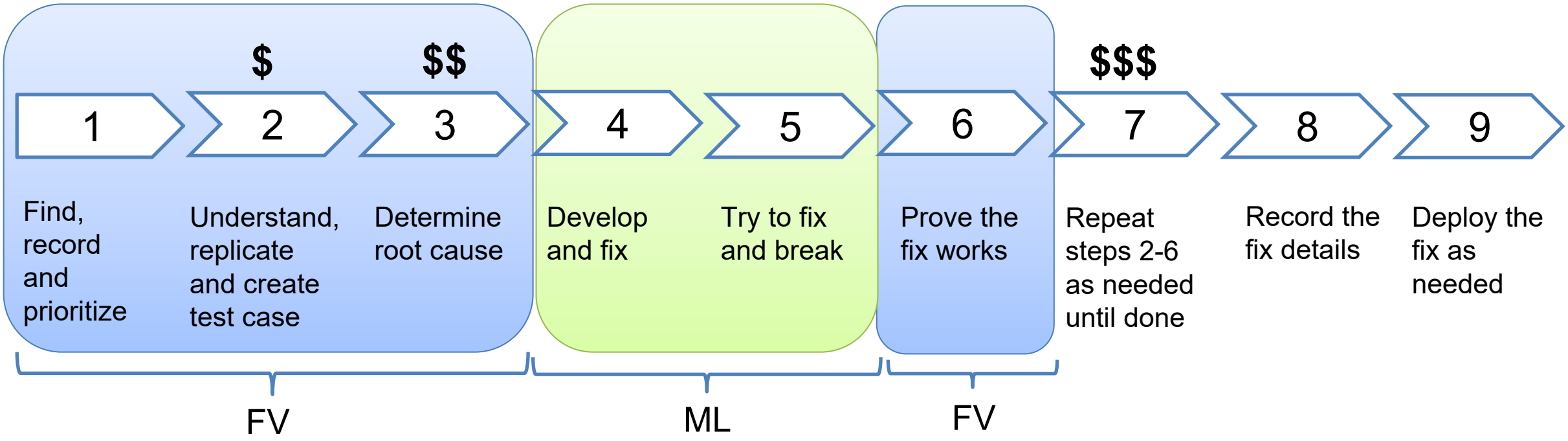
- ✓ Prove software correctness
- ✓ Guarantee safety/security properties
- ✓ Find bugs exhaustively

**But they require:**

- Expert knowledge
- Manual effort
- Specialized training

Could we combine Generative AI with Formal Verification?

# Find, Understand and Fix Bugs



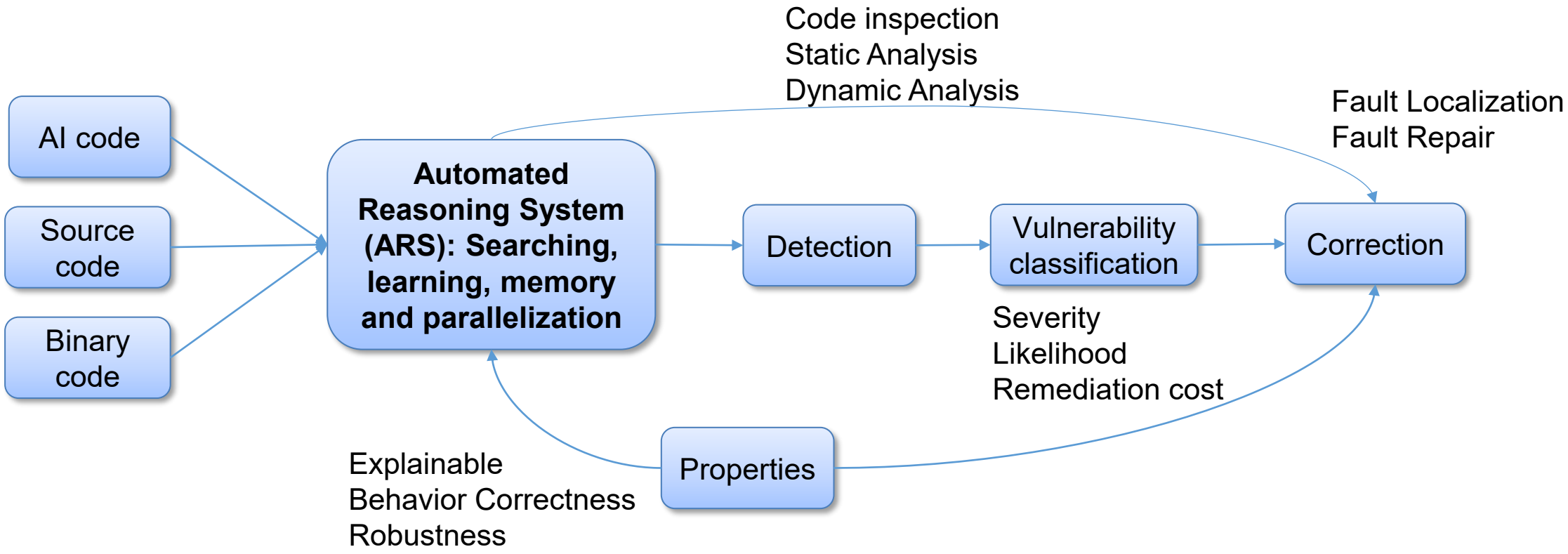

*"A significant percentage (50%+) of a **software project's cost today is not spent on the creativity activity of software construction** but rather on the **corrective activity of debugging and fixing errors**"*

The cost of poor software quality in the US: A 2022 Report

CISQ
Consortium for Information & Software Quality™

# Vision: Building Trustworthy Software and AI Systems

Develop an automated reasoning system for **safeguarding software and AI systems** against vulnerabilities



# Objective of this talk

Discuss how **generative AI and formal methods** can work together to establish a foundation **for building trustworthy software and AI systems**

- Introduce a **logic-based automated reasoning platform** to **find and fix software defects**
- Explain how **testing, verification, LLMs** can be combined to build **trustworthy software and AI systems**
- Develop an **automated reasoning system** for **safeguarding software and AI systems** against vulnerabilities

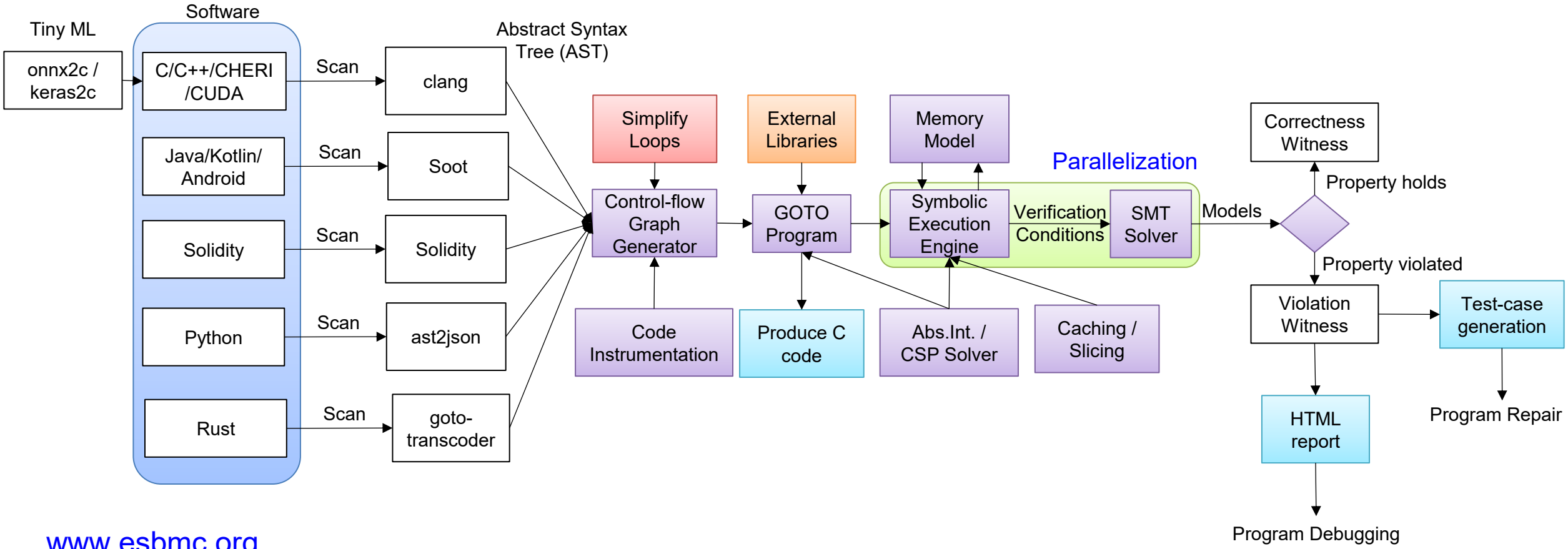
# Research Questions

Given a **program** and a **specification**, can we automatically **verify** that the **program performs as specified**?

Can we leverage **LLMs** to **speed up program verification** and **find more software vulnerabilities** than SOTA?

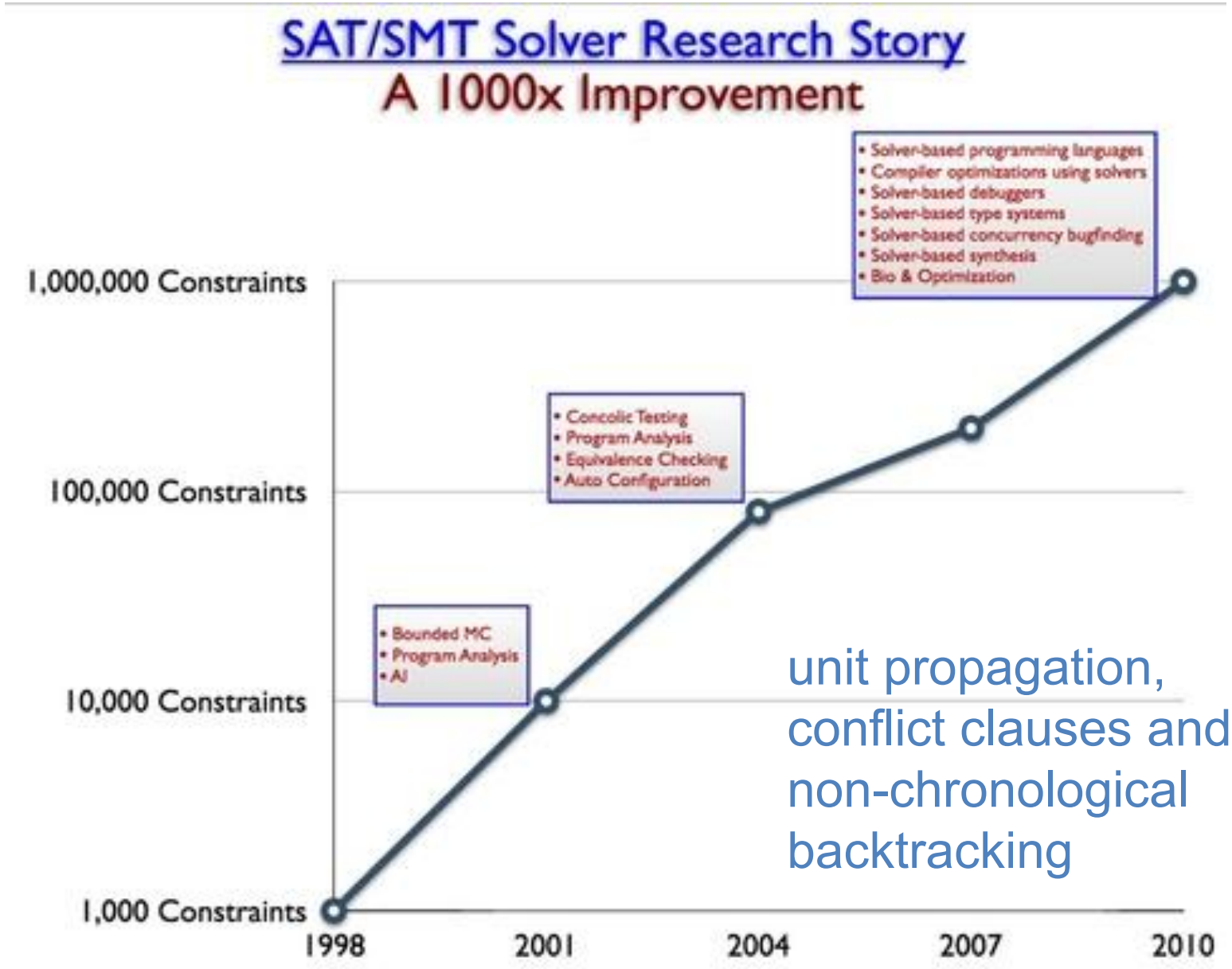
# ESBMC: A Logic-based Verification Platform

**Logic-based automated verification** for checking **safety** and **liveness** properties in **AI** and **software systems**

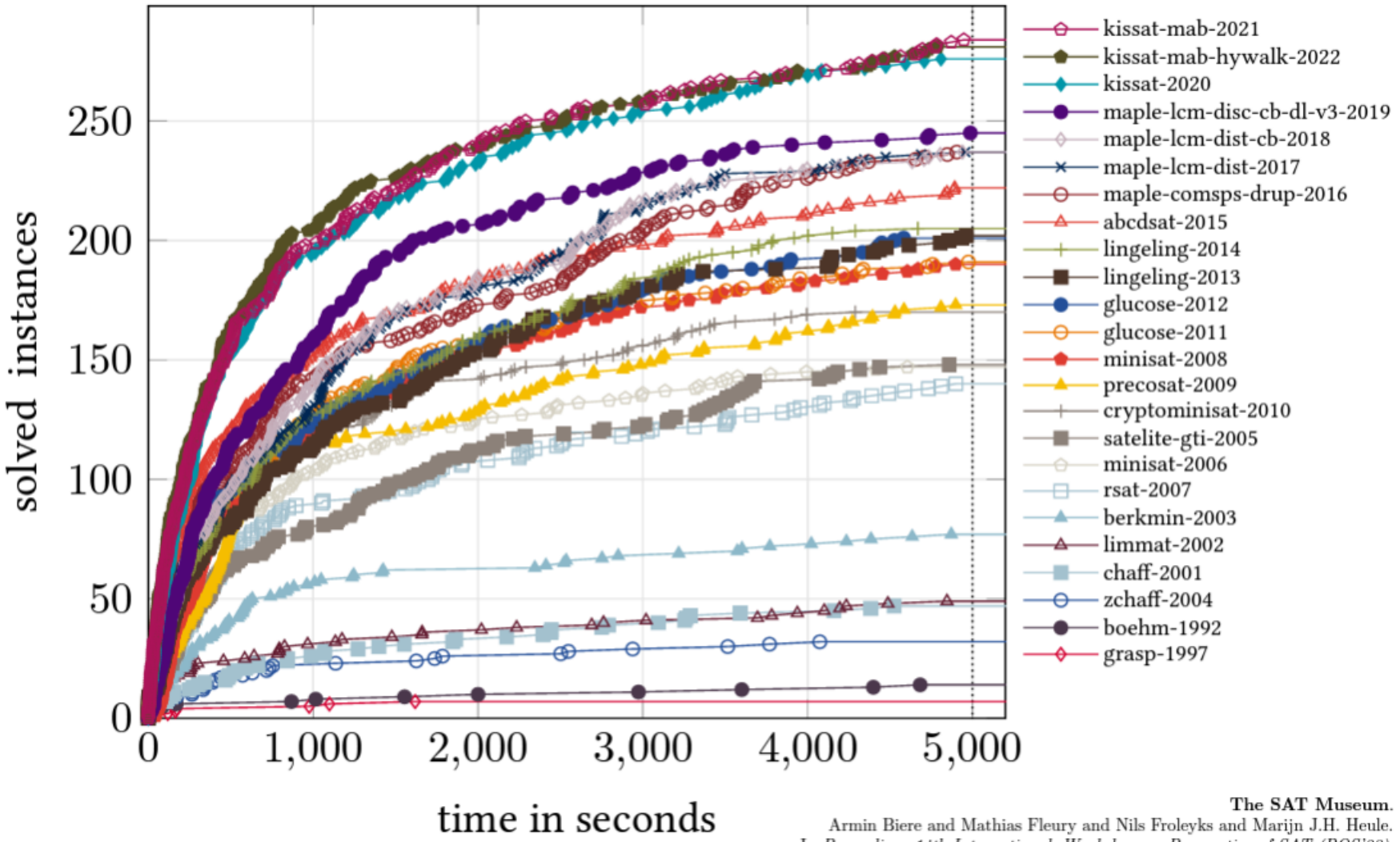



www.esbmc.org

# SAT solving as enabling technology



## SAT Competition All Time Winners on SAT Competition 2022 Benchmarks

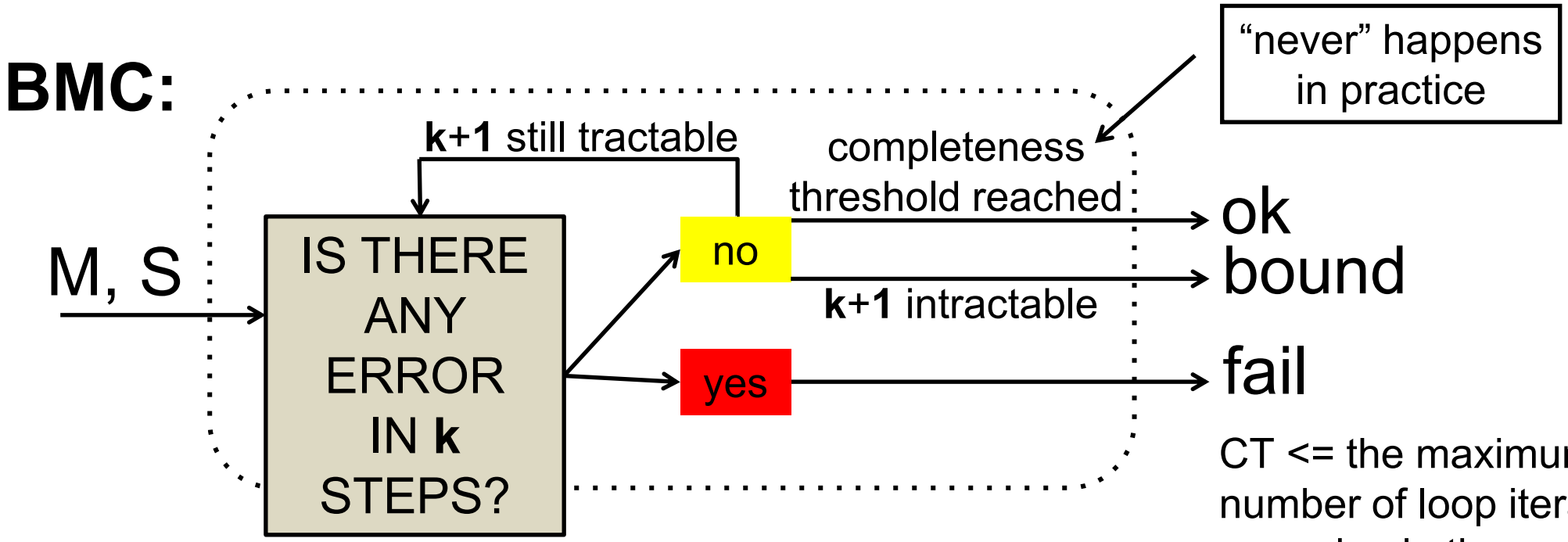



https://cca.informatik.uni-freiburg.de/satmuseum

**The SAT Museum.**
Armin Biere and Mathias Fleury and Nils Froleyks and Marijn J.H. Heule.
In *Proceedings 14th International. Workshop on Pragmatics of SAT (POS'23)*,
vol. 3545, CEUR Workshop Proceedings, pages 72-87, CEUR-WS.org 2023.
[ paper - bibtex - data - zenodo - ceur - workshop - proceedings ]

https://cca.informatik.uni-freiburg.de/satmuseum/

# Bounded Model Checking (BMC)




**Can the given property fail in *k*-steps?**

$$I(S_0) \wedge T(S_0,S_1) \wedge \ldots \wedge T(S_{k-1},S_k) \wedge (\neg P(S_0) \vee \ldots \vee \neg P(S_k))$$

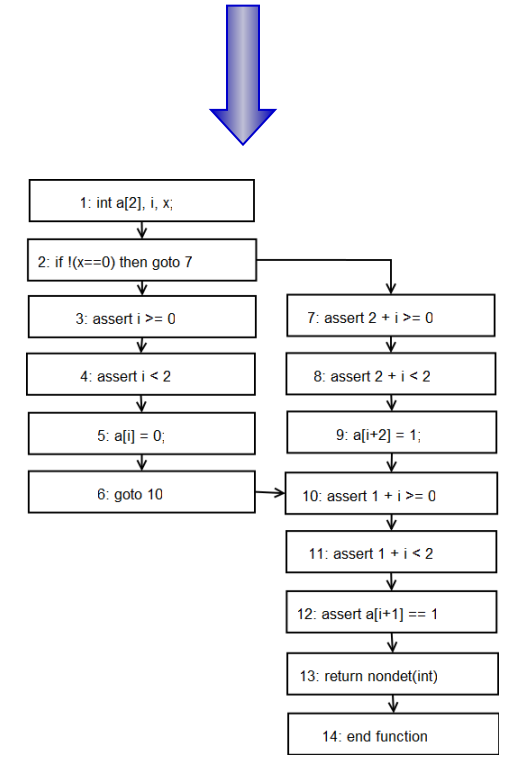
**Initial state** — **k-steps** — **Property fails in some step**

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: Symbolic Model Checking without BDDs. TACAS 1999: 193-207

# Software BMC

- program modeled as a state transition system

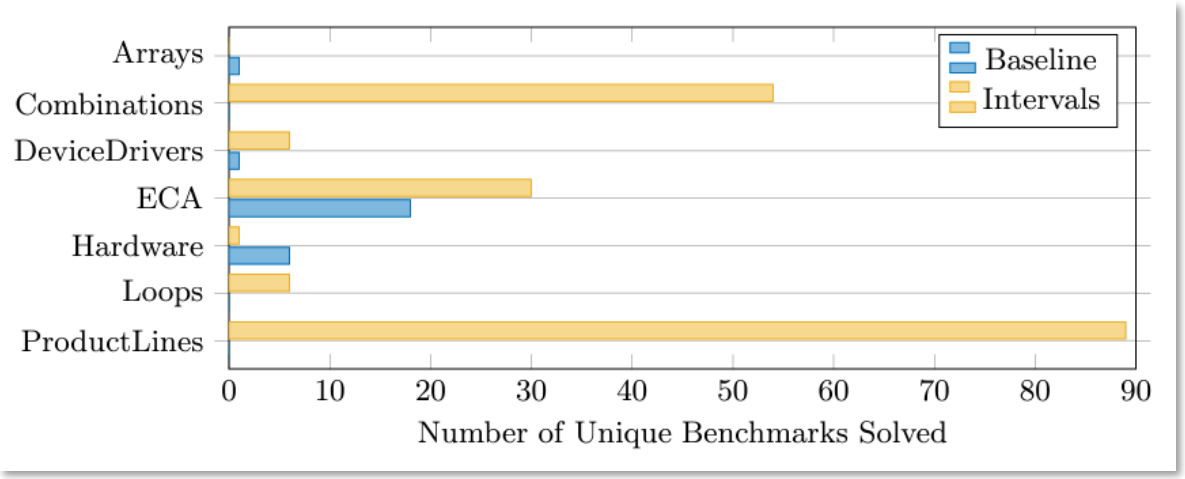
```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



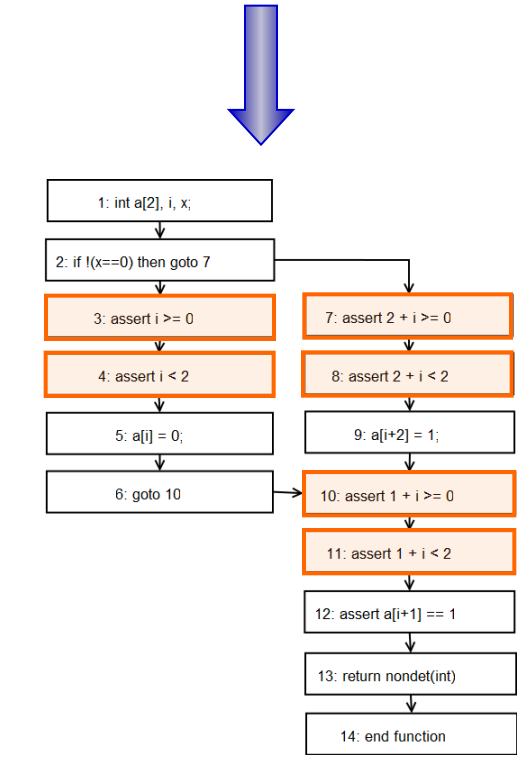
# Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



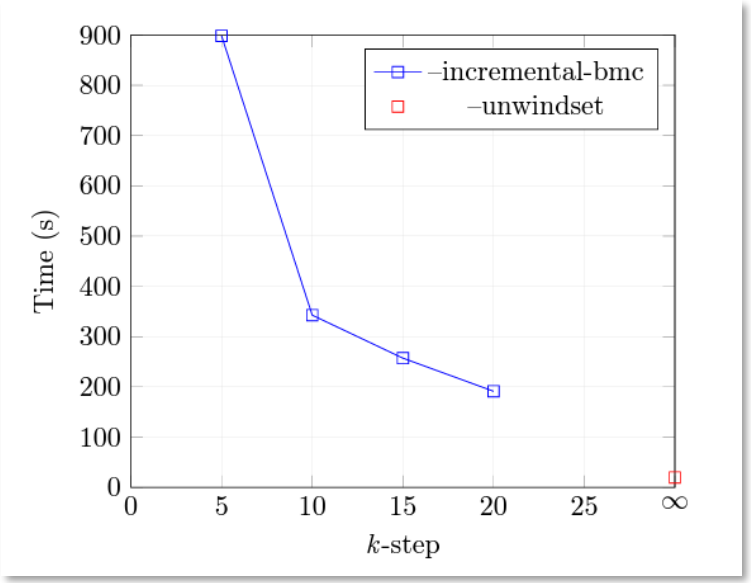

Menezes, R., Manino, E., Shmarov, F., Aldughaim, M., de Freitas, R., Lucas C. Cordeiro: Interval Analysis in Industrial-Scale BMC Software Verifiers: A Case Study.



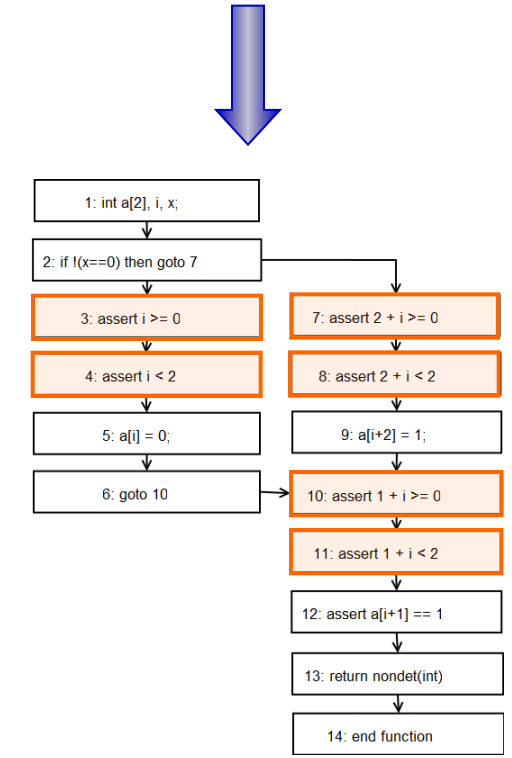
# Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



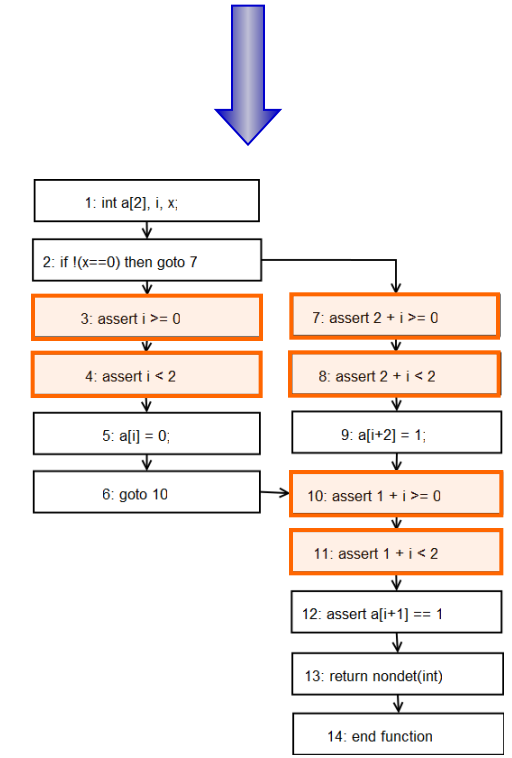

Wu T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L.: Verifying components of Arm(R) Confidential Computing Architecture with ESBMC. SAS 2024

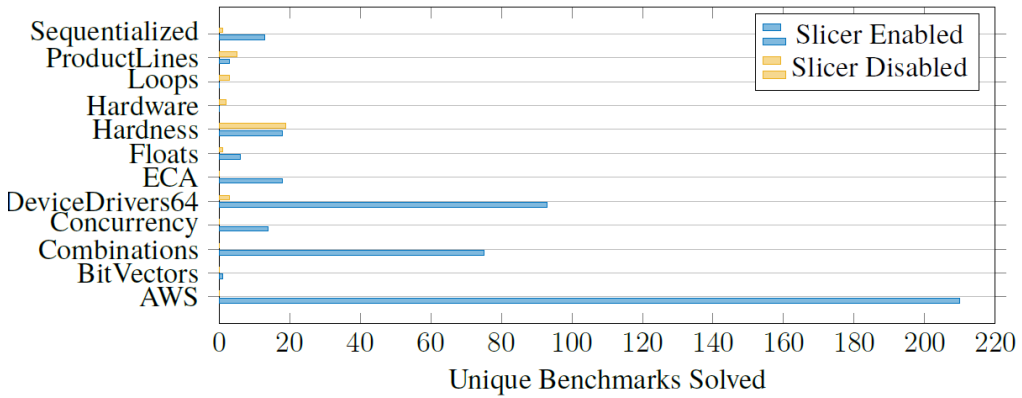


# Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



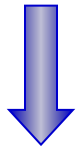



# Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } crucial
- front-end converts unrolled and **optimized program into SSA**

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



$g_1 = x_1 == 0$
$a_1 = a_0$ WITH $[i_0:=0]$
$a_2 = a_0$
$a_3 = a_2$ WITH $[2+i_0:=1]$
$a_4 = g_1 \ ? \ a_1 : a_3$
$t_1 = a_4[1+i_0] == 1$

# Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of $C \wedge \neg P$

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

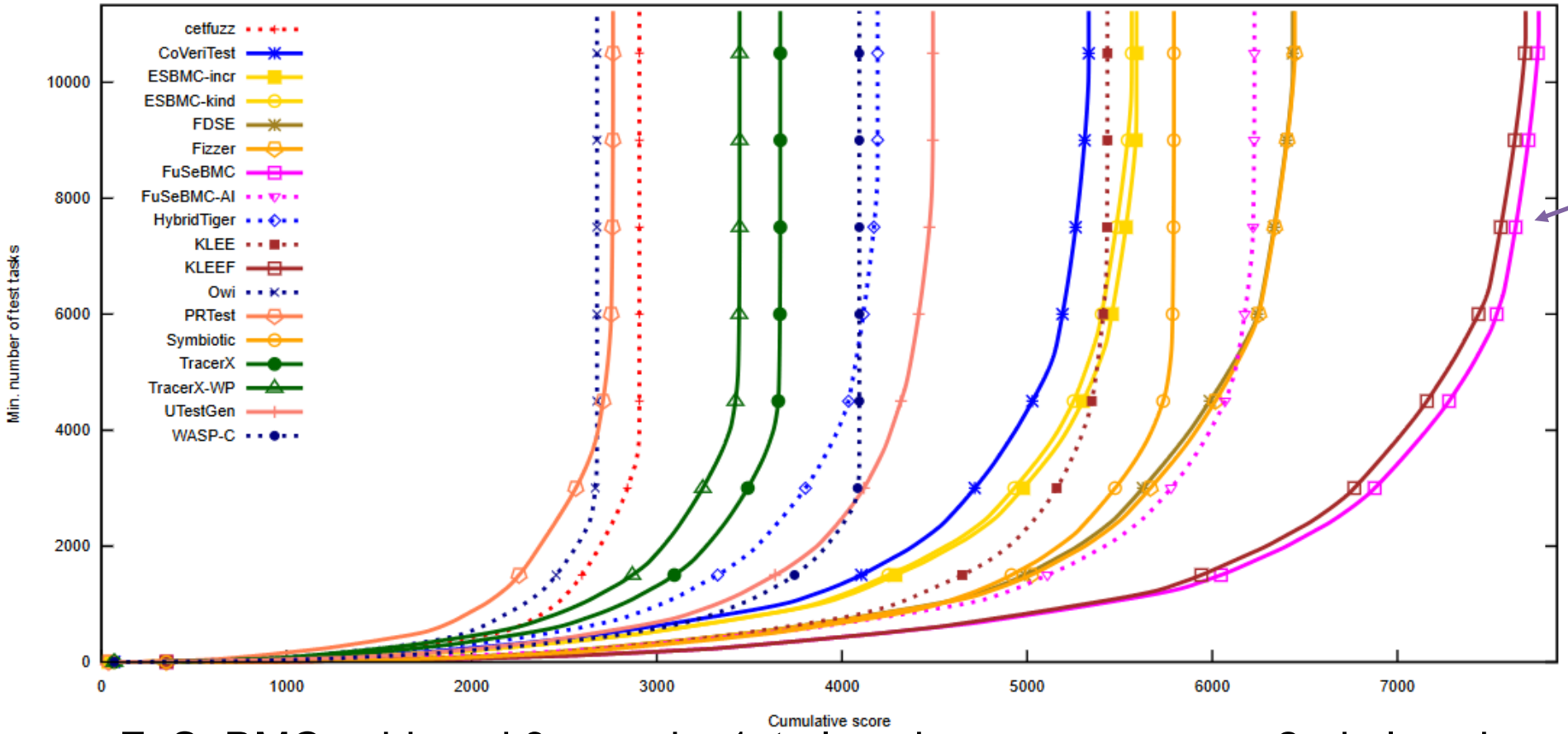
$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Trans. Software Eng. 38(4): 957-974 (2012)

# Most Influential Paper Award at ASE 2023

# Competition on Software Testing 2025: Results of the `Overall` Category




FuSeBMC

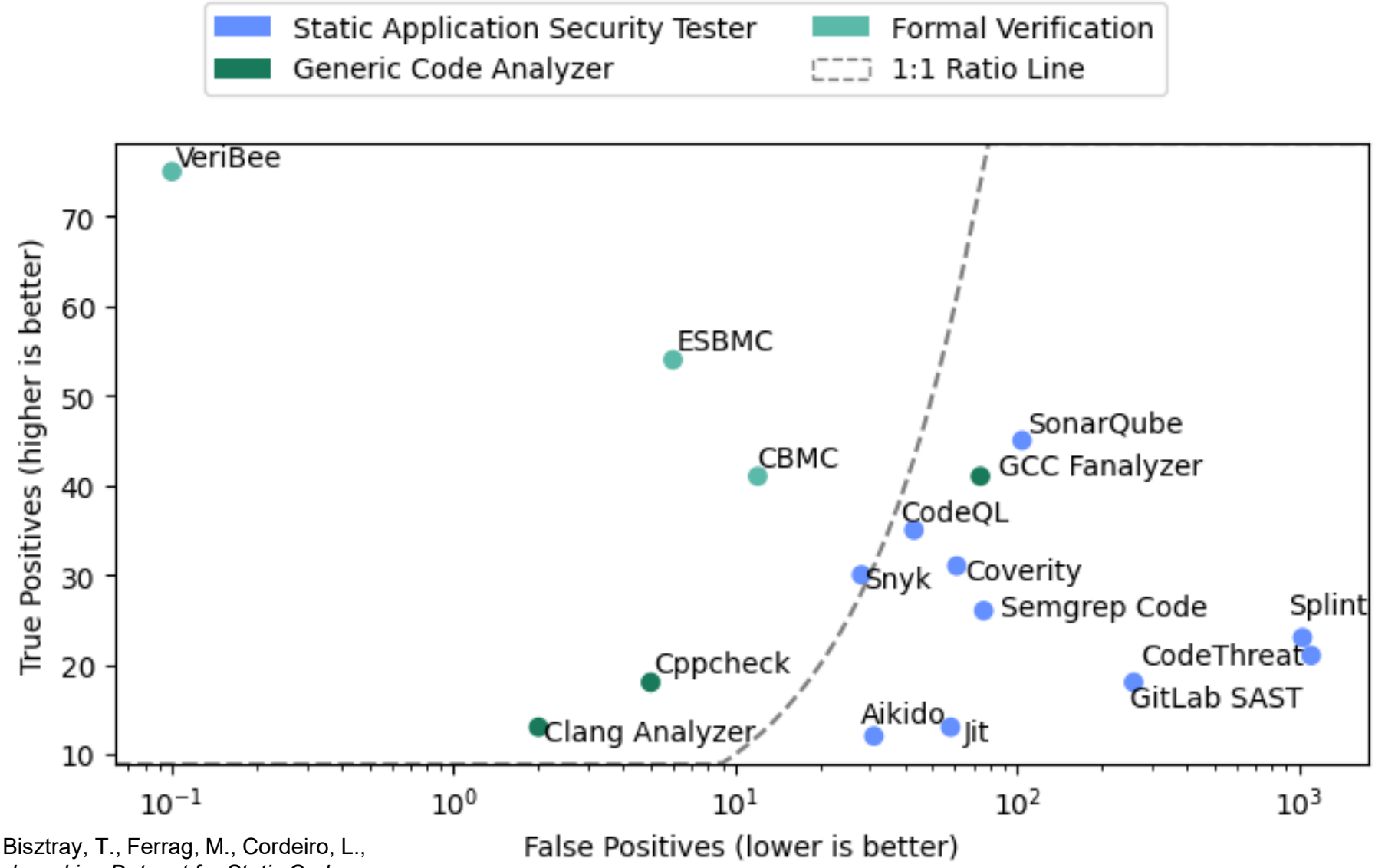
MANCHESTER 1824
The University of Manchester

FuSeBMC achieved 3 awards: 1st place in `Cover-Error`, 2nd place in `Cover-Branches`, and 1st place in `Overall`

Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

https://test-comp.sosy-lab.org/2025/

# Source Code Security




Dubniczky, R., Horvát, K., Bisztray, T., Ferrag, M., Cordeiro, L., Tihanyi, N.: *CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs Towards CWE Detection.* TASE 2025: 253-272

# Research Questions

Given a **program** and a **specification**, can we automatically **verify** that the **program performs as specified**?

Can we leverage **LLMs** to **speed up program verification** and **find more software vulnerabilities** than SOTA?

# Is this Program Correct?

```
def main() -> None:
    x: int = 0
    y: int = 50
    while x < 100:
        x = x + 1
        if x > 50:
            y = y + 1
    assert y == 100
```

```
def main() -> None:
  x: int = 0
  y: int = 50
  x = x + 1
  if x > 50:
      y = y + 1
  ... // 99 other loop unrollings
  assert y == 100
```

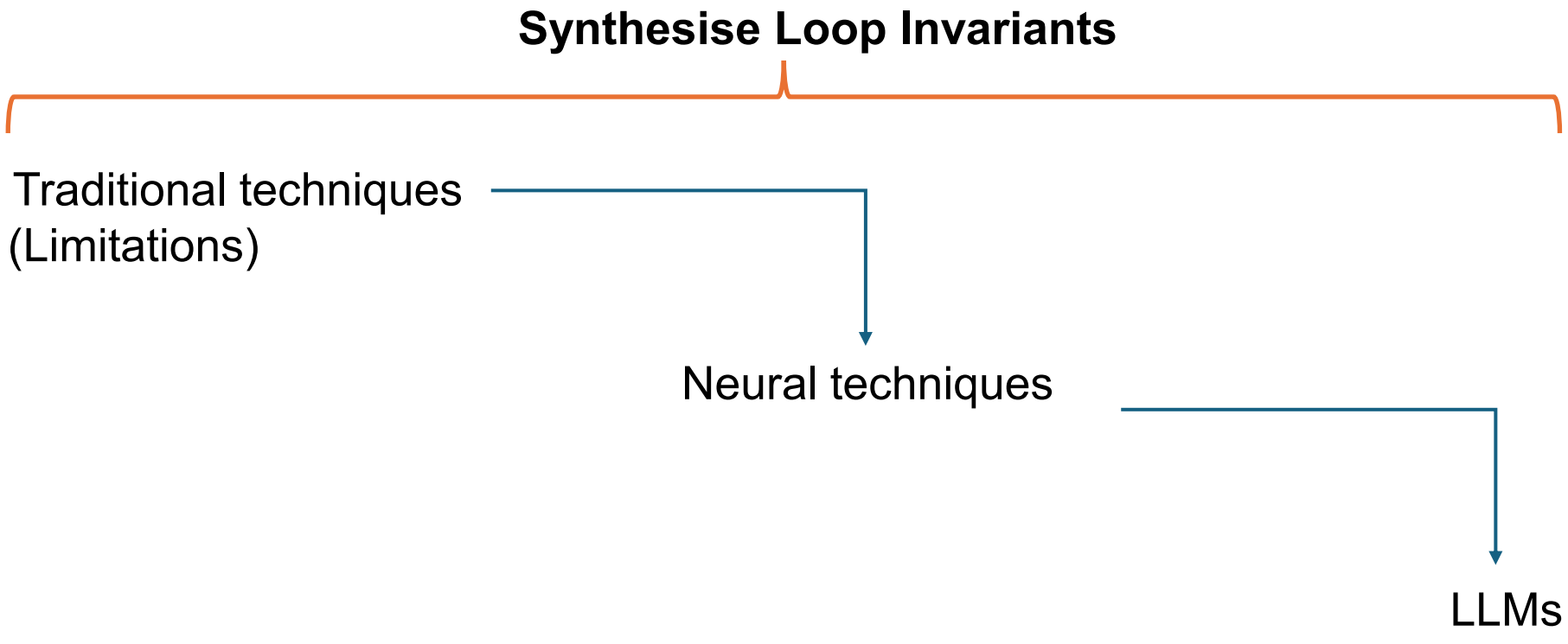
BMC can detect bugs, but requires assistance in proving correctness

Expensive

BMC struggles with loops that can't be statically bound or that have a large bound

# Loop Invariants

- **Inductive loop invariant:** A logical assertion that holds at a loop head whenever the program passes that location
- **Base Case:** The invariant holds before the first iteration of the loop
- **Step Case:** The invariant holds for every iteration of the loop



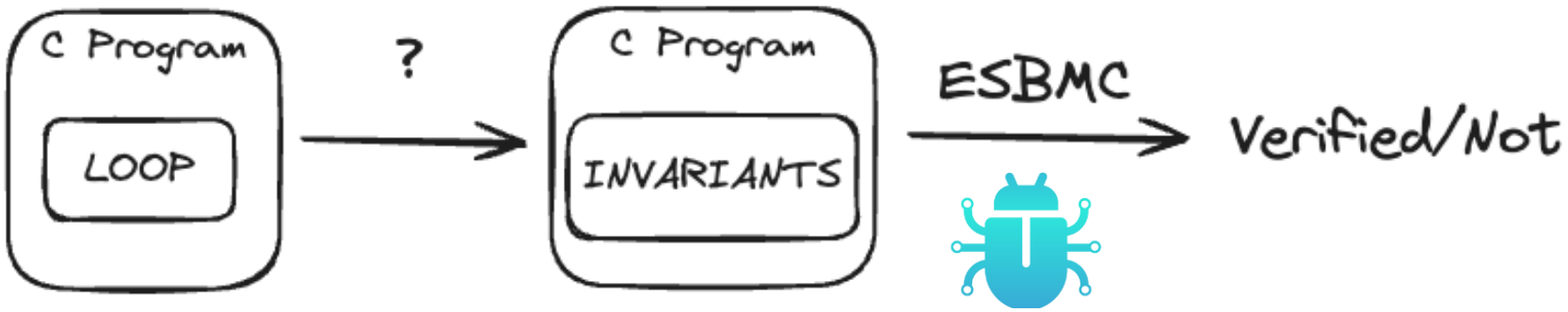
# Replace Loop with Loop Invariants

```
while x < 100:
    x += 1
    if x > 50:
        y += 1
```

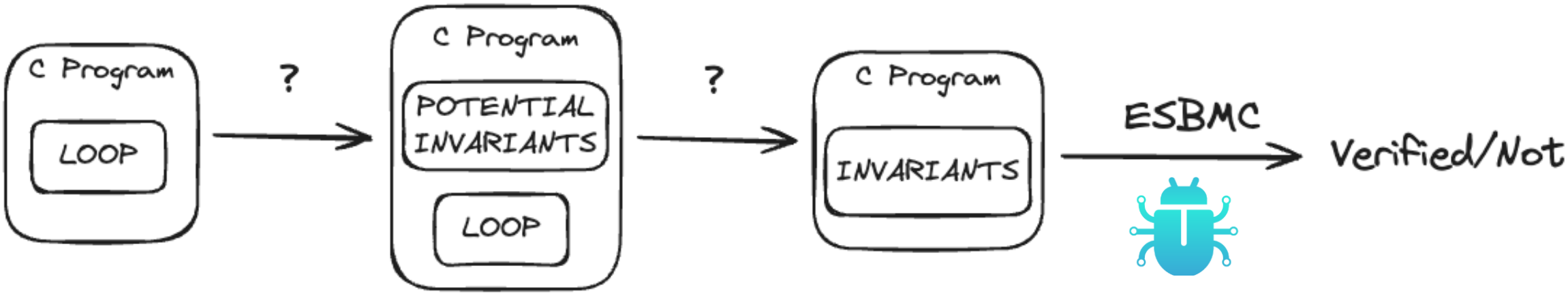
```
def main()
    x: int = 0
    y: int = 50
    # The loop keeps x between 0 and 100
    __ESBMC_assume(0 <= x && x <= 100)
    # If x is 50 or less then y is 50
    __ESBMC_assume(x <= 50 ==> y == 50)
    # If x is greater than 50 then y = x
    __ESBMC_assume(x >  50 ==> y == x)
    # At the end of the loop x is not < 100
    __ESBMC_assume(x >= 100)

    assert(y == 100)
```

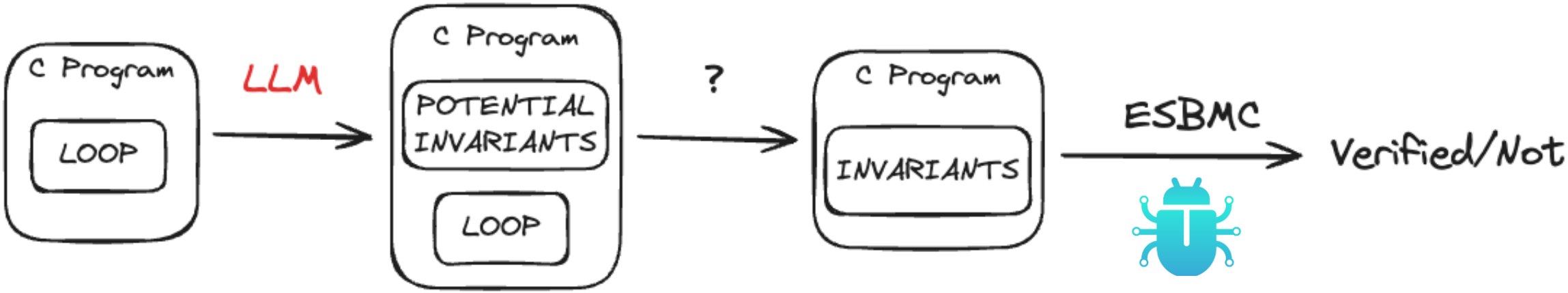
# Replace Loop with Loop Invariants



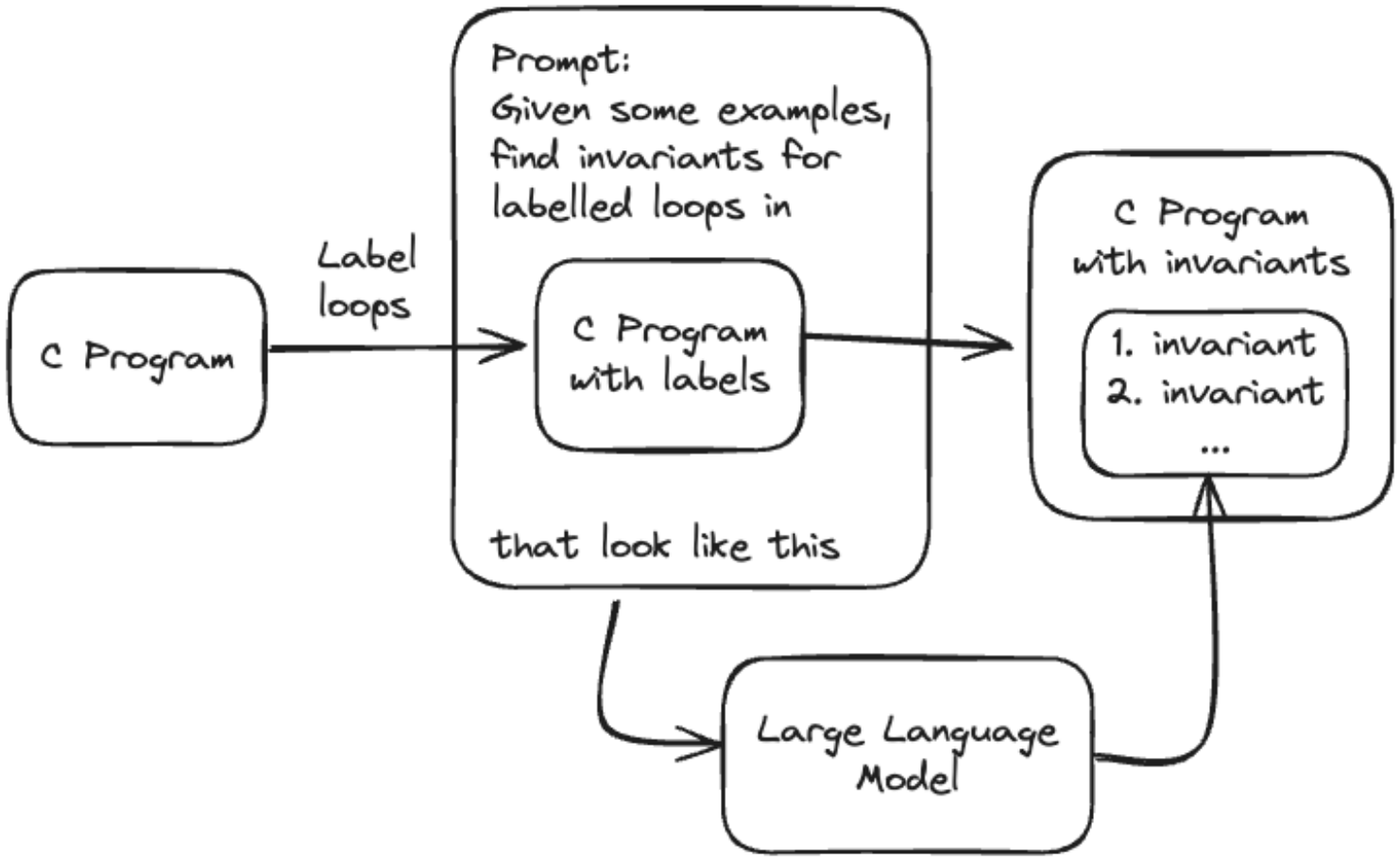
# Replace Loop with Loop Invariants



# Generating Potential Invariants



# Generating Potential Invariants



# Prompt Engineering for Invariant Generation

```
<Examples with explanation>

Based on these examples provided above can you
generate  a C invariant for the following code,

<Labelled Input C Program>

Print an invariant for this loop that holds in the
form '__invariant(...);'. They should help prove the
assertions. You can utilise '&&' or '||' if required.
No explanation.  Your answer should be in the form
'__invariant(. . .);'
```

Chain of Thought (CoT) Prompt Engineering

Small number of different prompts that vary by their examples e.g. :

- Single Invariant
- Multiple Invariants

Full prompts (more explanation), constrained prompts (less explanation), and combined prompts

# Prompt Engineering for Invariant Generation

```
int main()
{
  int x = 0;
  int y = 50;
  while (x < 100) {
    x = x + 1;
    if(x > 50) {
      y = y + 1;
    }
  }
  assert(y == 100);
}
```

```
Provide two loop invariants for the loop in the
program that are strong enough to prove the
assertion at the end of the program.

Answer:
Invariant 1: 0 <= x && x <= 100

This ensures that x is always within the expected
range throughout the loop's execution.

Invariant 2: x <= 50 && y == 50 || x > 50 && y == x

This captures the relationship between x and y:
If x is less than or equal to 50, y remains 50.
If x is greater than 50, y increments in tandem with
x, effectively making y equal to x.
```

# LLMs Can Get it Wrong!

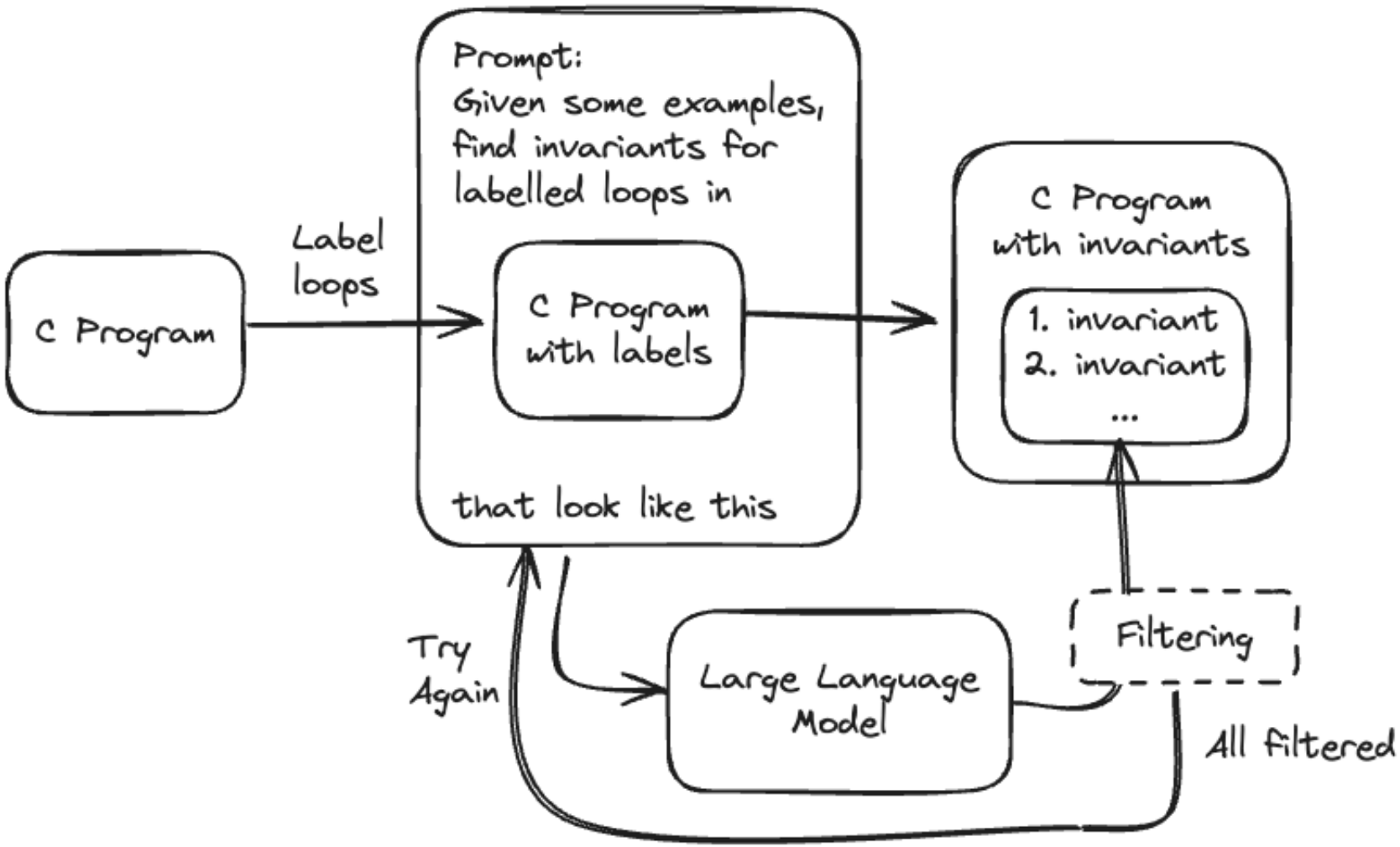
```
__invariant(c > 1 && c < 2 ==> i == 0);
__invariant(...);
__invariant(
This is the invariant
__invariant(...); for
the c program.
```

```
int x;
int y;
__invariant(x > max);
```

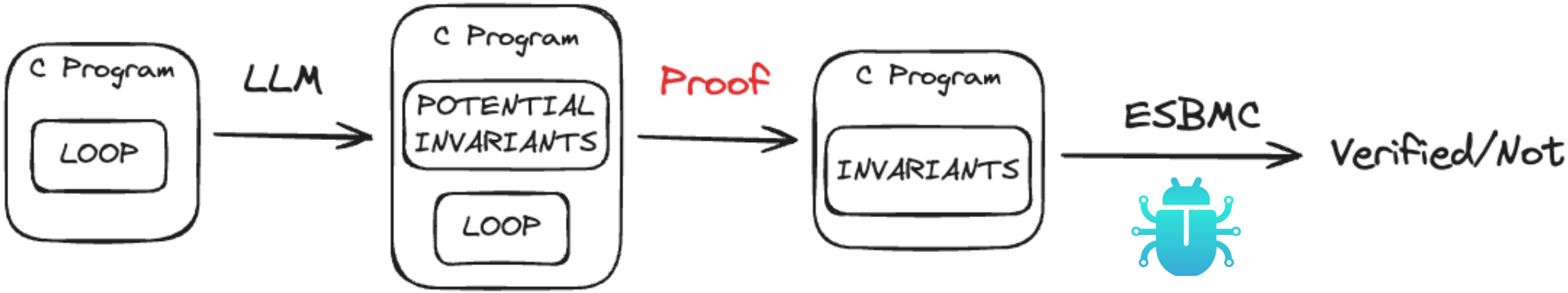
How to get better answers?

1. Constrain the prompt (no explanation as it can be confusing)
2. Filter the answers (simple regex filtering for now)

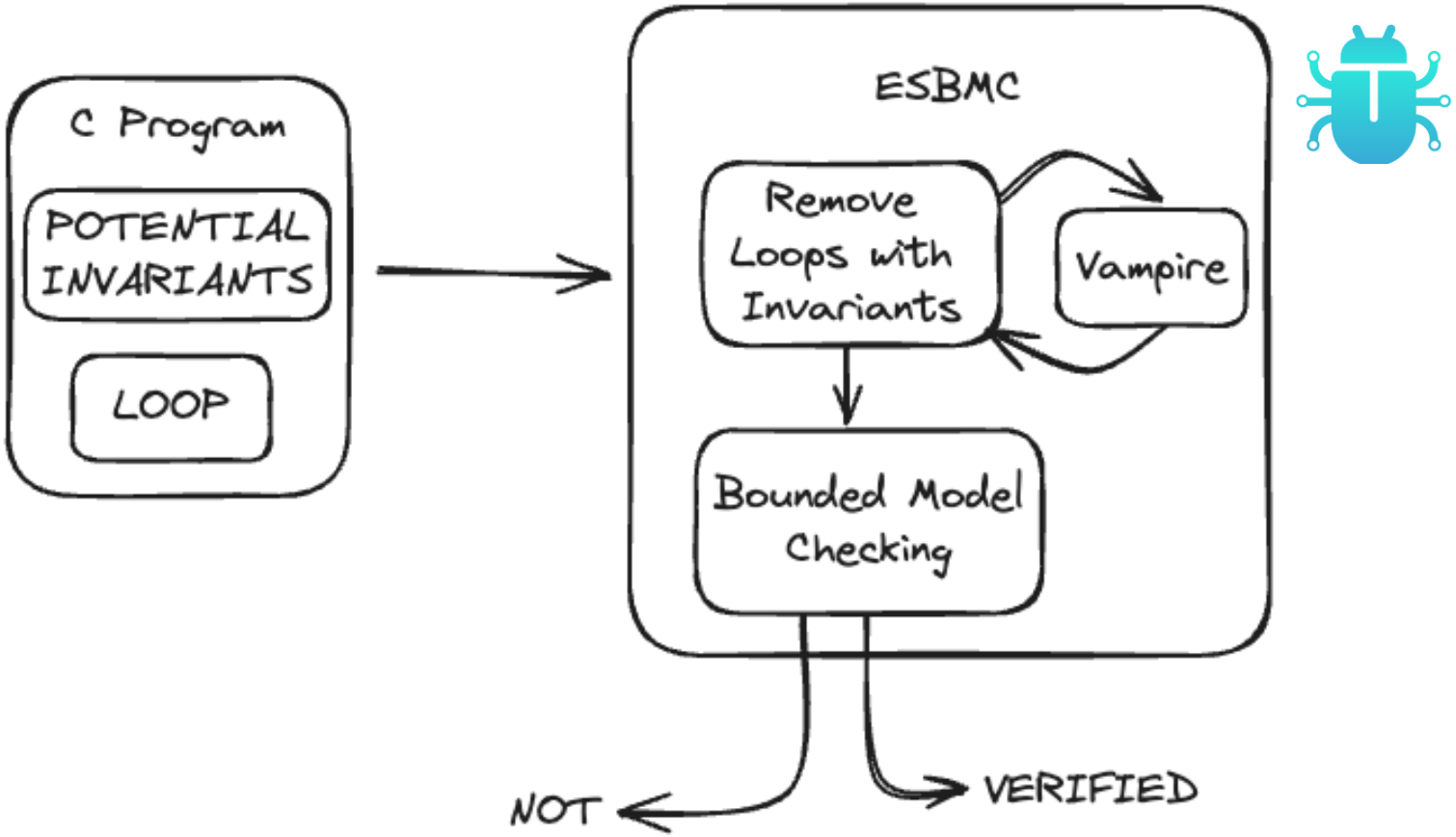
# Generating Potential Invariants



# Proving Potential Invariants



# Proving Potential Invariants



# Sound but not Complete

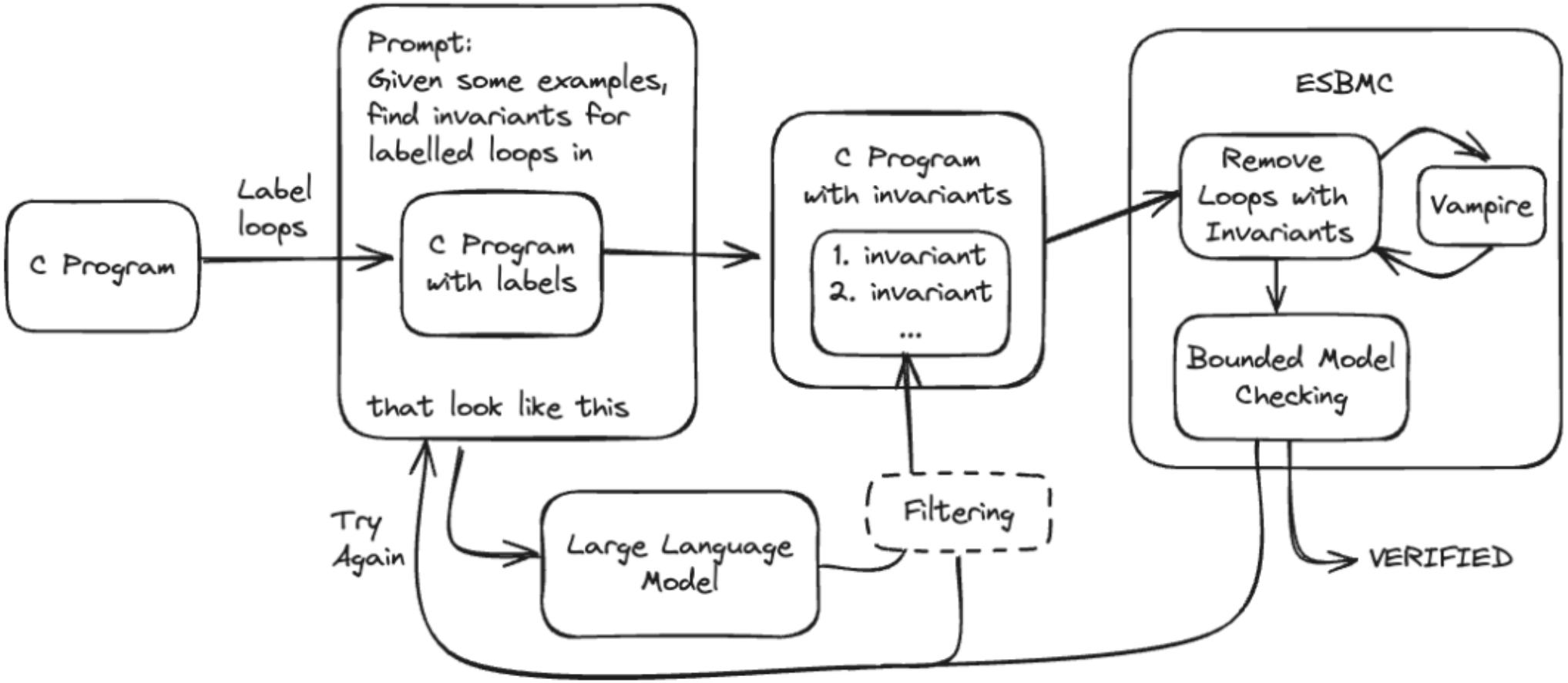
**The approach is Sound:**

Valid means that for any input, if the program terminates (we check partial correctness), then the assertions hold. No invalid assertions can be proven; there are no false positives

**The approach is not Complete**:

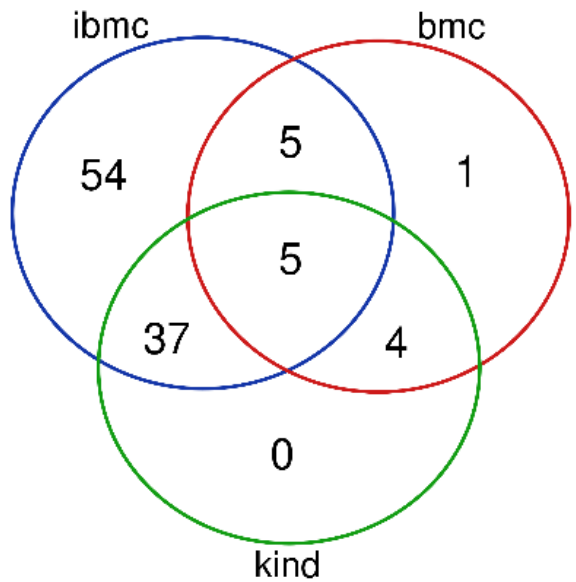
The LLM may never generate the invariants needed to prove a valid assertion, i.e., that sufficiently capture the semantics of the loop

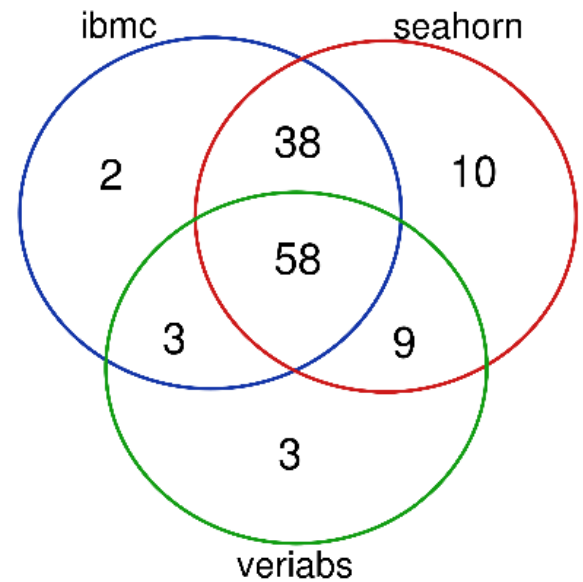
# The ESBMC ibmc Tool



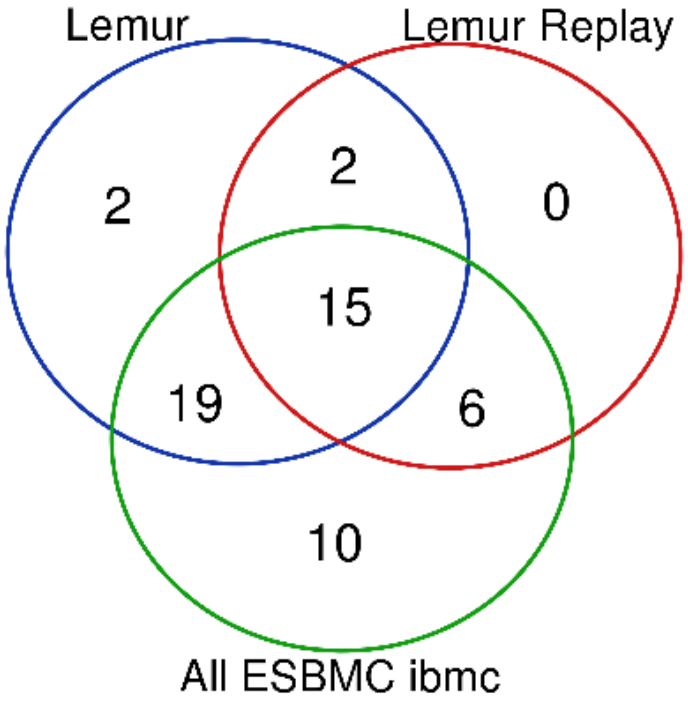
# Comparing to Other Verifiers

| Tool | Solved | Unique |
|---|---|---|
| ESBMC bmc | 16 | 0 |
| ESBMC k-induction | 46 | 0 |
| SeaHorn | 115 | 10 |
| VeriAbs | 73 | 2 |
| ESBMC ibmc | 101 | 2 |



# Comparison to LEMUR




LEMUR is similar as it also uses an LLM to suggest invariants.

We could not run LEMUR so replayed their invariants using our extended ESBMC

Using their invariants we verified 6 more programs

Using our invariants we verified 10 more programs

Potential lessons from both sides


# Distinguished Paper Award

# ASE 2024

**IEEE/ACM International Conference on Automated Software Engineering**

October 27 - November 1 Sacramento, California

*Presented to*

Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, Lucas C. Cordeiro

*for*

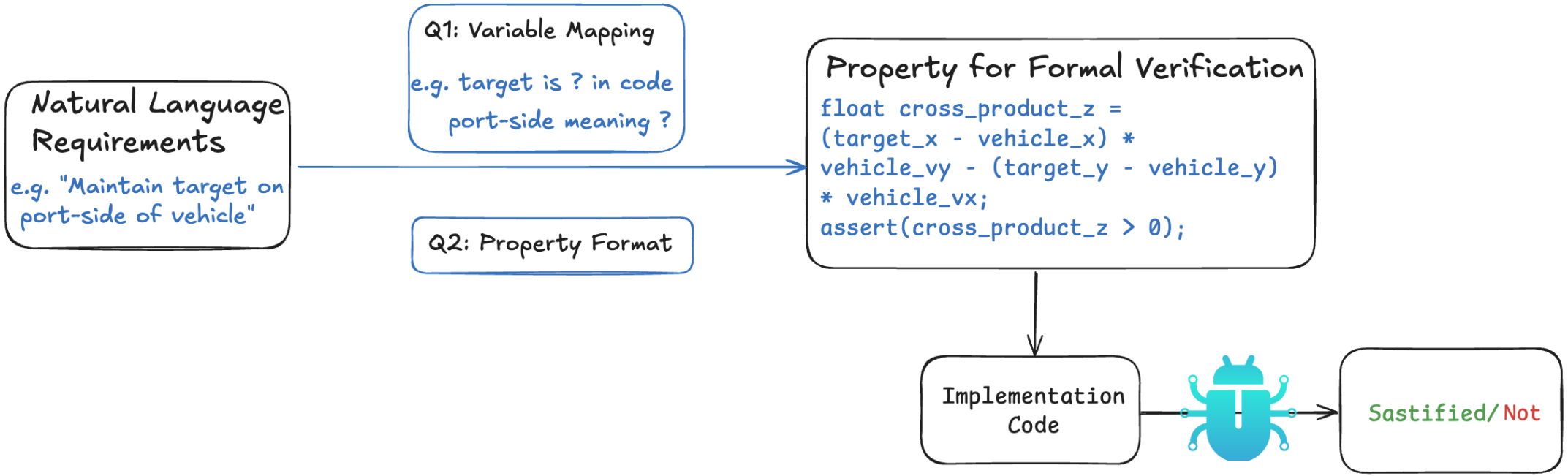
## LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling

Vladimir Filkov
General Chair

Baishakhi Ray
Research PC Co-Chair

Minghui Zhou
Research PC Co-Chair

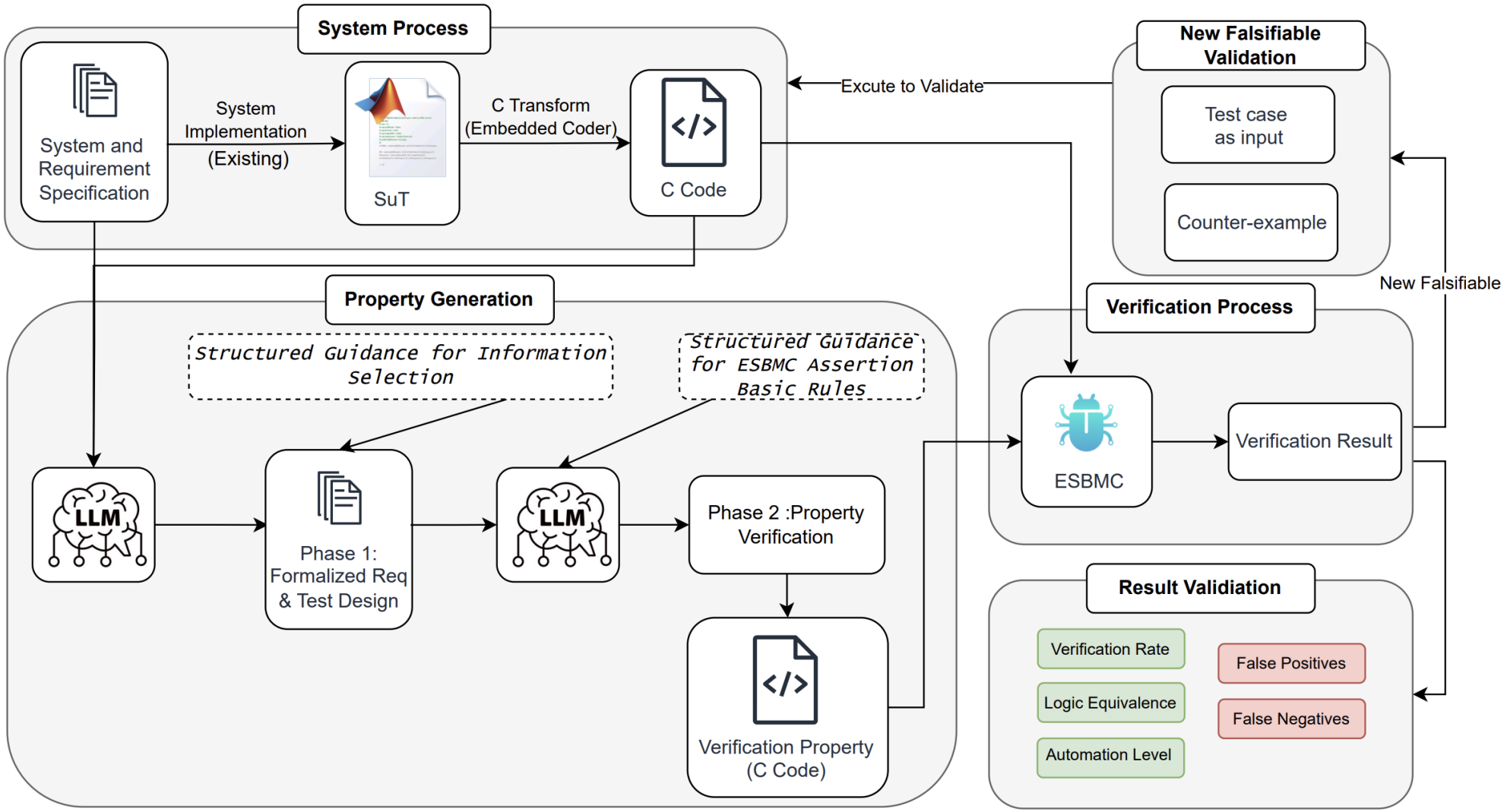
# Future Work: making rigorous verification as easy and fast as AI-assisted development




BMC can verify functional correctness, but requires an expert to write the property

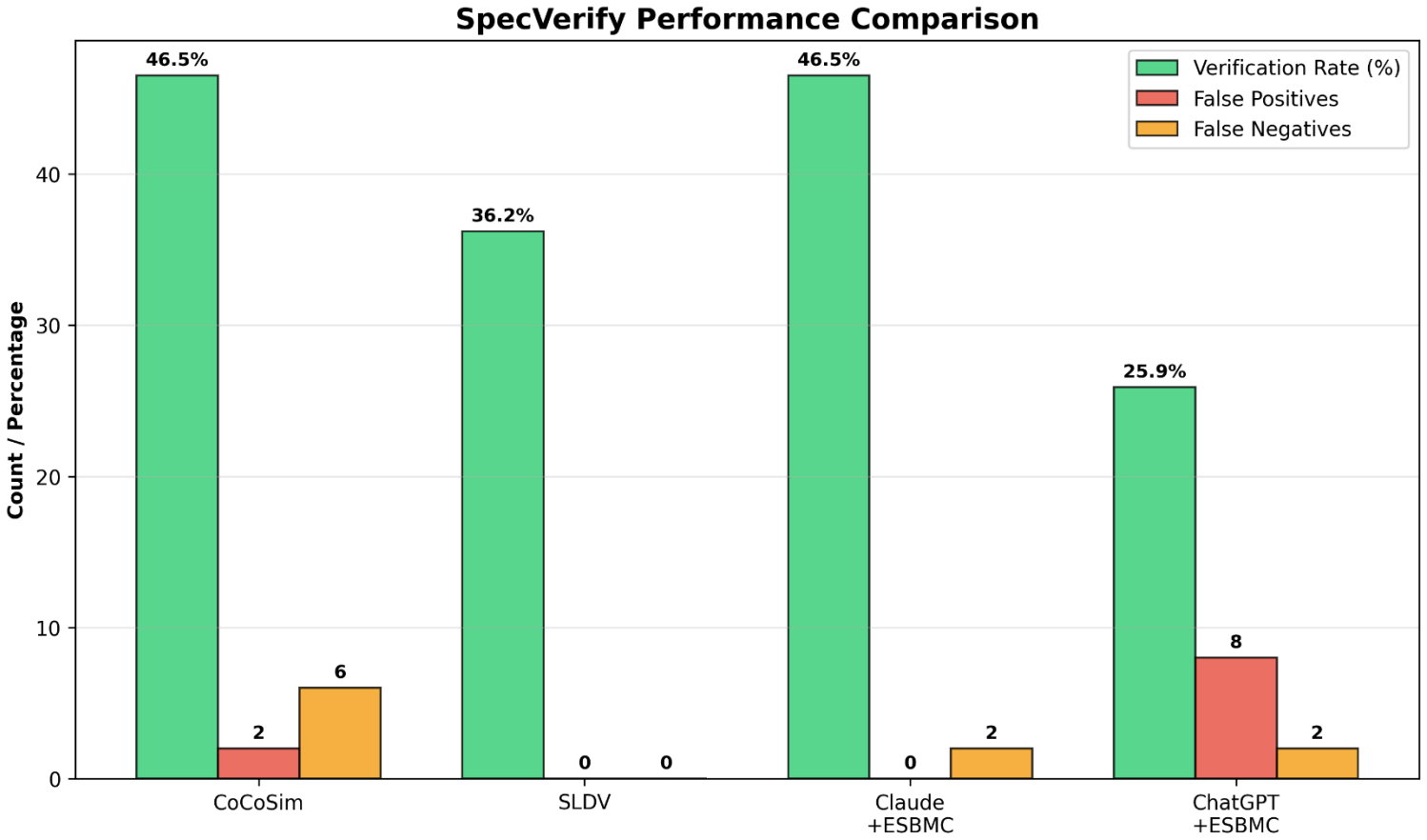
Variable and logic mapping are time-consuming for property generation

# SpecVerify: LLM-Aided Requirements




Wang, W., Farrell, M., Cordeiro, L. C., & Zhao, L. (2025, September). *Supporting Software Formal Verification with Large Language Models: An Experimental Study*. In 2025 IEEE 33rd International Requirements Engineering Conference (RE) (pp. 423-431).

# Comparing with Human-Expert Verification Result




**Key Achievements:**

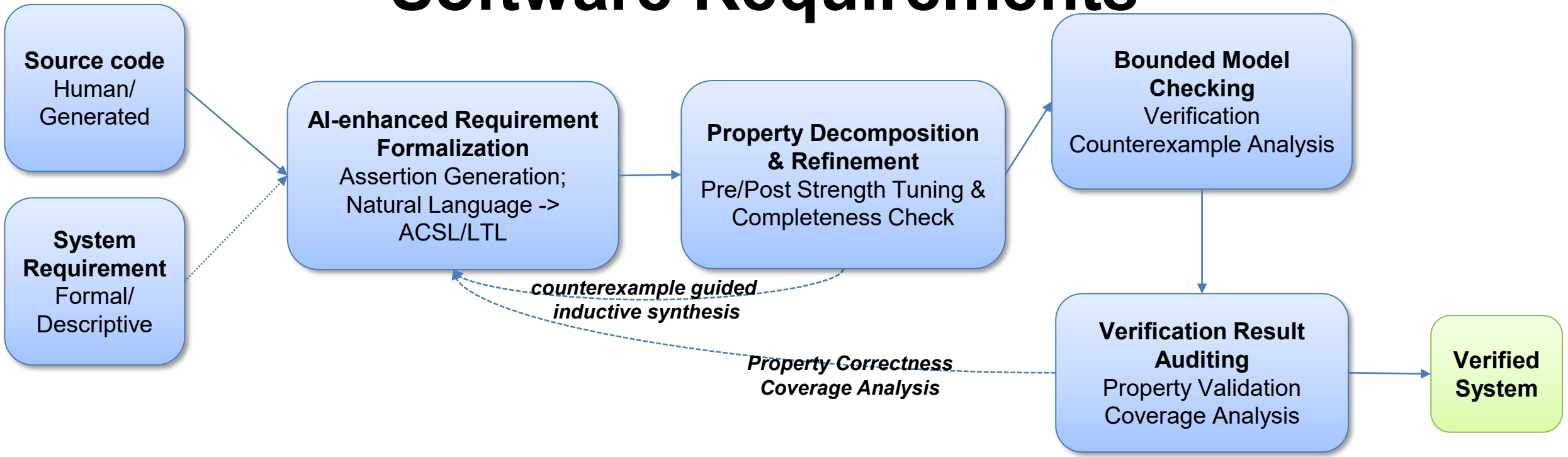
- 79.31% logical equivalence with expert-verified properties
- **28% better** than SLDV
- 46.5% verification rate (matches CoCoSim baseline)
- Superior accuracy: 2 false negatives, 0 false positives

**Key Advantage:**

- Full automation - no manual variable mapping
- Found 2 new floating-point errors
- Direct implementation-level verification


Wang, W., Farrell, M., Cordeiro, L. C., & Zhao, L. (2025, September). *Supporting Software Formal Verification with Large Language Models: An Experimental Study*. In 2025 IEEE 33rd International Requirements Engineering Conference (RE) (pp. 423-431).

# Roadmap: Automatically Formal Verification of Software Requirements




**1 Formalization** — Completed

Automated requirements formalization (SpecVerify)

**2 Decomposition** — In Progress

System-level property decomposition with refinement

**3 Auditing** — Planned

Verification result auditing and quality assurance

# Acknowledgements









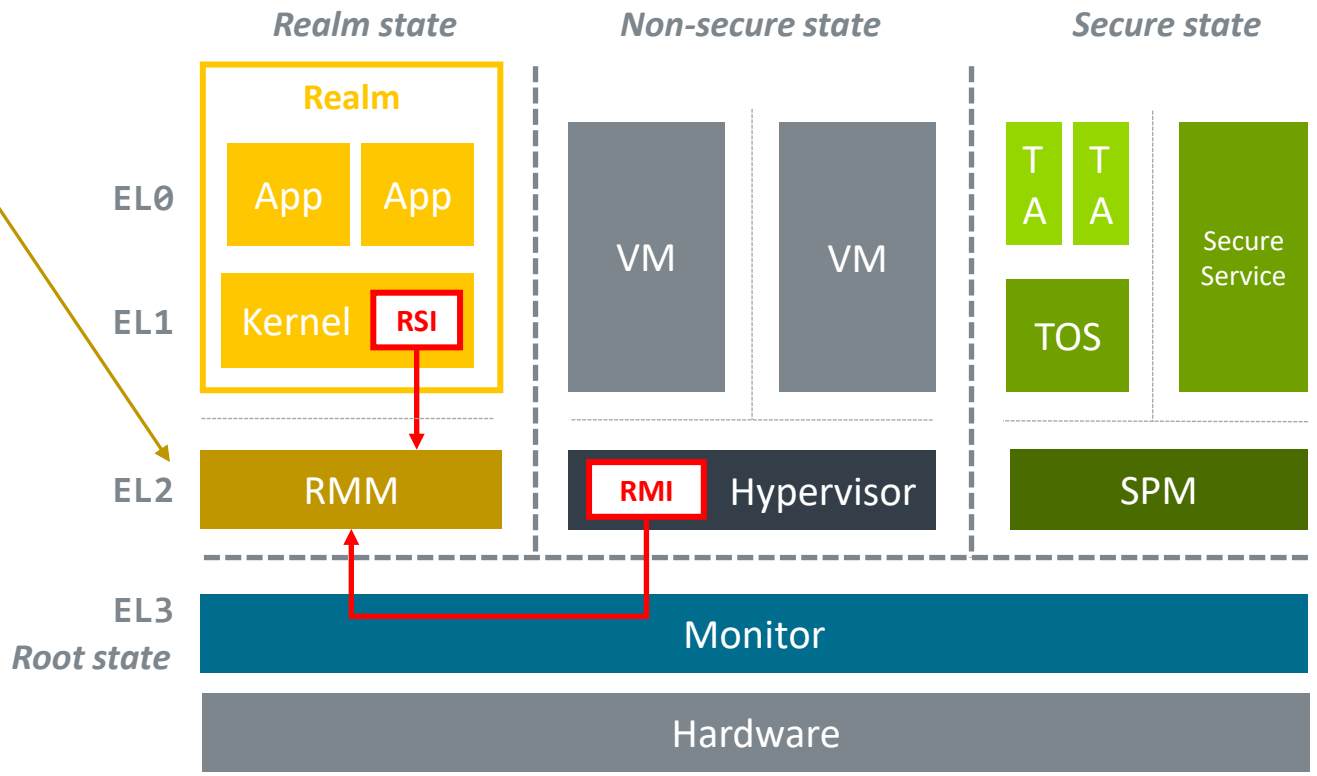





# Verifying Components of Arm® Confidential Computing Architecture with ESBMC

## Realm Management Monitor (RMM)

- Provides services to Host and Realm
  - Contains no policy
  - Performs no dynamic memory allocation
- Realm Management Interface (RMI)
  - Secure Monitor Call Calling Convention (SMCCC) interface called by Host
  - Create/destroy Realms
  - Manage Realm memory, manipulating stage 2 translation tables
- Realm Services Interface (RSI)
  - SMCCC interface called by Realm
  - Measurement and attestation
  - Handshakes involved in some memory management flows



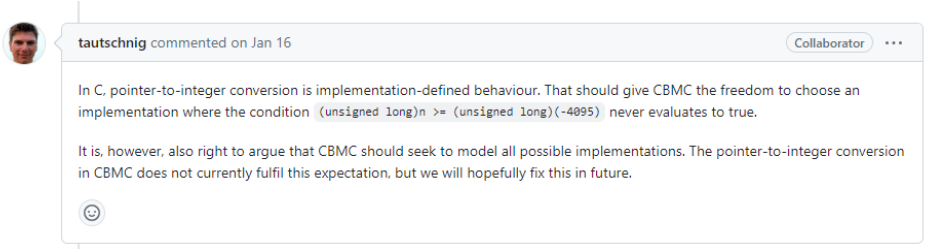

Arm CCA is an architecture that provides Protected Execution Environments called Realms

Wu, T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L., Verifying components of Arm® Confidential Computing Architecture with ESBMC. In SAS 2024

# Verifying Components of Arm® Confidential Computing Architecture with ESBMC

| Test_benchmarks | esbmc multi | cbmc multi |
|---|---|---|
| RMI_REC_DESTROY | 20 | 20 |
| RMI_GRANULE_DELEGATE | safe | safe |
| RMI_GRANULE_UNDELEGATE | 1 | 1 |
| RMI_REALM_ACTIVATE | **3** | **safe** |
| RMI_REALM_DESTROY | **15** | **1** |
| RMI_REC_AUX_COUNT | 1 | 1 |
| RMI_FEATURES | safe | safe |
| RMI_DATA_DESTROY | **>=24** | **22** |

```
#include <assert.h>
extern int nondet_int();
int main() {
  int m = nondet_int();
  int *n = &m;
  if((unsigned long)n >= (unsigned long)(-4095))
    assert((unsigned int)(-1 * (long)n) < 6);
  int a = -2048;
  if((unsigned long)a >= (unsigned long)(-4095))
    assert((unsigned int)(-1 * (long)a) < 6);
}
```




https://github.com/diffblue/cbmc/issues/8161

Wu, T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L., Verifying components of Arm® Confidential Computing Architecture with ESBMC. In SAS 2024

# Intel Core Power Management Firmware

Intel routinely employs ESBMC to **automate firmware analysis**

ESBMC has been applied to the **Authenticated Code Module**, where it **found over 30 vulnerabilities**

ESBMC is part of the CI pipeline for developing microcode for the Core family of processors

## P6 Microcode Can Be Patched

*Intel Discloses Details of Download Mechanism for Fixing CPU Bugs*

*"Taking an unusual approach to fixing bugs, Intel has implemented a microcode patch capability in its P6 processors, including Pentium Pro and Pentium II. This capability allows the microcode to be altered after the processor is fabricated, repairing bugs that are found after the processor is designed. Intel has already used this feature several times to correct minor bugs, and in the future, it may save the company from recalling CPUs if a major problem is discovered."*

Menezes, R., Manino, E., Shmarov, F., Aldughaim, M., de Freitas, R. Cordeiro, L.: *Interval Analysis in Industrial-Scale BMC Software Verifiers: A Case Study*. CoRR abs/2406.15281 (2024)

# WolfMQTT Verification

- **wolfMQTT** library is a client implementation of the MQTT protocol written in C for **IoT devices**

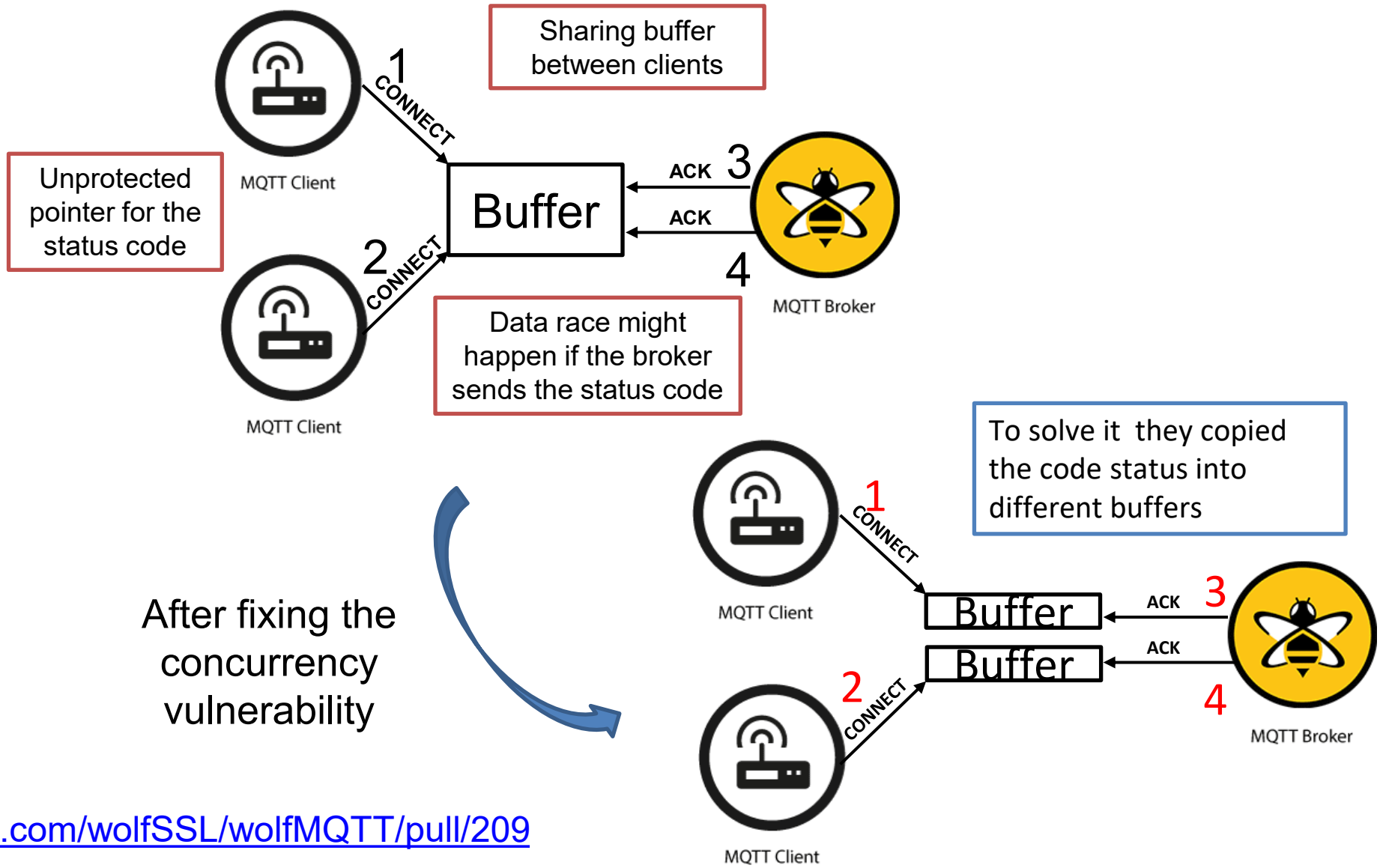
`subscribe_task` and `waitMessage_task` are called through different threads accessing `packet_ret`, causing a data race in `MqttClient_WaitType`

Here is where the data race might happen! Unprotected pointer

```
Int main(){
Pthread_t th1, th2;
static MQTTCtx mqttCtx;
pthread_create(&th1, subscribe_task, &mqttCtx))
pthread_create(&th2, waitMessage_task, &mqttCtx))}

static void *subscribe_task(void *client){
.....
MqttClient_WaitType(client,msg,MQTT_PACKET_TYPE_ANY,
0,timeout_ms);
.....}
static void *waitMessage_task(void *client){
…
MqttClient_WaitType(client, msg, MQTT_PACKET_TYPE_ANY,
0,timeout_ms);
.....}
static int MqttClient_WaitType(MqttClient *client,
void *packet_obj,
   byte wait_type, word16 wait_packet_id, int timeout_ms)
{
.....
            rc = wm_SemLock(&client->lockClient);
            if (rc == 0) {
                if (MqttClient_RespList_Find(client,
(MqttPacketType)wait_type,
                        wait_packet_id, &pendResp)) {
                    if (pendResp->packetDone) {
                        rc = pendResp->packet_ret;
.....}
```

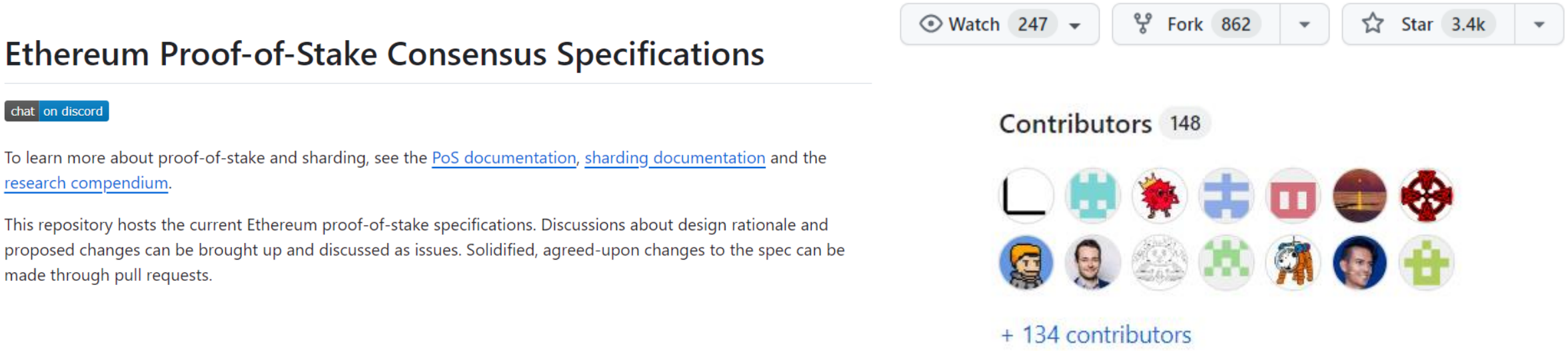
# WolfMQTT Verification




https://github.com/wolfSSL/wolfMQTT/pull/209

# Ethereum Consensus Specifications

- Consensus protocol dictates how the participants in Ethereum agree on the validity of transactions and the system's state
- Git repository with **Markdown** documents describing specifications
- Infrastructure to generate **Python** libraries from Markdown



# ESBMC-Python Benchmark

## Ethereum Consensus Specification

Markdown

consensus-specs / specs / phase0 / **beacon-chain.md**

Preview | Code | Blame | 1939 lines (1617 loc) · 71.4 KB

**Math**

`integer_squareroot`

```
def integer_squareroot(n: uint64) -> uint64:
    """
    Return the largest integer ``x`` such that ``x**2 <= n``.
    """
    x = n
    y = (x + 1) // 2
    while y < x:
        x = y
        y = (x + n // x) // 2
    return x
```

`xor`

```
def xor(bytes_1: Bytes32, bytes_2: Bytes32) -> Bytes32:
    """
    Return the exclusive-or of two 32-byte strings.
    """
    return Bytes32(a ^ b for a, b in zip(bytes_1, bytes_2))
```

eth2spec Python Library

mainnet.py

lib > python3.10 > site-packages > eth2spec-1.4.0b4-py3.10.egg > eth2spec > bellatrix > mainnet.py > integer_squareroot

```
1461
1462
1463 def integer_squareroot(n: uint64) -> uint64:
1464     """
1465     Return the largest integer ``x`` such that ``x**2 <= n``.
1466     """
1467     x = n
1468     y = (x + 1) // 2
1469     while y < x:
1470         x = y
1471         y = (x + n // x) // 2
1472     return x
1473
1474
1475 def xor(bytes_1: Bytes32, bytes_2: Bytes32) -> Bytes32:
1476     """
1477     Return the exclusive-or of two 32-byte strings.
1478     """
1479     return Bytes32(a ^ b for a, b in zip(bytes_1, bytes_2))
1480
1481
1482 def bytes_to_uint64(data: bytes) -> uint64:
1483     """
1484     Return the integer deserialization of ``data`` interpreted as ``ENDIANNESS``-endian.
1485     """
1486     return uint64(int.from_bytes(data, ENDIANNESS))
```

Python Application

integer_squareroot.py

eth2bmc > samples > helpers > math > integer_squareroot.py > ...

```
1 from eth2spec.bellatrix import mainnet as spec
2 from eth2spec.utils.ssz.ssz_typing import (uint64)
3
4 x = uint64(16)
5 assert spec.integer_squareroot(x) == 4
6
7 x = uint64(25)
8 assert spec.integer_squareroot(x) == 5
```

ESBMC

Verification Output

# Handle `integer_squareroot` bound case #3600

Merged hwwhww merged 3 commits into `dev` from `integer_squareroot` 2 weeks ago

Conversation 4 | Commits 3 | Checks 15 | Files changed 5

**hwwhww** commented 2 weeks ago • edited — Contributor

**Credits to the University of Manchester Bounded Model Checking (BMC) project team: Bruno Farias, Youcheng Sun, and Lucas C. Cordeiro for reporting this issue!** 🙏 💯

This team is an Ethereum Foundation ESP "Bounded Model Checking for Verifying and Testing Ethereum Consensus Specifications (FY22-0751)" project grantee. They used ESBMC model checker to find this issue.

## Description

`integer_squareroot` raises ValueError exception when `n` is maxint of `uint64`, i.e., `2**64 - 1`.

However, we only use `integer_squareroot` in

1. `integer_squareroot(total_balance)`
2. `integer_squareroot(SLOTS_PER_EPOCH)`

With the current Ether total supply + EIP-1559, it's unlikely to hit the overflow bound in a very long time. ( 🤞 🔊 )

That said, it should be fixed to return the expected value.

https://github.com/ethereum/consensus-specs/pull/3600