Collaborators/funders:

- Systems and Software Security / FM Research Group
- ARM Centre of Excellence
- PPGEE, PPGI – UFAM
- Centre for Digital Trust and Society
- UKRI, EPSRC, EU Horizon and industrial partners

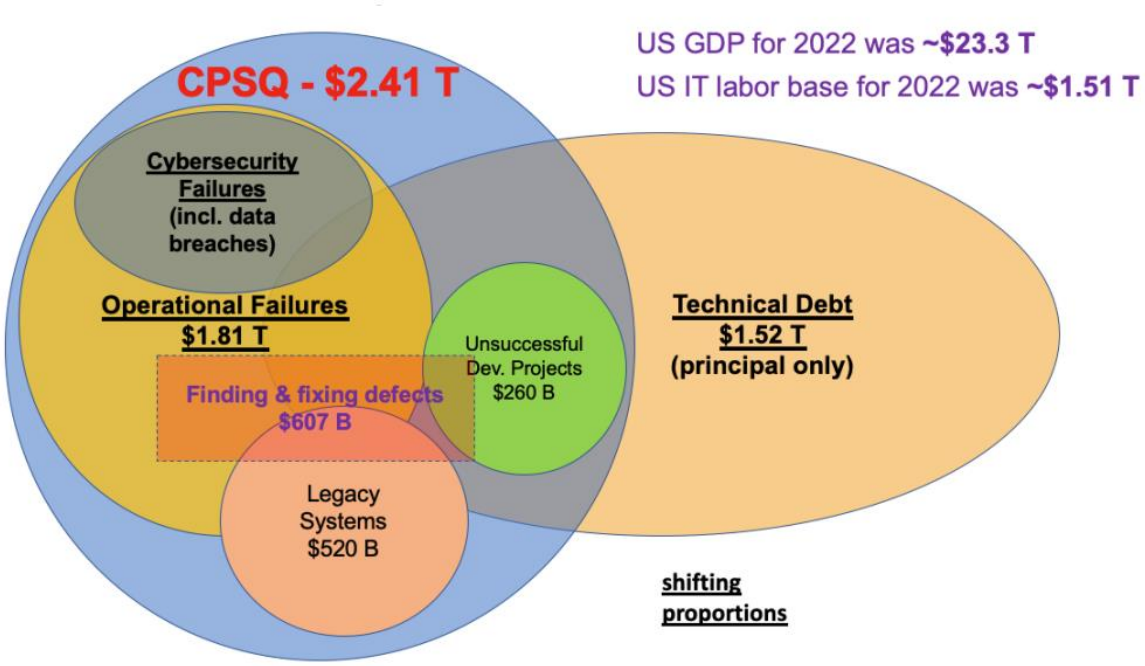
The University of Manchester

An Exploration of Automated Software Testing, Verification, and Repair using ESBMC and ChatGPT

Lucas Cordeiro
lucas.cordeiro@manchester.ac.uk
https://ssvlab.github.io/lucasccordeiro/

How much could software errors cost your business?

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



TD relies on temporary easy-to-implement solutions to achieve short-term results at the expense of efficiency in the long run

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

CISQ Consortium for Information & Software Quality™

Objective of this talk

Discuss **automated testing, verification, and repair** techniques to establish a **robust foundation for building secure software systems**

- Introduce a **logic-based automated reasoning platform** to find and repair **software vulnerabilities**
- Explain **testing, verification, and repair** techniques to build **secure software systems**
- Present recent advancements towards a **hybrid approach** to protecting against **memory safety and concurrency vulnerabilities**

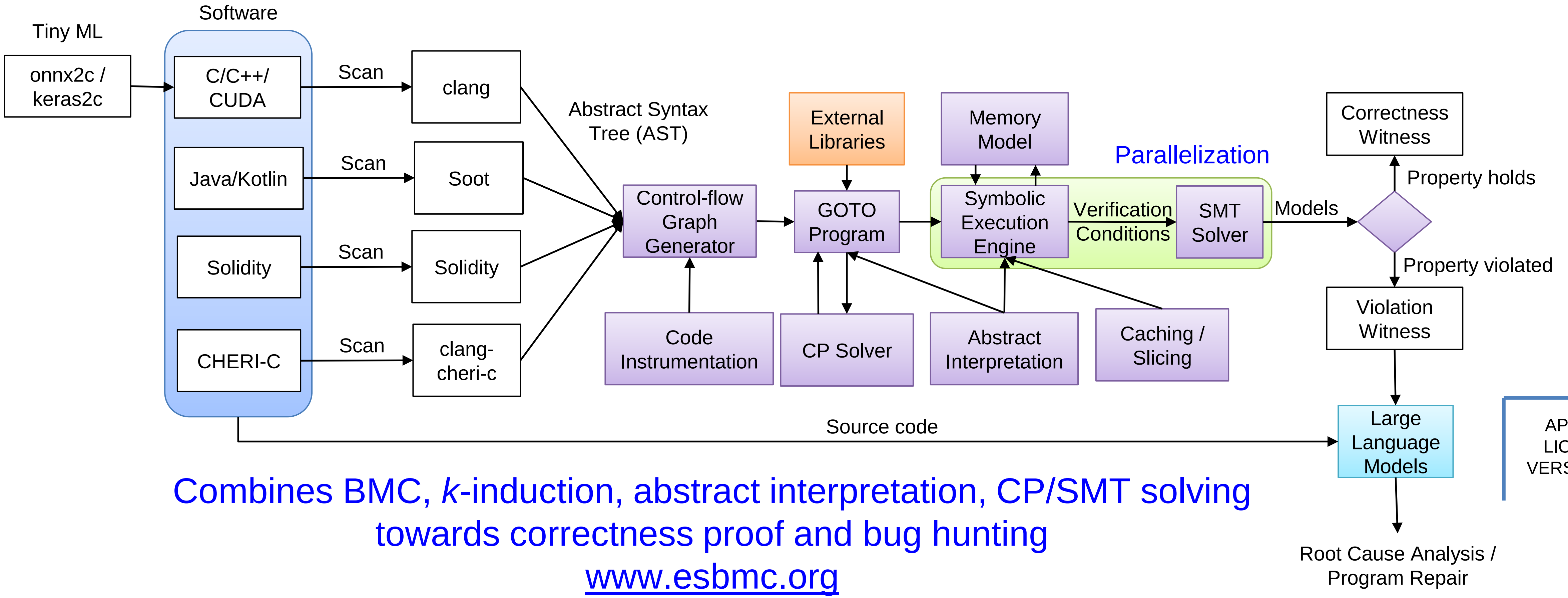
Research Questions

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

Can we leverage **program analysis/synthesis** to **discover and fix** more **software vulnerabilities** than existing state-of-the-art approaches?

ESBMC: An Automated Reasoning Platform

Logic-based automated reasoning for checking the **safety** and **security** of **AI and software systems**



Combines BMC, *k*-induction, abstract interpretation, CP/SMT solving towards correctness proof and bug hunting

www.esbmc.org

The Bitter Lesson by Rich Sutton
March 13, 2019

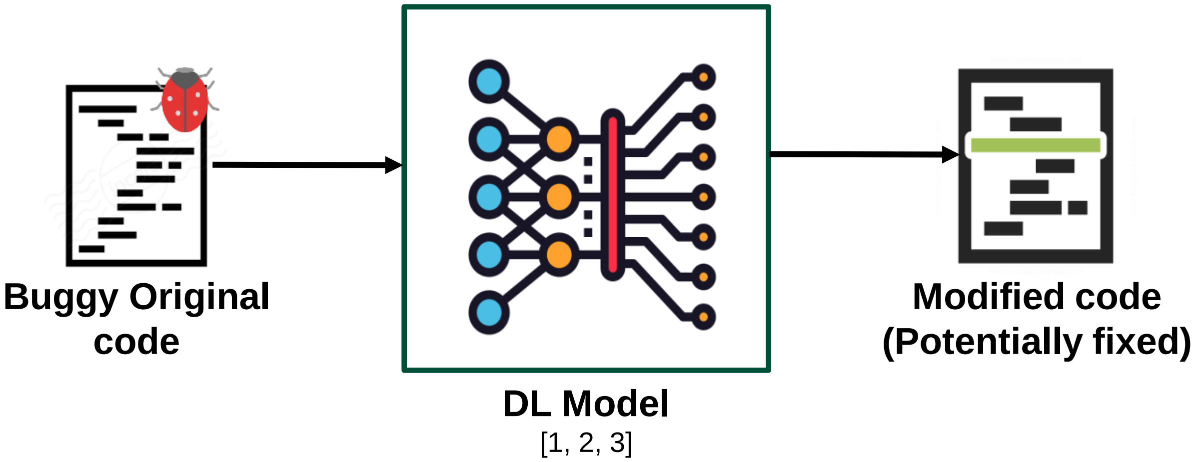
“The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation”

“The two methods that seem to scale arbitrarily in this way are *search* and *learning*”

Agenda

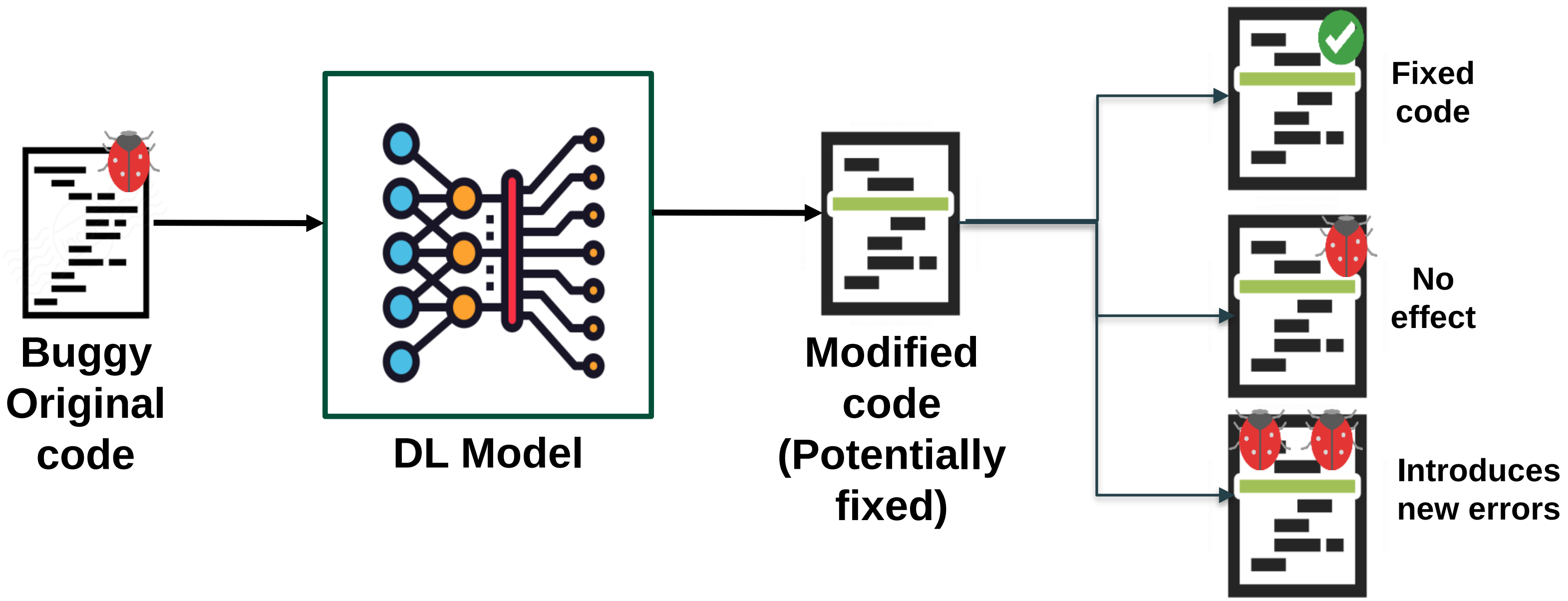
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Software Verification and Testing with the ESBMC framework
- Towards verification of C programs for CHERI platforms with ESBMC

Deep learning and Automated Program Repair

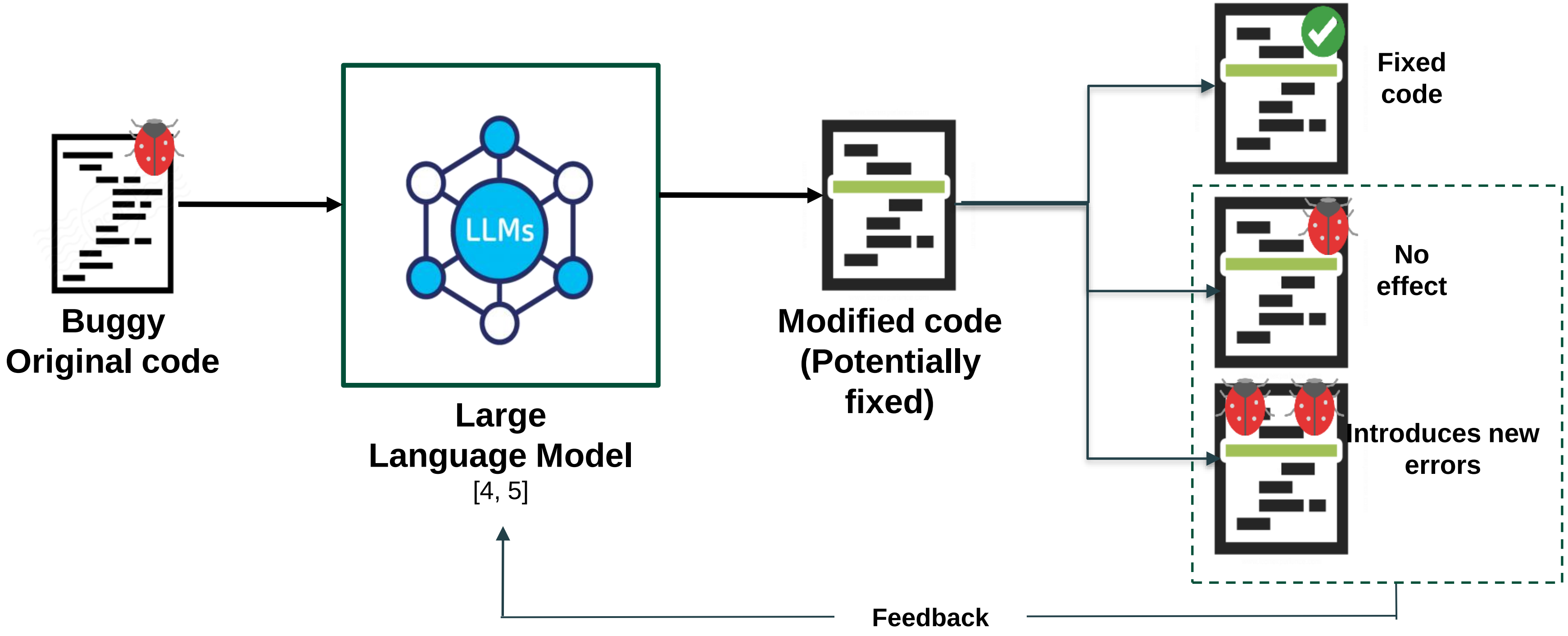


[1] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: End-to-End Program Repair with LLMs. arXiv e-prints. 2023 Mar:arXiv-2303.
[2] Li Y, Wang S, Nguyen TN. Dlfix: Context-based code transformation learning for automated program repair. InProceedings of the ACM/IEEE 42nd International Conference on Software Engineering 2020 Jun 27 (pp. 602-614).
[3] Gupta R, Pal S, Kanade A, Shevade S. Deepfix: Fixing common c language errors by deep learning. In Proceedings of the aaai conference on artificial intelligence 2017 Feb 12 (Vol. 31, No. 1).

Deep learning and Automated Program Repair



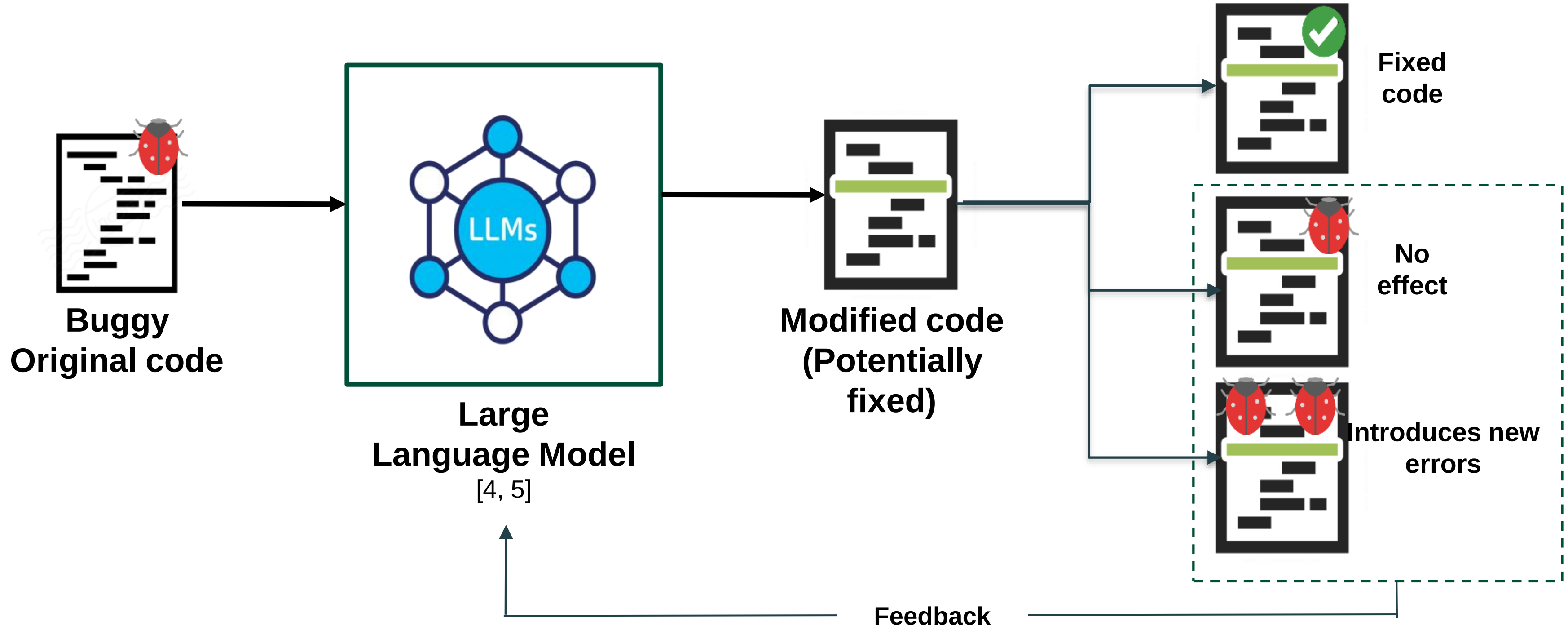
Large Language Models and Automated Program Repair



[4] Wang X, Wang Y, Wan Y, Mi F, Li Y, Zhou P, Liu J, Wu H, Jiang X, Liu Q. Compilable neural code generation with compiler feedback. arXiv preprint arXiv:2203.05132. 2022 Mar 10.

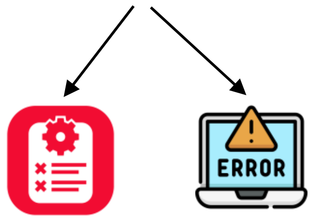
[5] Xia CS, Zhang L. Conversational automated program repair. arXiv preprint arXiv:2301.13246. 2023 Jan 30.

Large Language Models and Automated Program Repair

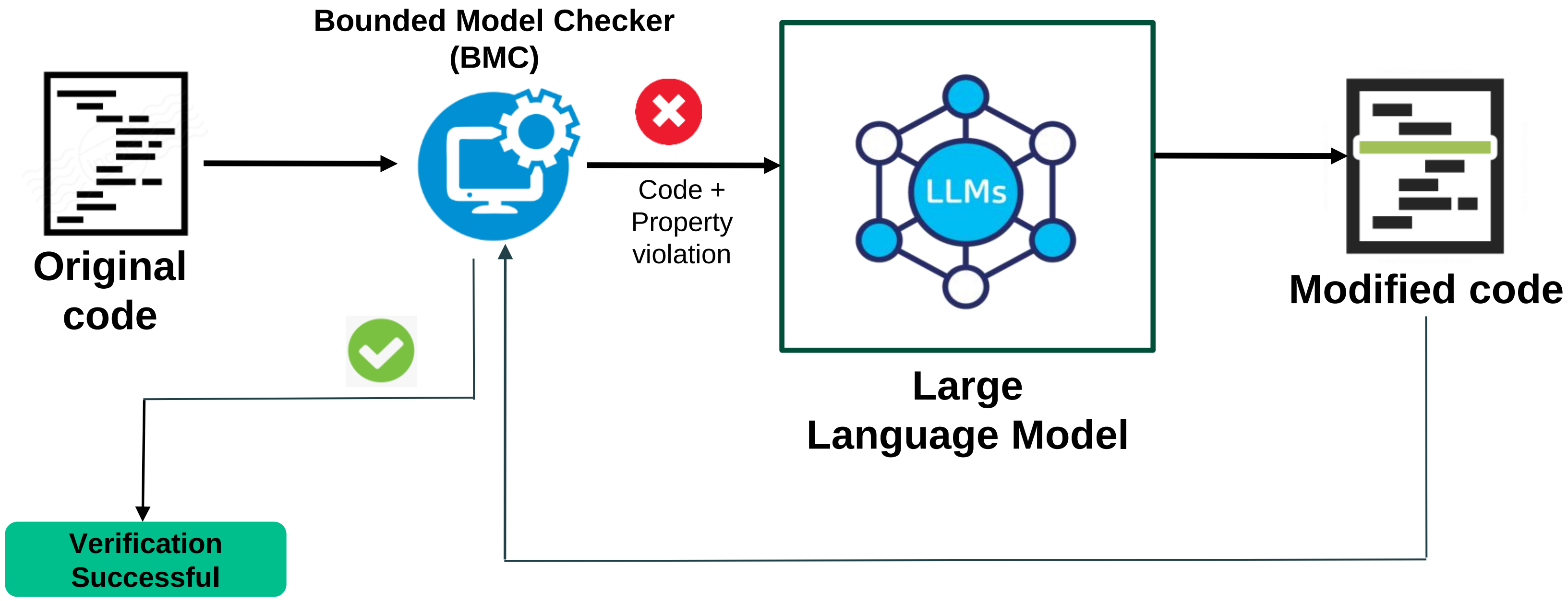


Compile-time error feedback misses run-time errors.

Test-suite may not be available.

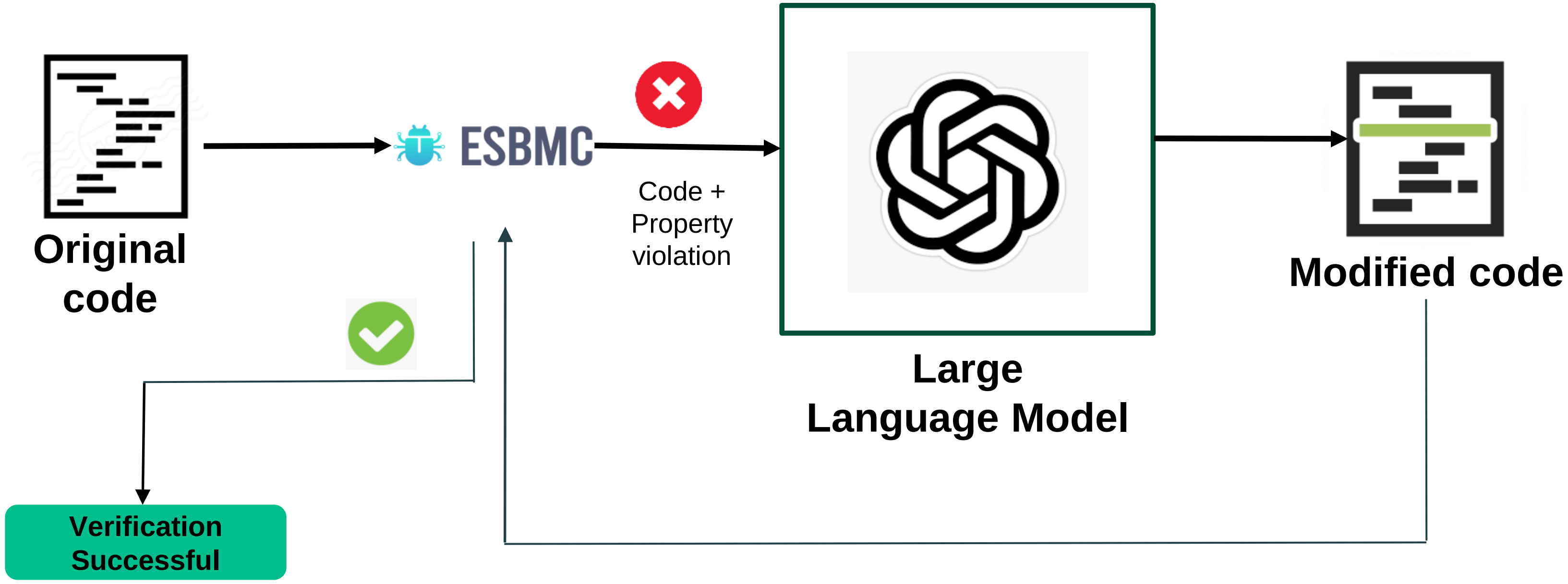


LLM + Formal Verification for Self-Healing Software



[6] Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M. Cordeiro, L.: A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification. Under review at the ACM Transactions on Software Engineering and Methodology, 2023.

LLM + Formal Verification for Self-Healing Software



LLM to Find Software Vulnerabilities

C++ program example

```
int main() {
    int x=77;
    int y=x*x*x;
    int z=y*y;
    unsigned int r= z/1000;
    printf("Result %d\n", r);
    return 0;
}
```

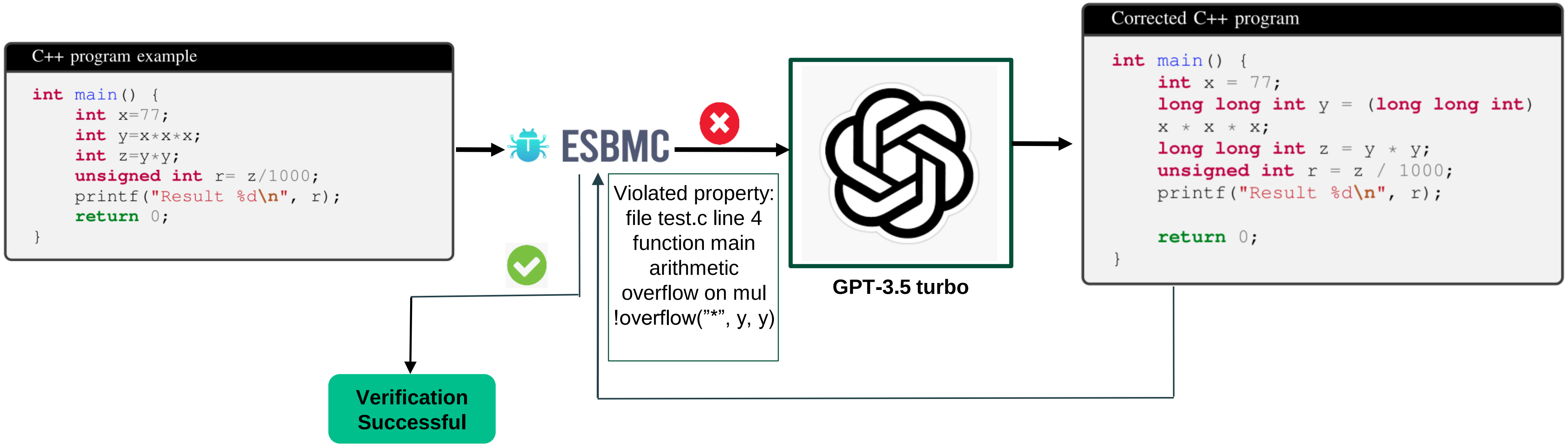
While we were in the process of preparing this presentation, if we asked GPT-3.5 *"Is there any problem with this code?"*, the response was an incorrect answer:

GPT-3.5 turbo

LLM answer:

"There is no problem with this code. The code correctly calculates the value of z as the square of y and then calculates the value of r as z divided by 1000. Finally, the code prints the result of r to the console. The calculations do not result in an integer overflow or any other issues, and the program should run correctly."

LLM + Formal Verification for Self-Healing Software



Experimental Evaluation

Set-up

Code Generation

- **Processor:** AMD Ryzen Threadripper PRO 3995WX
- **Cores:** 16
- **RAM:** 256 GB

Code Repair

- **Model:** MacBook Pro (2017)
- **RAM:** 16 GB RAM of LPDDR3 RAM (2133 MHz)
- **Processor:** 2.5 GHz Intel Core i7-7660U

Benchmarks

Generate 1000 programs with GPT-3.5 turbo with the following prompt.

Code generation prompt

Generate a minimum of 10 and a maximum of 50 lines of C code. Use at least two functions. Use strings, arrays, bit manipulations, and string manipulations inside the code. Be creative! Always include every necessary header. Only give me the code without any explanation. No comment in the code.

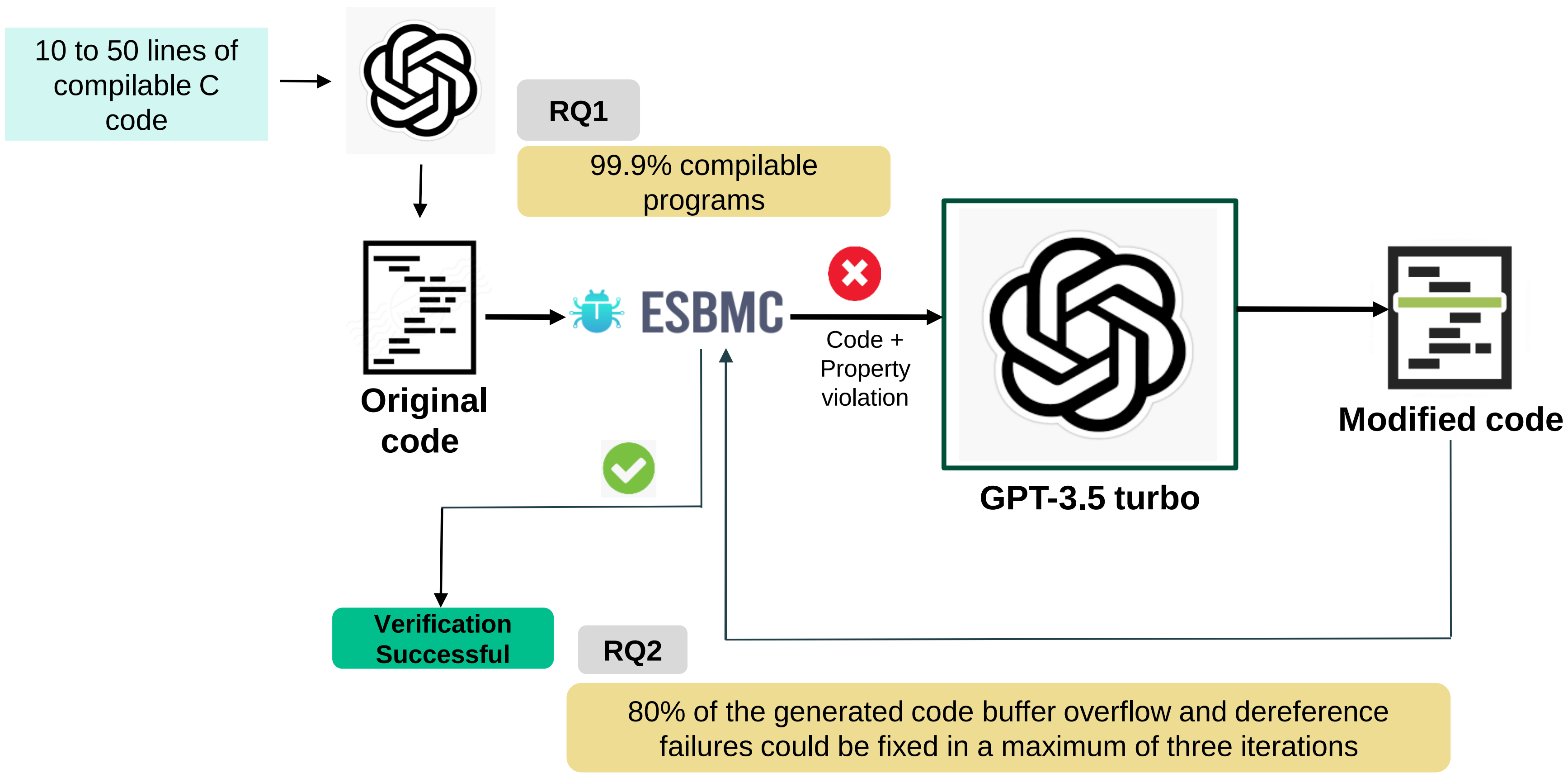
Objectives

To answer the following research questions.

RQ1: (Code generation) Are the state-of-the-art GPT models capable of producing compilable, semantically correct programs?

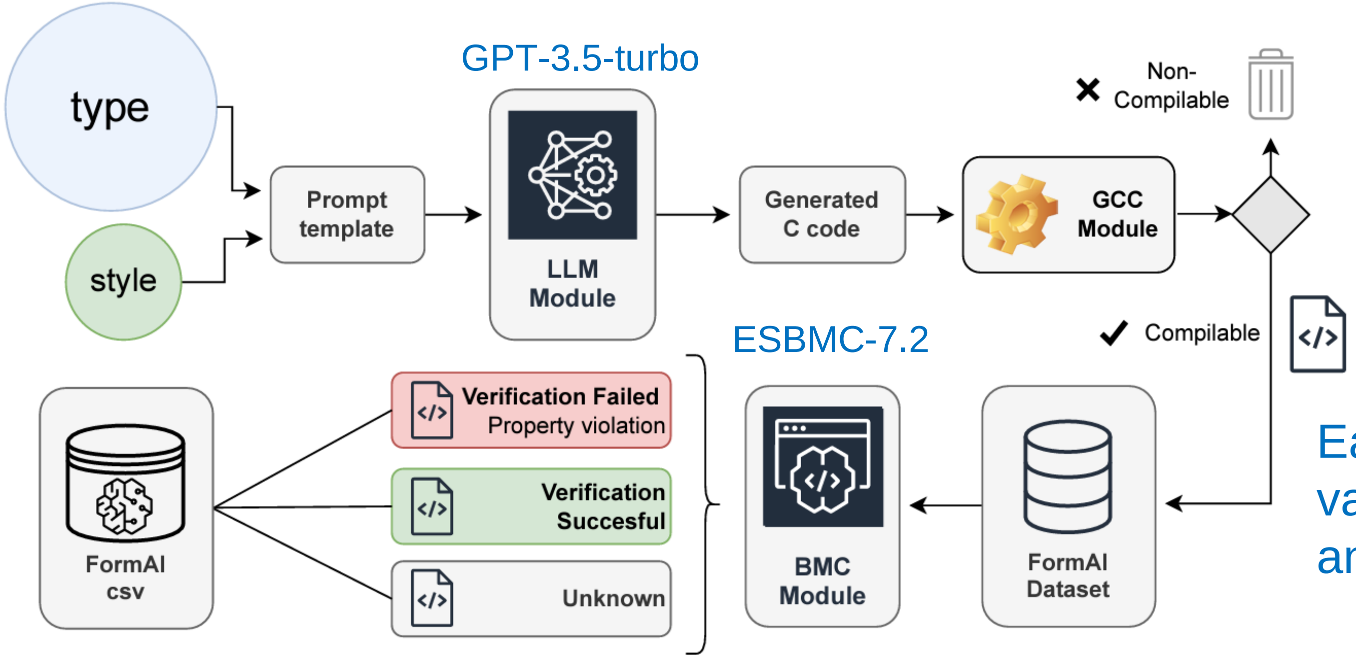
RQ2: (Code repair) Can external feedback improve the bug detection and patching ability of the GPT models?

Experimental Results



The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification

- The first AI-generated repository consisting of **112k independent and compilable C programs**



Each program varies between 50 and 600 lines

- It covers diverse programming tasks from **network management and table games to string manipulation**

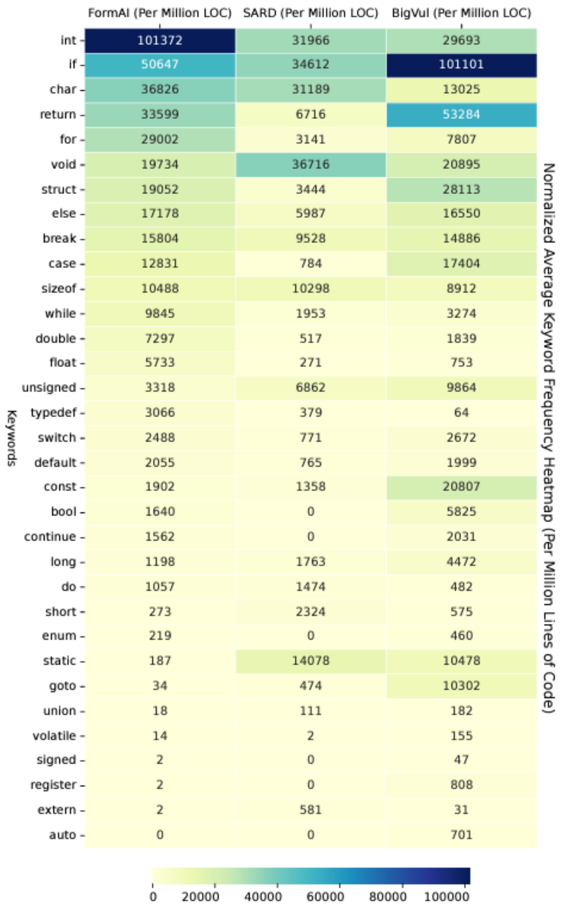
Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M., Cordeiro, L., Mavroeidis, V.: The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification. Accepted at ACM PROMISE, 2023

Comparison of Various Datasets Based on their Labeling Classifications

Dataset	Only C	Source	#Code Snips.	#Vuln. Snips.	Multi. Vulns/Snip.	Comp./ Gran.	Vuln. Label.	Avg. LOC	Label. Method
Big-Vul	✘	Real-World	188,636	100%	✘	✘/Func.	CVE/CVW	30	PATCH
Draper	✘	Syn.+Real-World	1,274,366	5.62%	✔	✘/Func.	CWE	29	STAT
SARD	✘	Syn.+Real-World	100,883	100%	✘	✔/Prog.	CWE	114	BDV+STAT+MAN
Juliet	✘	Synthetic	106,075	100%	✘	✔/Prog.	CWE	125	BDV
Devign	✘	Real-World	27,544	46.05%	✘	✘/Func.	CVE	112	ML
REVEAL	✘	Real-World	22,734	9.85%	✘	✘/Func.	CVE	32	PATCH
DiverseVul	✘	Real-World	379,241	7.02%	✘	✘/Func.	CWE	44	PATCH
FormAI	✔	AI-gen.	112,000	51.24%	✔	✔/Prog.	CWE	79	ESBMC

Legend:
PATCH: GitHub Commits Patching a Vuln. **Man**: Manual Verification, **Stat**: Static Analyser, **ML**: Machine Learning Based, **BDV**: By design vulnerable

C Keyword Frequency and Associated CWEs



#Vulns	Vuln.	Associated CWE-numbers
88,049	$\mathcal{BOF}$	CWE-20, CWE-120, CWE-121, CWE-125, CWE-129, CWE-131, CWE-628, CWE-676, CWE-680, CWE-754, CWE-787
31,829	$\mathcal{DFN}$	CWE-391, CWE-476, CWE-690
24,702	$\mathcal{DFA}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-755, CWE-787
23,312	$\mathcal{ARO}$	CWE-190, CWE-191, CWE-754, CWE-680, CWE-681, CWE-682
11,088	$\mathcal{ABV}$	CWE-119, CWE-125, CWE-129, CWE-131, CWE-193, CWE-787, CWE-788
9823	$\mathcal{DFI}$	CWE-416, CWE-476, CWE-690, CWE-822, CWE-824, CWE-825
5810	$\mathcal{DFF}$	CWE-401, CWE-404, CWE-459
1620	$\mathcal{OTV}$	CWE-119, CWE-125, CWE-158, CWE-362, CWE-389, CWE-401, CWE-415, CWE-459, CWE-416, CWE-469, CWE-590, CWE-617, CWE-664, CWE-662, CWE-685, CWE-704, CWE-761, CWE-787, CWE-823, CWE-825, CWE-843
1567	$\mathcal{DBZ}$	CWE-369

- $\mathcal{ARO} \subseteq \mathcal{VF}$: Arithmetic overflow
- $\mathcal{BOF} \subseteq \mathcal{VF}$: Buffer overflow on `scanf()/fscanf()`
- $\mathcal{ABV} \subseteq \mathcal{VF}$: Array bounds violated
- $\mathcal{DFN} \subseteq \mathcal{VF}$: Dereference failure : NULL pointer
- $\mathcal{DFF} \subseteq \mathcal{VF}$: Dereference failure : forgotten memory
- $\mathcal{DFI} \subseteq \mathcal{VF}$: Dereference failure : invalid pointer
- $\mathcal{DFA} \subseteq \mathcal{VF}$: Dereference failure : array bounds violated
- $\mathcal{DBZ} \subseteq \mathcal{VF}$: Division by zero
- $\mathcal{OTV} \subseteq \mathcal{VF}$: Other vulnerabilities

The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification

FORMAI DATASET: A LARGE COLLECTION OF AI-GENERATED C PROGRAMS AND THEIR VULNERABILITY CLASSIFICATIONS

Citation Author(s): Norbert Tihanyi (Technology Innovation Institute)
Tamas Bisztray (University of Oslo)
Ridhi Jain (Technology Innovation Institute)
Mohamed Amine Ferrag (Technology Innovation Institute)
Lucas C. Cordeiro (University of Manchester)
Vasileios Mavroeidis (University of Oslo)

Submitted by: Norbert Tihanyi
Last updated: Mon, 06/19/2023 - 15:07
DOI: 10.21227/vp9n-wv96
Data Format: *.csv (zip);

Categories: Artificial Intelligence, Security
Keywords: artificial intelligence, Software Vulnerability Dataset

WARNING: BE CAREFUL WHEN RUNNING THE COMPILED PROGRAMS, SOME CAN CONNECT TO THE WEB, SCAN YOUR LOCAL NETWORK, OR DELETE A RANDOM FILE FROM YOUR FILE SYSTEM. ALWAYS CHECK THE SOURCE CODE AND THE COMMENTS IN THE FILE BEFORE RUNNING IT!!!

https://github.com/FormAI-Dataset

DATASET FILES

- FormAI dataset: Vulnerability Classification (No C source cod included) FormAI_dataset_human_readable-V1.csv (15.95 MB)
- FormAI dataset: 112000 compilable AI-generated C code FormAI_dataset_C_samples-V1.zip (97.61 MB)
- FormAI dataset: Vulnerability Classification (C source code included in CSV) FormAI_dataset_classification-V1.zip (60.66 MB)

SecureFalcon: The Next Cyber Reasoning System for Cyber Security

Mohamed Amine Ferrag*, Ammar Battah*, Norbert Tihanyi*, Merouane Debbah†,
Thierry Lestable*, and Lucas C. Cordeiro‡
*Technology Innovation Institute, 9639 Masdar City, Abu Dhabi, UAE
*Email: firstname.lastname@tii.ae
†Khalifa University of Science and Technology, P O Box 127788, Abu Dhabi, UAE
†Email: merouane.debbah@ku.ac.ae
‡University of Manchester, Manchester, UK
‡Email: lucas.cordeiro@manchester.ac.uk

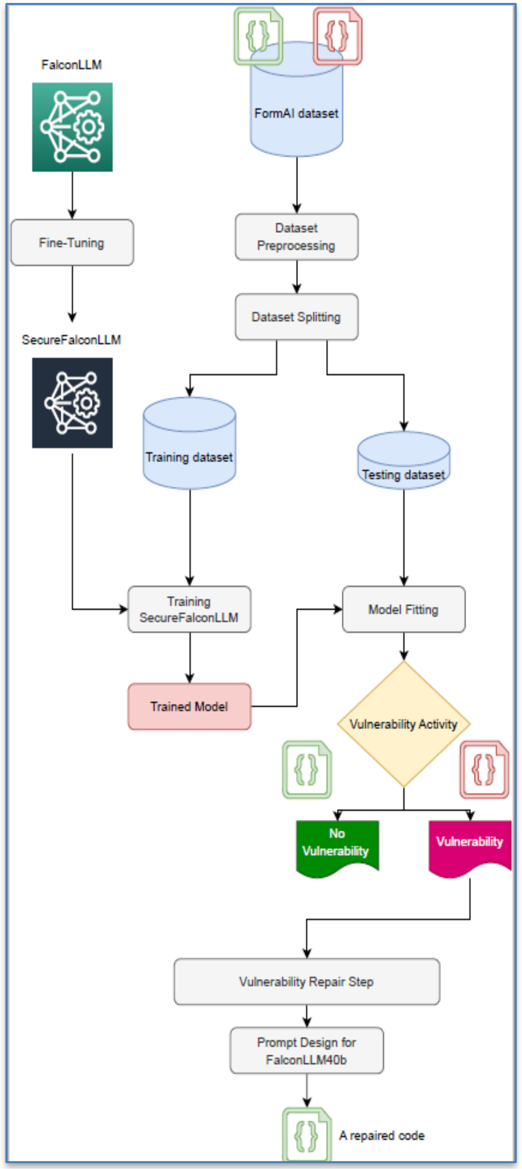


TABLE VII: Classification report of *SecureFalcon* 121M with LR = 2e-5.

	Precision	Recall	F1-Score	Support
0	0.89	0.84	0.86	4528
1	0.95	0.97	0.96	15533
Accuracy		0.94		
Macro avg	0.92	0.90	0.91	20061
Weighted avg	0.94	0.94	0.94	20061

0: NOT VULNERABLE, 1: VULNERABLE

TABLE VIII: Classification report of *SecureFalcon* 121M with LR = 2e-2.

	Precision	Recall	F1-Score	Support
0	0.67	0.80	0.73	4528
1	0.94	0.88	0.91	15533
Accuracy		0.87		
Macro avg	0.80	0.84	0.82	20061
Weighted avg	0.88	0.87	0.87	20061

0: NOT VULNERABLE, 1: VULNERABLE

Vulnrebality detected by *SecureFalcon* model

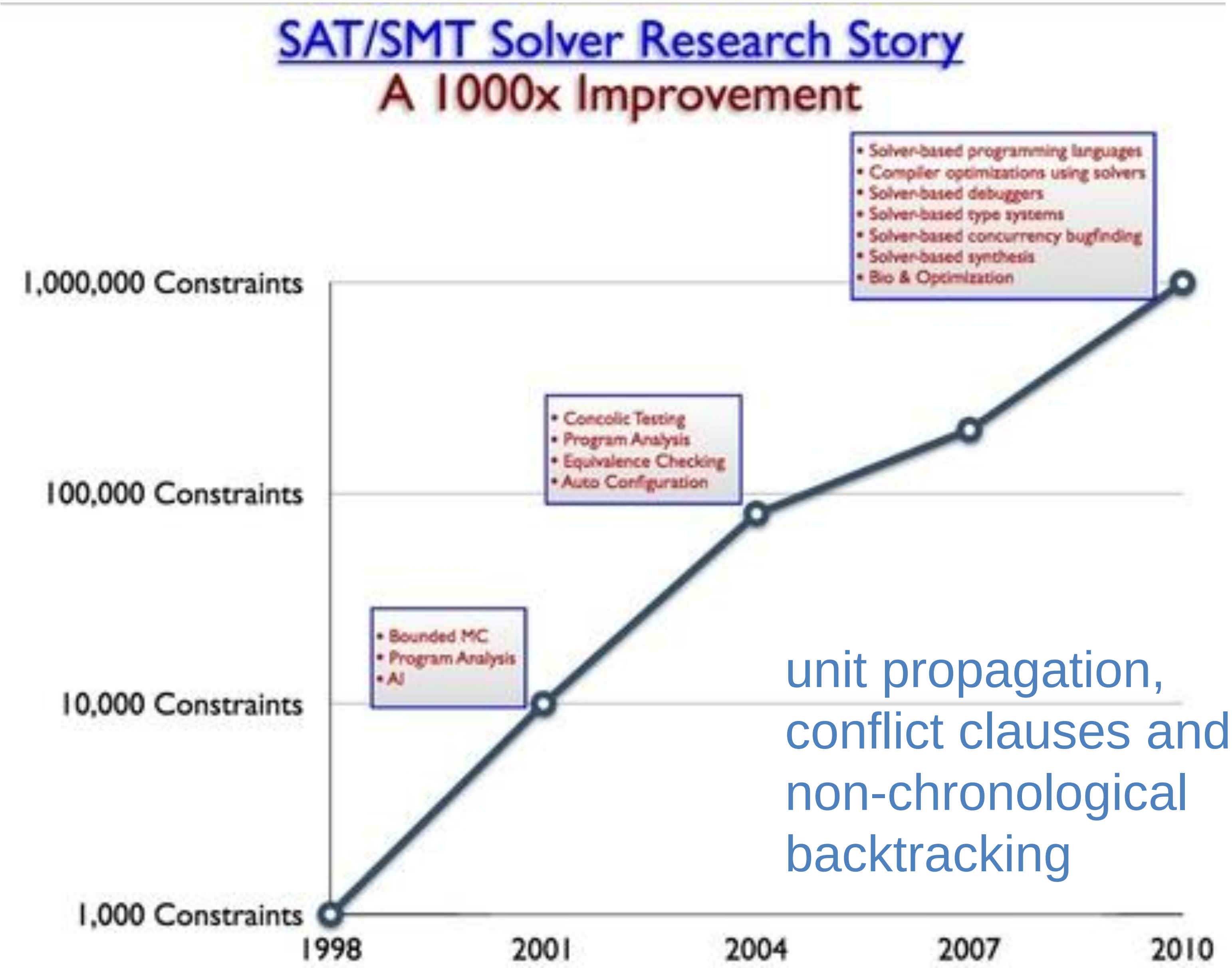
The following C code is vulnerable to a buffer overflow vulnerability. Please repair it.

```c
#include <stdio.h>
#include <string.h>

void secretFunction() {
    printf("Congratulations!\n");
}
void vulnerableFunction(char* str) {
    char buffer[30];
    strcpy(buffer, str);
}
int main(int argc, char** argv) {
    if(argc != 2) {
        printf("Please supply one argument.\n");
        return 1;
    }
    vulnerableFunction(argv[1]);
    printf("Executed normally.\n");
    return 0;
}
```

Agenda

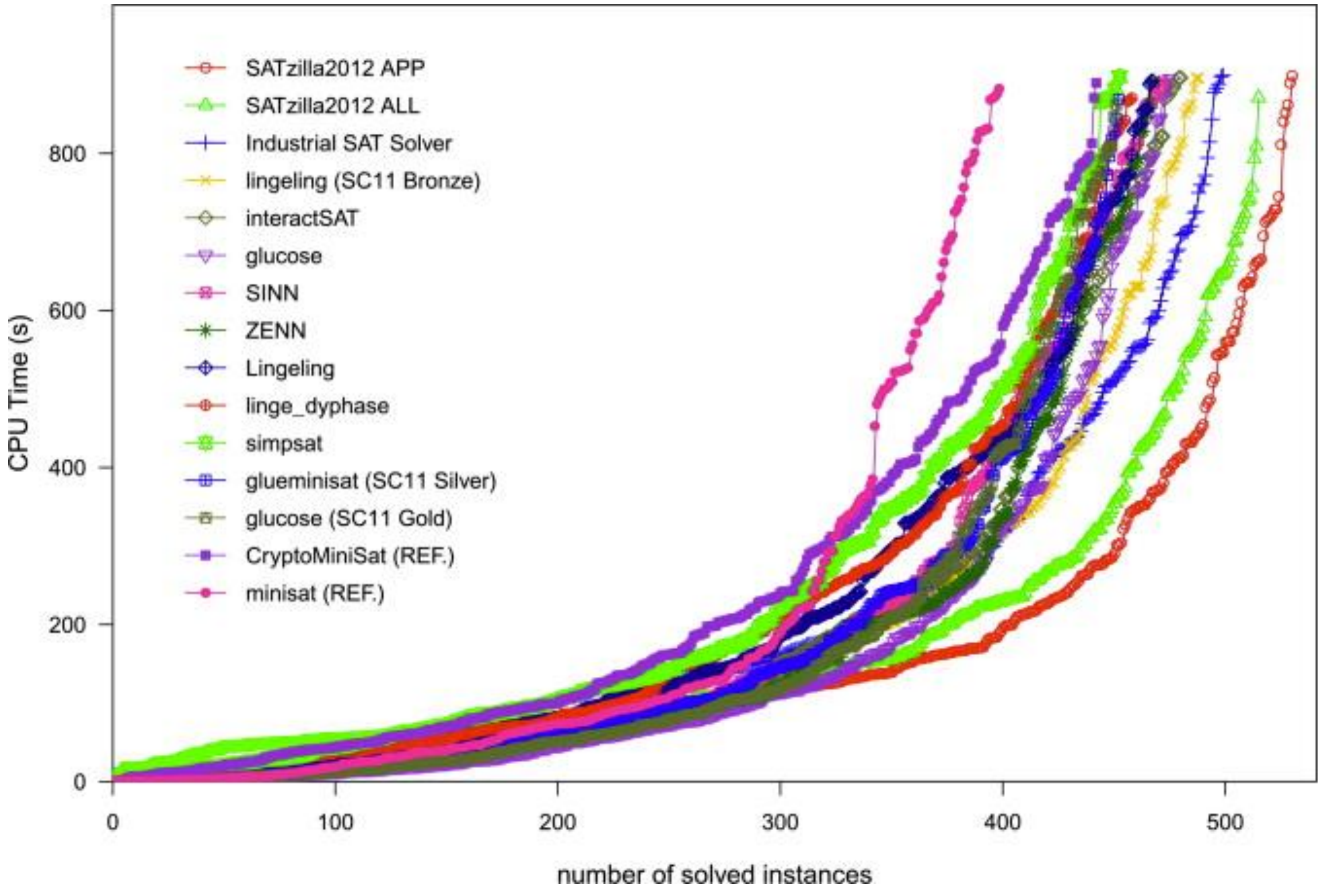
- Towards Self-Healing Software via Large Language Models and Formal Verification
- Software Verification and Testing with the ESBMC framework
- Towards verification of C programs for CHERI platforms with ESBMC

SAT solving as enabling technology



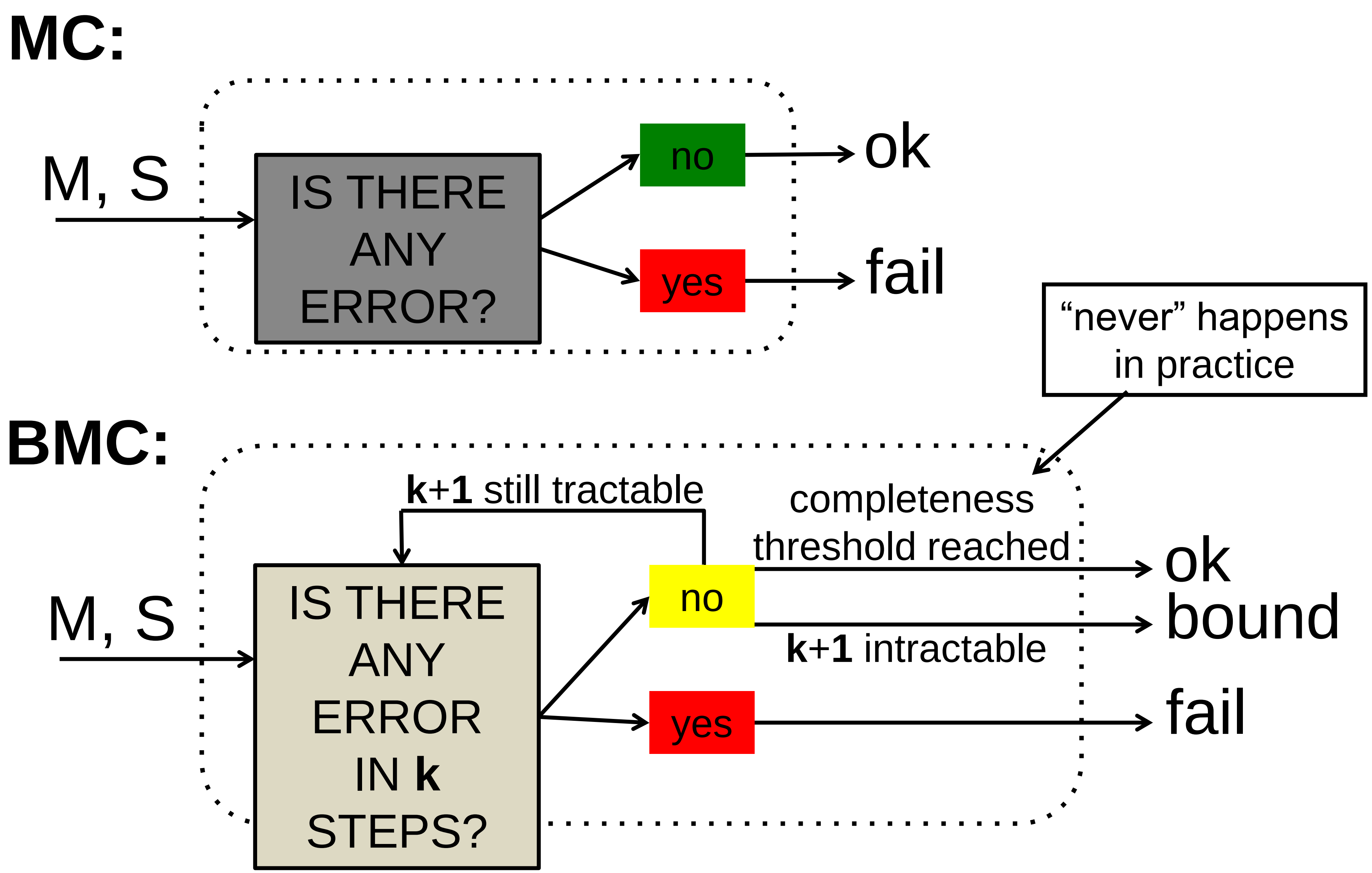
unit propagation, conflict clauses and non-chronological backtracking

SAT Competition



http://www.satcompetition.org/

Bounded Model Checking (BMC)

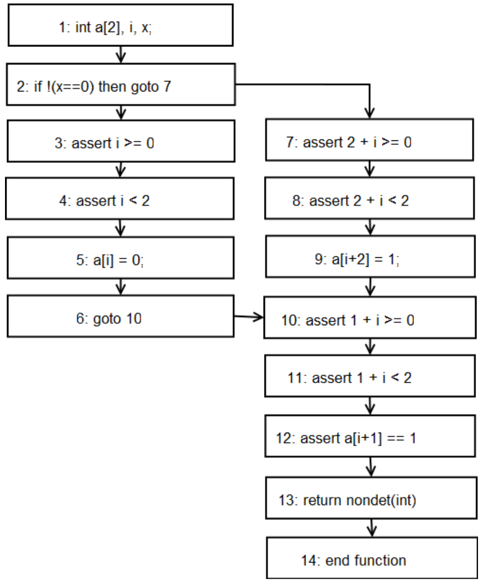


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

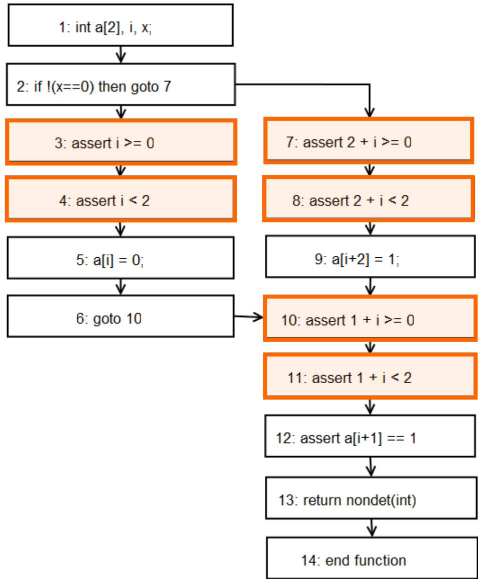


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

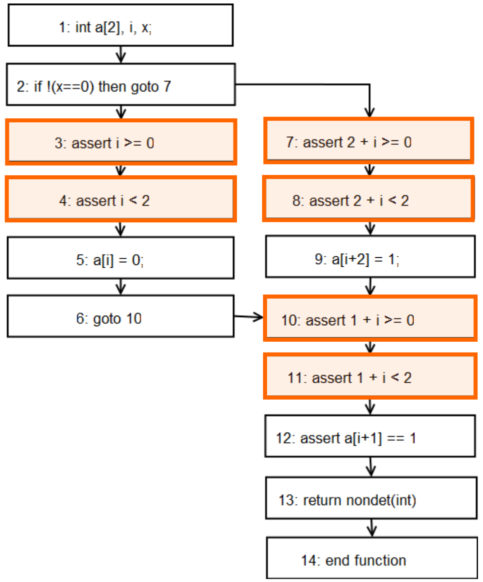


Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added safety properties as extra nodes
- program unfolded up to given bounds

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```



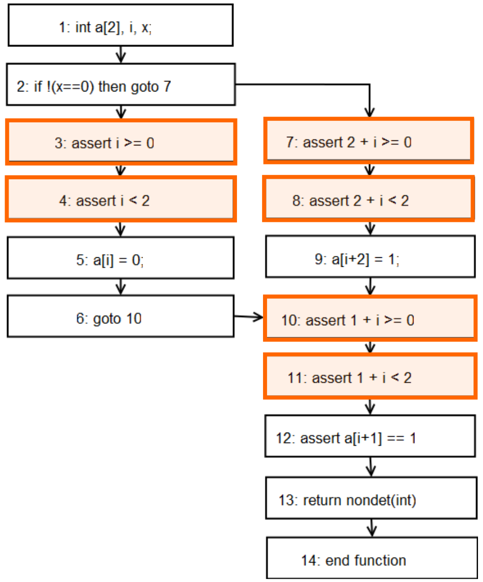
Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added 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

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```



Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added 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 getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

$$
\begin{aligned}
g_1 &= x_1 == 0 \\
a_1 &= a_0 \text{ WITH } [i_0{:=}0] \\
a_2 &= a_0 \\
a_3 &= a_2 \text{ WITH } [2+i_0{:=}1] \\
a_4 &= g_1 \ ?\ a_1 : a_3 \\
t_1 &= a_4[1+i_0] == 1
\end{aligned}
$$

Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added 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*

```c
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```c
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

$$C := \left[\begin{array}{l} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{array}\right]$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{array}\right]$$

Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added 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

```
int getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, ”ML”);
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf(“Access Denied\n”);
    exit(0);
  }
  printf(“Access Granted\n”);
}
```

$$C := \left[\begin{array}{l} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{array}\right]$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{array}\right]$$

Software BMC

- program modelled as transition system
  - *state*: *pc* and program variables
  - derived from control-flow graph
  - added 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 getPassword() {
  char buf[2];
  gets(buf);
  return strcmp(buf, "ML");
}
```

```
void main(){
  int x=getPassword();
  if(x){
    printf("Access Denied\n");
    exit(0);
  }
  printf("Access Granted\n");
}
```

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

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

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)

Induction-Based Verification for Software

***k*-induction** checks loop-free programs...

- **base case ($\boldsymbol{base_k}$):** find a counter-example with up to *k* loop unwindings (plain BMC)
- **forward condition ($\boldsymbol{fwd_k}$):** check that *P* holds in all states reachable within *k* unwindings
- **inductive step ($\boldsymbol{step_k}$):** check that whenever *P* holds for *k* unwindings, it also holds after next unwinding
  - havoc variables
  - assume loop condition
  - run loop body (*k* times)
  - assume loop termination

⇒ iterative deepening if inconclusive

Gadelha, M., Ismail, H., Cordeiro, L.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. 19(1): 97-114 (2017)

Induction-Based Verification for Software

k=1
while k<=$max\_iterations$ **do**
  if $base_{P,\phi,k}$ **then**
    return *trace s[0..k]*
  else
    $k=k+1$
    if $fwd_{P,\phi,k}$ **then**
      return *true*
    else if $step_{P',\phi,k}$ **then**
      return *true*
  end if
end
return *unknown*

```
unsigned int x=*;
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

```
unsigned int x=*;
while(x>0) x--;
assert(x<=0);
assert(x==0);
```

```
unsigned int x=*;
assume(x>0);
while(x>0) x--;
assume(x<=0);
assert(x==0);
```

Automatic Invariant Generation

- Infer invariants based on **intervals** as abstract domain via a dependence graph
  – *E.g., a ≤ x ≤ b (integer and floating-point)*
  – Inject intervals as assumptions and contract them via CSP
  – Remove unreachable states

Line	***Interval for "a"***	***Restriction***
4	$(-\infty, +\infty)$	*None*
6	$(-\infty, 100]$	$a \leq 100$
7	$(100, +\infty)$	$a > 100$

```
1 int main()
2 {
3   int a = *;
4
5   while(a <= 100)
6     a++;
7   assert(a>10);
8   return 0;
9 }
```

k-Induction proof rule "hijacks" loop conditions to nondeterministic values, thus computing intervals become essential

_k_-Induction can prove the correctness of more programs when the invariant generation is enabled

Gadelha, M., Monteiro, F., Cordeiro, L., Nicole, D.: ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution). TACAS (3) 2019: 209-213

BMC of Software Using Interval Methods via Contractors

1) Analyze intervals and properties
   – Static Analysis / Abstract Interpretation
2) Convert the problem into a CSP
   – Variables, Domains and Constraints
3) Apply contractor to CSP
   – Forward-Backward Contractor
4) Apply reduced intervals back to the program

```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

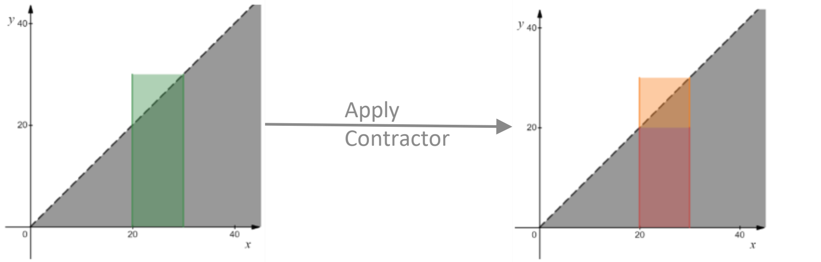
```
__ESBMC_assume(y <= 30 && y >= 20);
```

This **assumption** prunes our search space to the orange area

```
unsigned int x=nondet_uint();
unsigned int y=nondet_uint();
__ESBMC_assume(x >= 20 && x <= 30);
__ESBMC_assume(y <= 30);
assert(x >= y);
```

Domain: $[x] = [20, 30]$ and $[y] = [0, 30]$

Constraint: $y - x \leq 0$

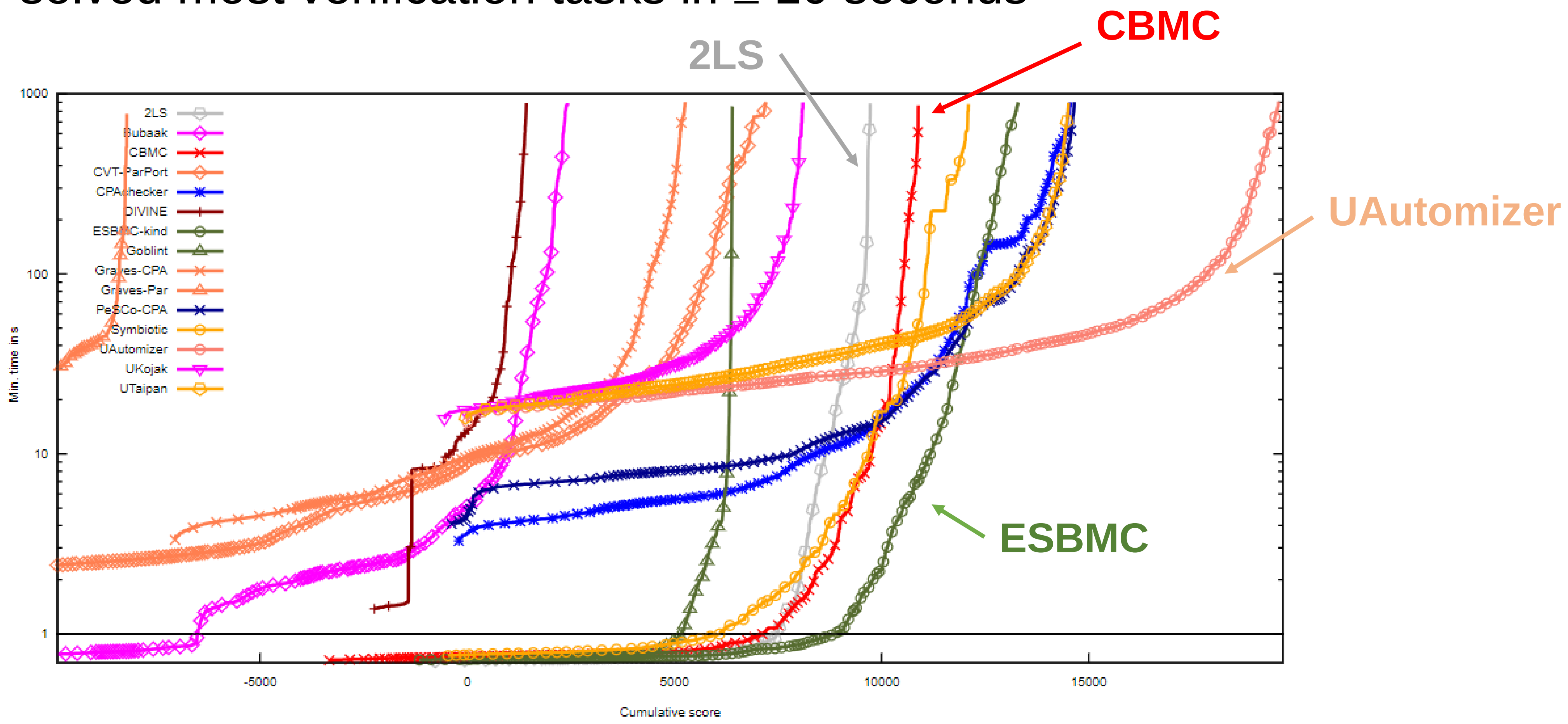


$[x] = [20, 30]$ and $[y] = [0, 30]$ $\quad$ $[x] = [20, 30]$ and $[y] = [20, 30]$

$f(x) > 0$	$I = [0, \infty)$	
$f(x) = y - x$	$[f(x)_1] = I \cap [y_0] - [x_0]$	Forward-step
$x = y - f(x)$	$[x_1] = [x_0] \cap [y_0] - [f(x)_1]$	Backward-step
$y = f(x) + x$	$[y_1] = [y_0] \cap [f(x)_1] + [x_1]$	Backward-step

Intl. Software Verification Competition (SV-Comp 2023)

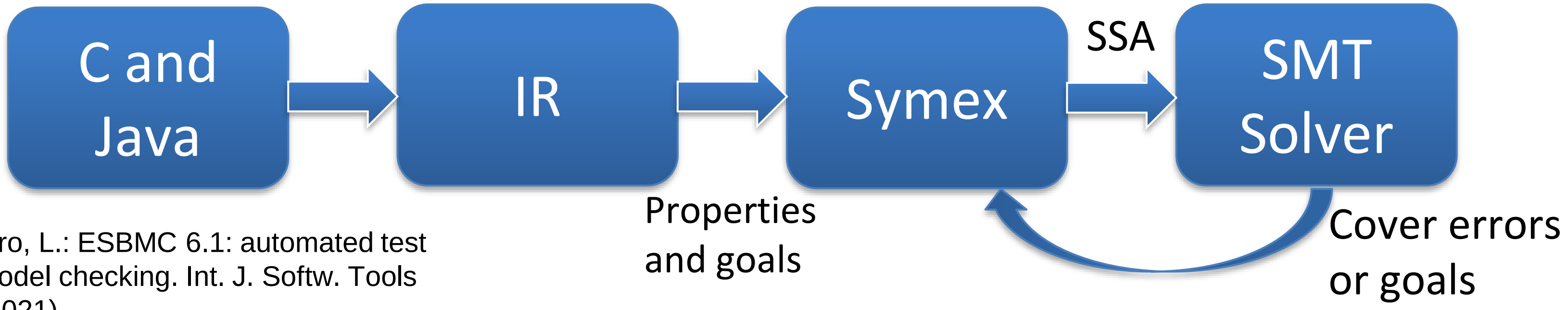
- SV-COMP 2023, 23805 verification tasks, max. score: 38644
- ESBMC solved most verification tasks in ≤ 10 seconds



Verification of the `Overall` Category

White-box Fuzzing:
Bug Finding and Code Coverage

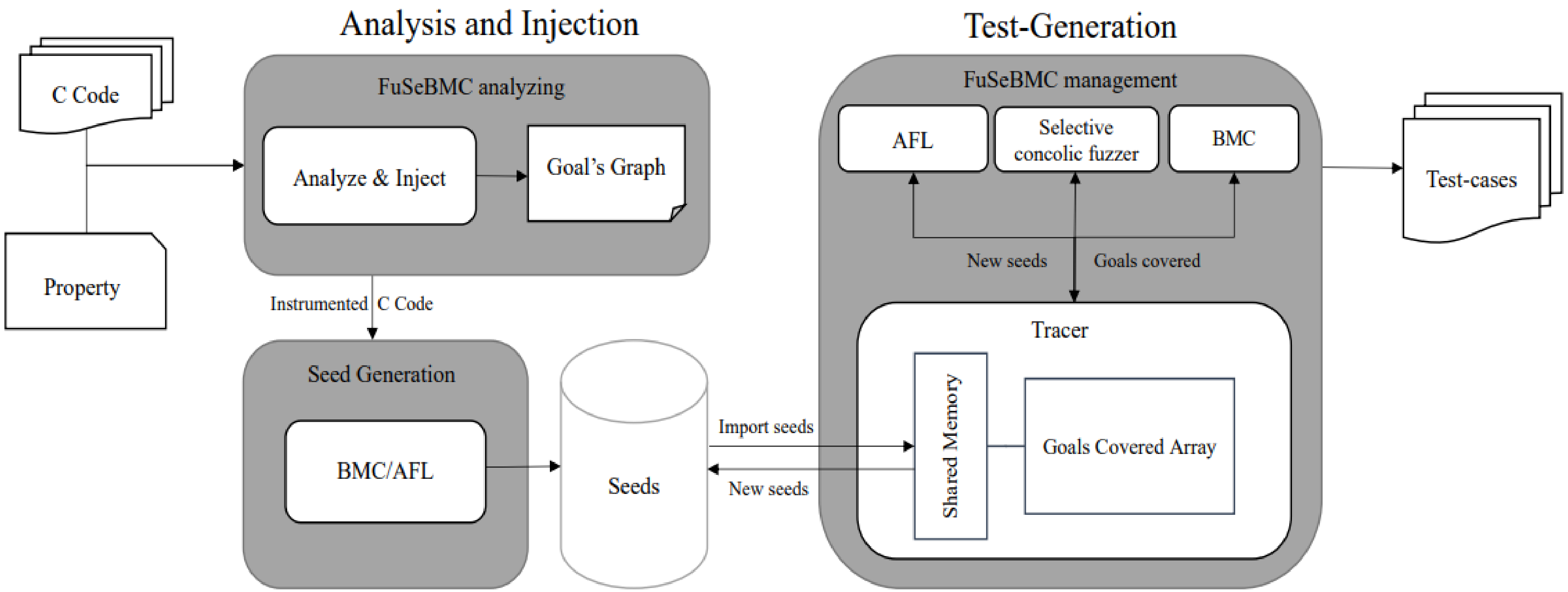
- Translate the program to an **intermediate representation** (IR)
- Add properties to check **errors** or goals to check **coverage**
- **Symbolically** execute IR to produce an SSA program
- Translate the resulting SSA program into a **logical formula**
- Solve the formula iteratively to cover errors and goals
- Interpret the solution to figure out the **input conditions**
- Spit those input conditions out as a test case



Gadelha, M., Menezes, R., Cordeiro, L.: ESBMC 6.1: automated test case generation using bounded model checking. Int. J. Softw. Tools Technol. Transf. 23(6): 857-861 (2021).

FuSeBMC v4 Framework

- Use **Clang** tooling infrastructure
- Employ three engines in its **reachability analysis**: **one BMC and two fuzzing engines**
- Use a **tracer** to coordinate the various engines

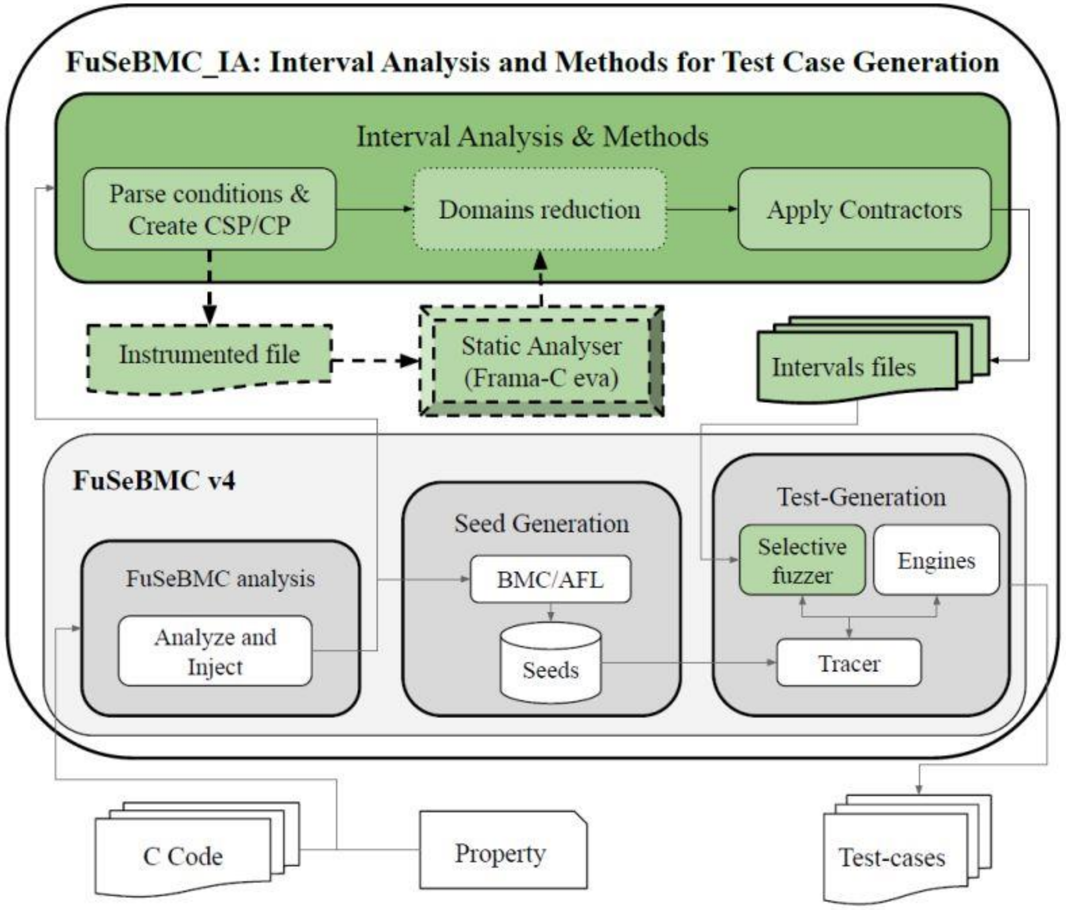


Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

Interval Analysis and Methods for Automated Test Case Generation

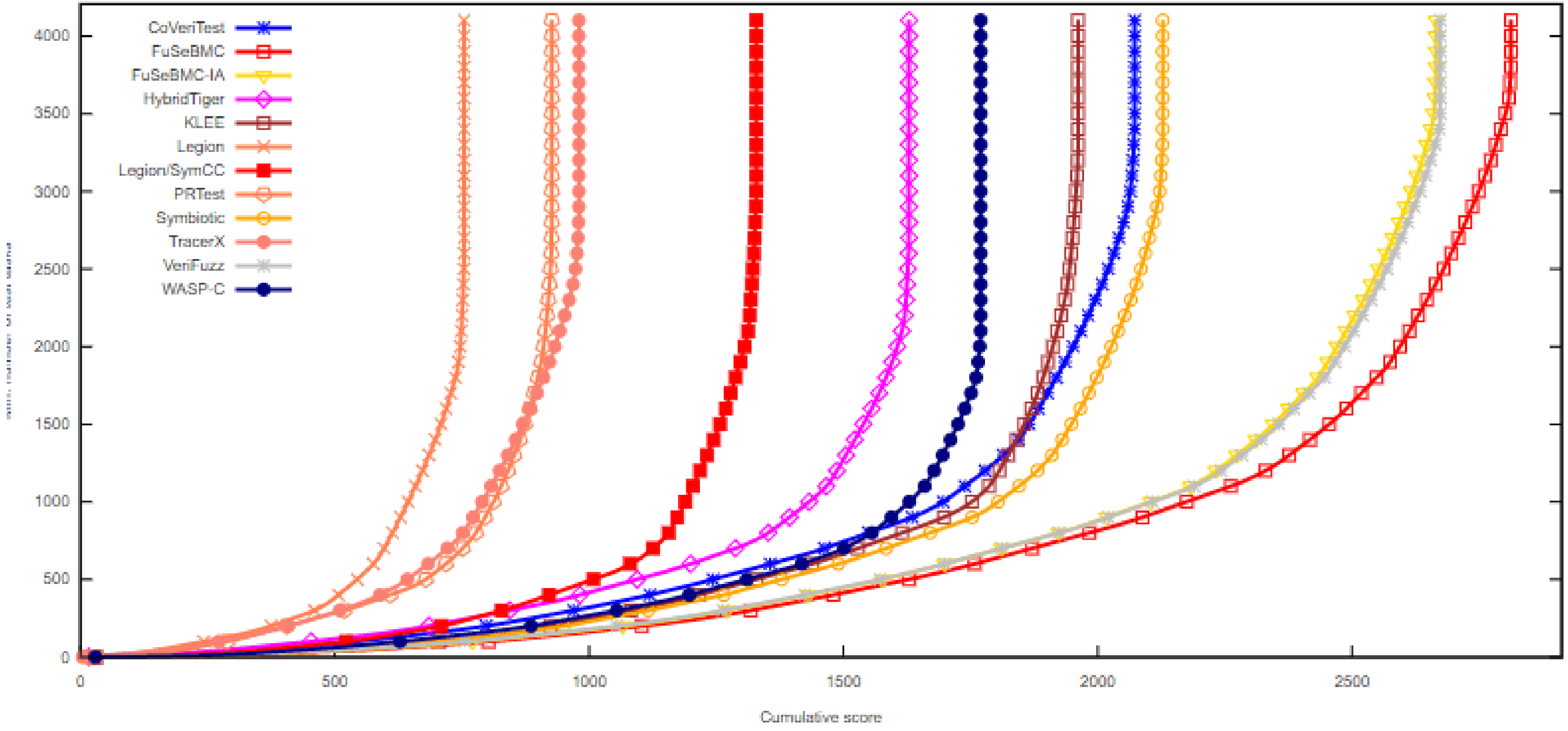
This combined method can **reduce CPU time**, **memory usage**, and **energy consumption**

We advocate that combining **cooperative verification** and **constraint programming** is essential to leverage a **modular cooperative cloud-native testing platform**



Aldughaim, M., Alshmrany, K., Gadelha, M., de Freitas, R., Cordeiro, L.: FuSeBMC_IA: Interval Analysis and Methods for Test Case Generation - (Competition Contribution). FASE 2023: 324-329

Competition on Software Testing 2023: Results of the `Overall` Category

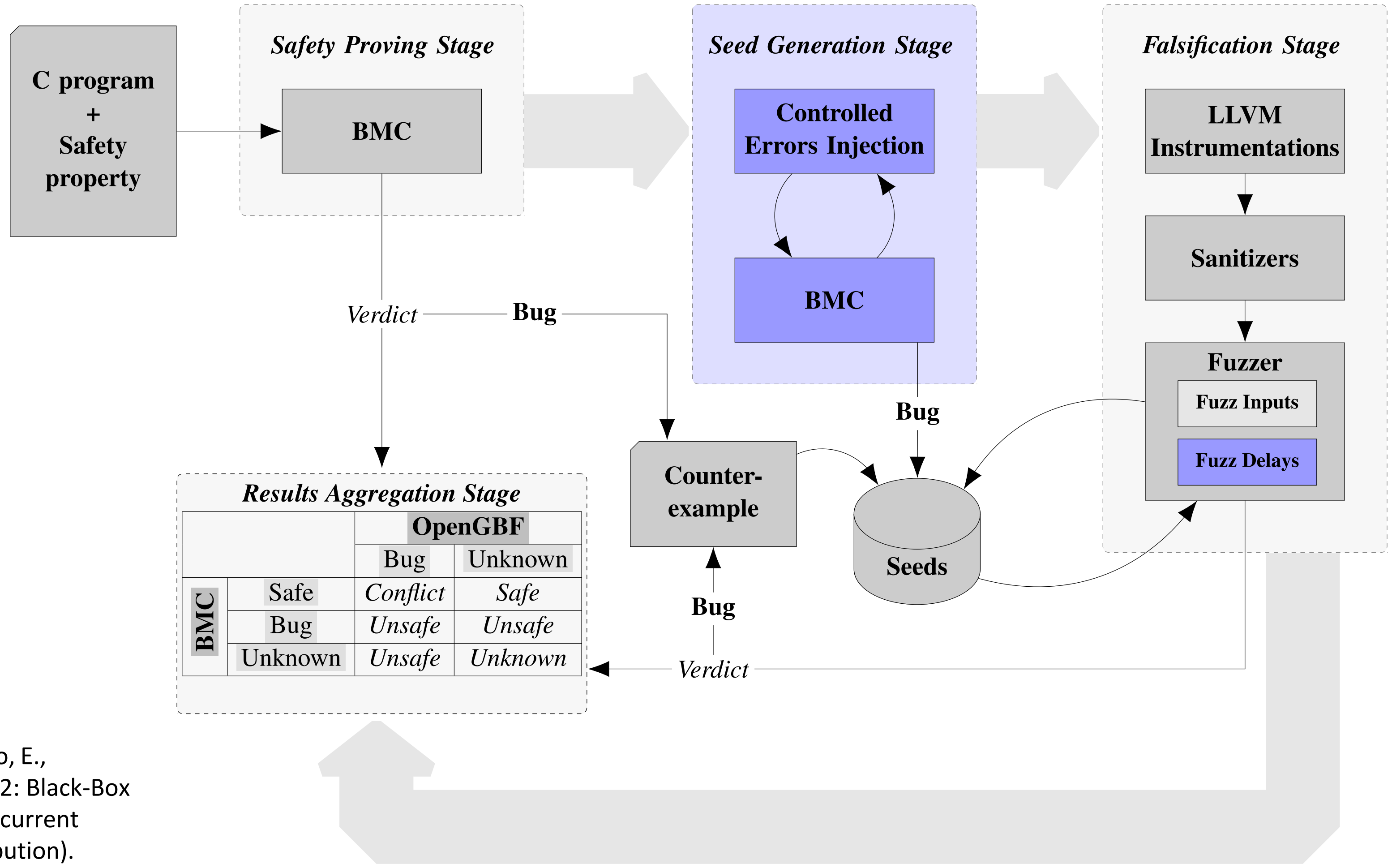


FuSeBMC achieved 3 awards: 1st place in `Cover-Error`, 1st place in `Cover-Branches`, and 1st place in `Overall`

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/2023/

EBF: Black-Box Cooperative Verification for Concurrent Programs



		OpenGBF	
		Bug	Unknown
BMC	Safe	*Conflict*	*Safe*
	Bug	*Unsafe*	*Unsafe*
	Unknown	*Unsafe*	*Unknown*

Aljaafari, F., Shmarov, F., Manino, E., Menezes, R., Cordeiro, L.: EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs - (Competition Contribution). TACAS (2) 2023: 541-546

EBF 4.0 with different BMC tools

- **BMC** 6 min + **OpenGBF** 5 min + **results Aggregation** 4 min = 15 min
- **RAM limit** is 15 GB per Benchexec run
- **ConcurrencySafety main** from SV-COMP 2022
  - Witness validation switched off
- Ubuntu 20.04.4 LTS with 160 GB RAM and 25 cores

Verification outcome	Tool							
	EBF	*Deagle*	*EBF*	*Cseq*	*EBF*	*ESBMC*	*EBF*	*CBMC*
Correct True	240	240	172	**177**	65	**70**	139	**146**
Correct False	**336**	319	**333**	313	**308**	268	**320**	303
Incorrect True	0	0	0	0	0	0	0	0
Incorrect False	0	0	0	0	0	1	0	3
Unknown	**187**	204	**258**	273	**390**	424	**304**	311

- EBF4.0 **increases** the number of **detected bugs for** BMC tools
- EBF4.0 provides a better **trade-off** between **bug finding** and **safety proving** than each BMC engine

WolfMQTT Verification

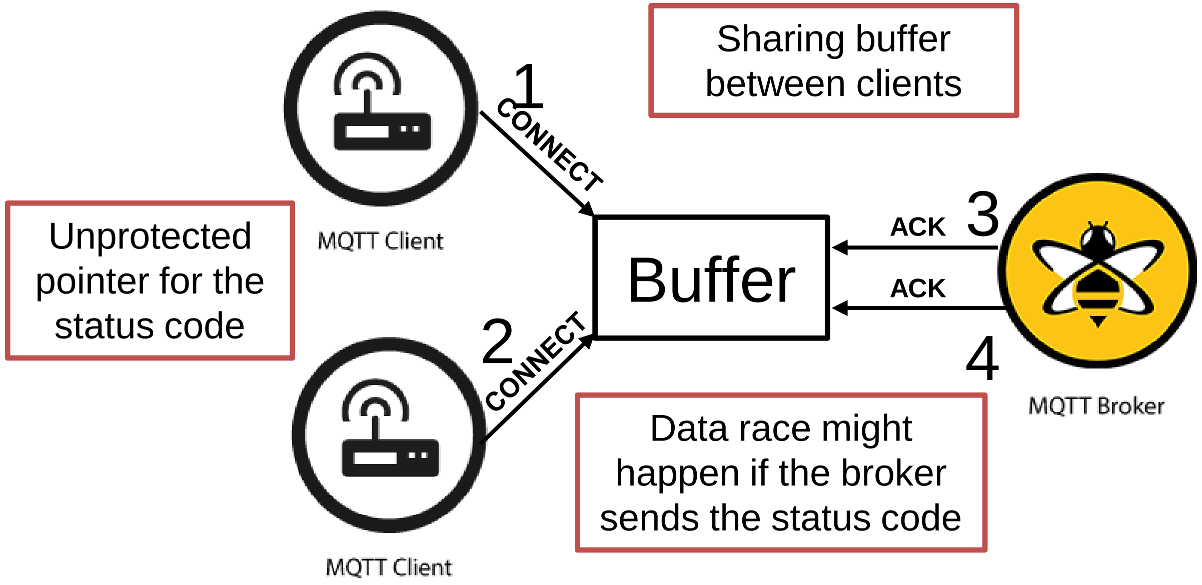
- **wolfMQTT** library is a client implementation of the MQTT protocol written in C for **IoT devices**

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

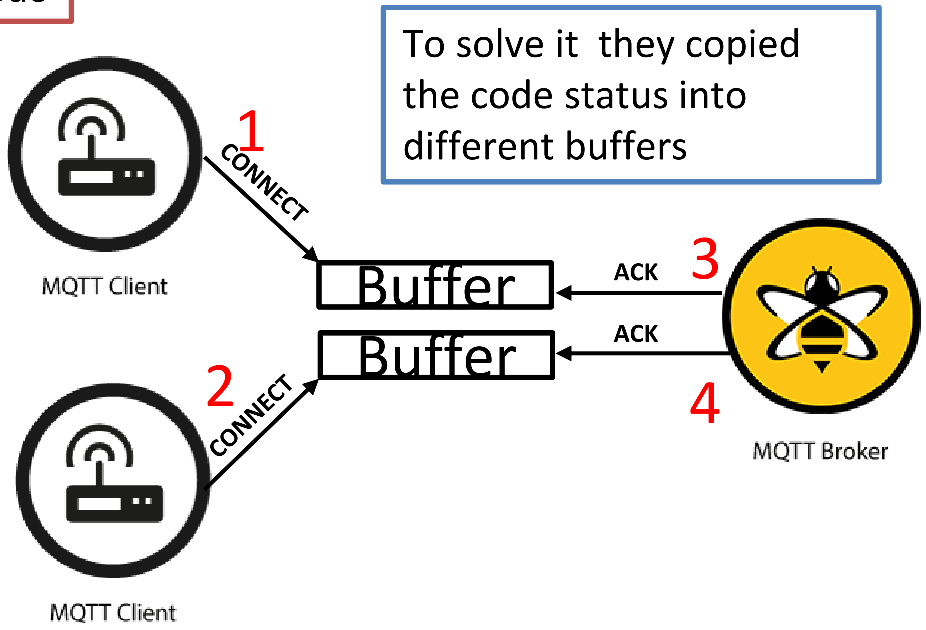
```
Int main(){
Pthread_t th1, th2;
static MQTTCtx mqttCtx;
pthread_create(&th1, subscribe_task, &mqttCtx))
pthread_create(&th2, waitMessage_task, &mqttCtx))}

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;
.....}
```

Here is where the data race might happen! Unprotected pointer

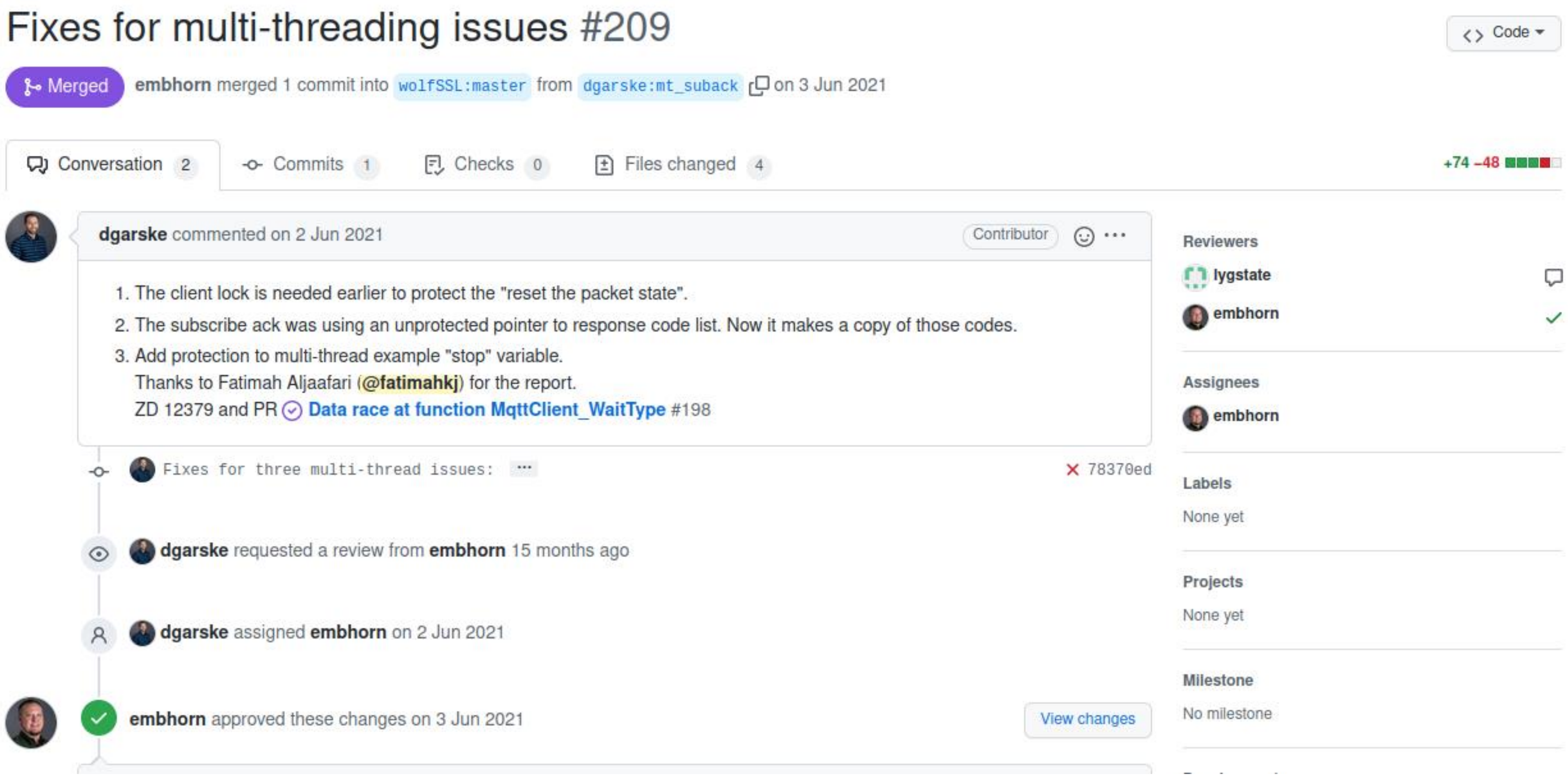
WolfMQTT Verification



After fixing the concurrency vulnerability



Bug Report



https://github.com/wolfSSL/wolfMQTT

wolfSSL
TLS 1.3
wolfSSL

Agenda

- Towards Self-Healing Software via Large Language Models and Formal Verification
- Software Verification and Testing with the ESBMC framework
- Towards verification of C programs for CHERI platforms with ESBMC

Capability Hardware Enhanced RISC Instructions (CHERI)

63		0
permissions (15 bits)	reserved	base and bounds (41 bits)
pointer address (64 bits)		

CHERI 128-bit capability

Mnemonic	Description
CGetBase	Move base to a GPR
CGetLen	Move length to a GPR
CGetTag	Move tag bit to a GPR
CGetPerm	Move permissions to a GPR
CGetPCC	Move the PCC and PC to GPRs
CIncBase	Increase base and decrease length
CSetLen	Set (reduce) length
CClearTag	Invalidate a capability register
CAndPerm	Restrict permissions
CToPtr	Generate **C0**-based integer pointer from a capability
CFromPtr	CIncBase with support for NULL casts
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CLC	Load capability register

CHERI Clang/LLVM and **LLD**[1] - compiler and linker for CHERI ISAs

[1]https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html

CheriBSD[2] - adaptation of FreeBSD to support CHERI ISAs

[2]https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html

ARM Morello[3] - SoC development board with a CHERI-extended ARMv8-A processor

[3]https://www.arm.com/architecture/cpu/morello

CHERI-C program

```c
#include <stdlib.h>
#include <string.h>
#include <cheri/cheric.h>

void main() {
   int n = nondet_uint() % 1024;                                /* models arbitrary user input */
   char a[n+1], *__capability b = cheri_ptr(a, n+1);
   b[n] = 17;                                                   /* succeeds */
   char *__capability c = cheri_setbounds(b-1, n);              /* fails: not the same object */
   /* ... */                                                    /* more CHERI-C API checks */
   memset_c(c, 42, n);                                          /* setting memory through a capability */
}
```

CHERI-C API

New capability types

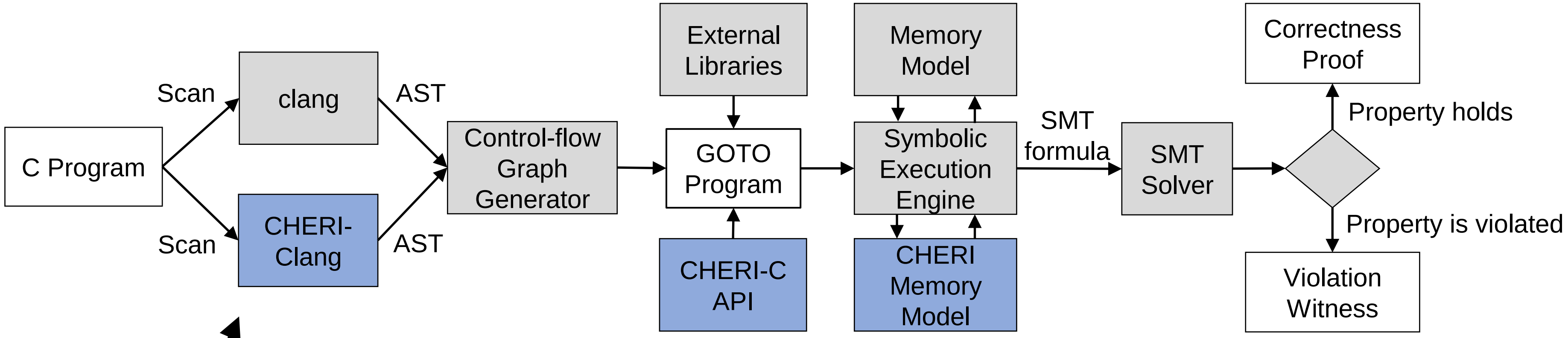
Pure-capability CHERI-C model

```
#include <stdlib.h>
#include <string.h>
#include <cheri/cheric.h>

void main() {
  int n = nondet_uint() % 1024;
  char a[n+1], *__capability b = cheri_ptr(a, n+1);
  b[n] = 17;
  char *__capability c = cheri_setbounds(b-1, n);
  /* ... */
  memset_c(c, 42, n);
}
```

```
#include <string.h>
#include <stdio.h>

void main(void) {
  int n = nondet_uint() % 1024;
  char a[n+1], *b = a;
  b[n] = 17;
  char *c = b-1;
  memset(c, 42, n);
}
```

All pointers are automatically replaced with capabilities by the CHERI Clang/LLVM compiler

ESBMC-CHERI



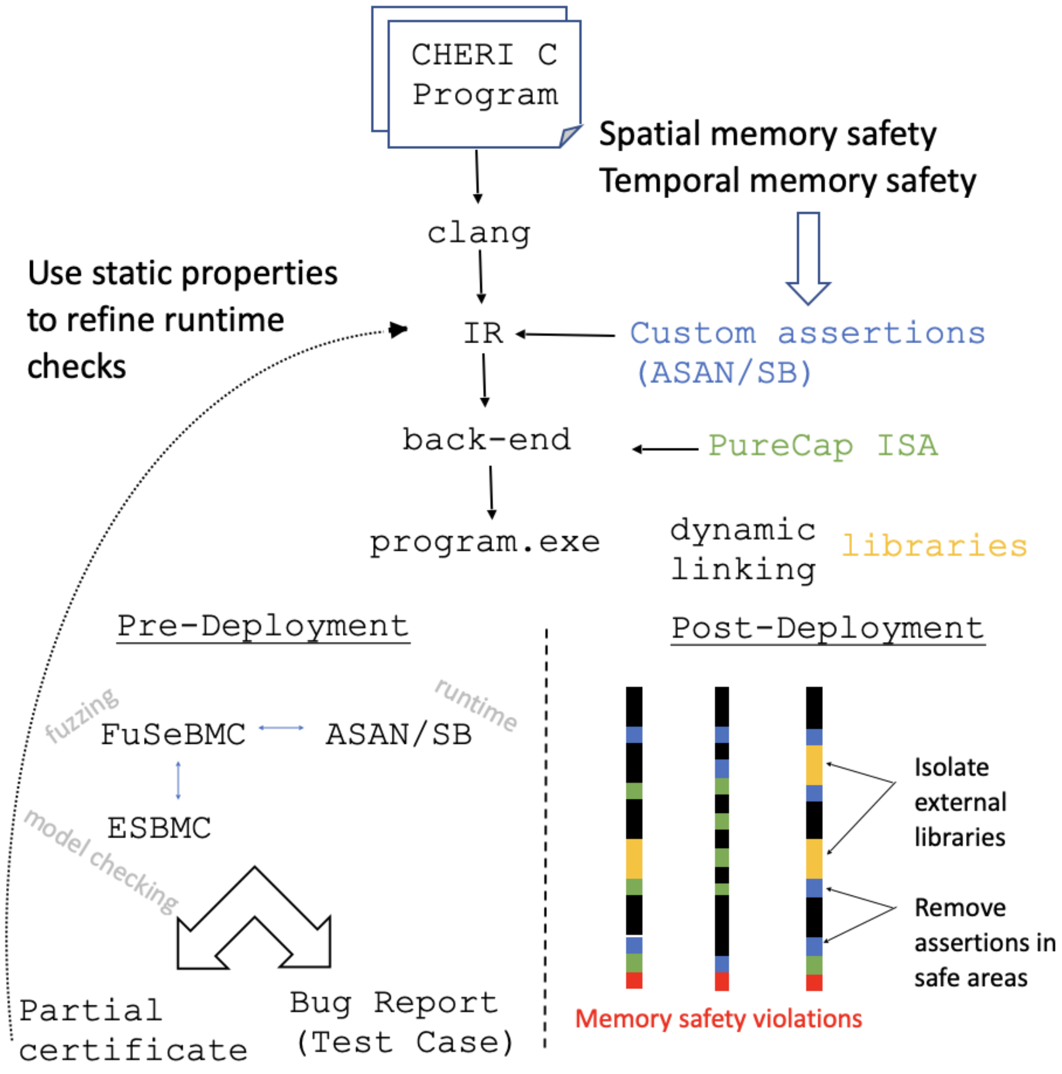
CHERI Clang/LLVM compiler

Implement computational model for CHERI-C API functions inside ESBMC (e.g., ***cheri_setbounds***)

- New capability types
- Tagged memory
- Capability dereferencing

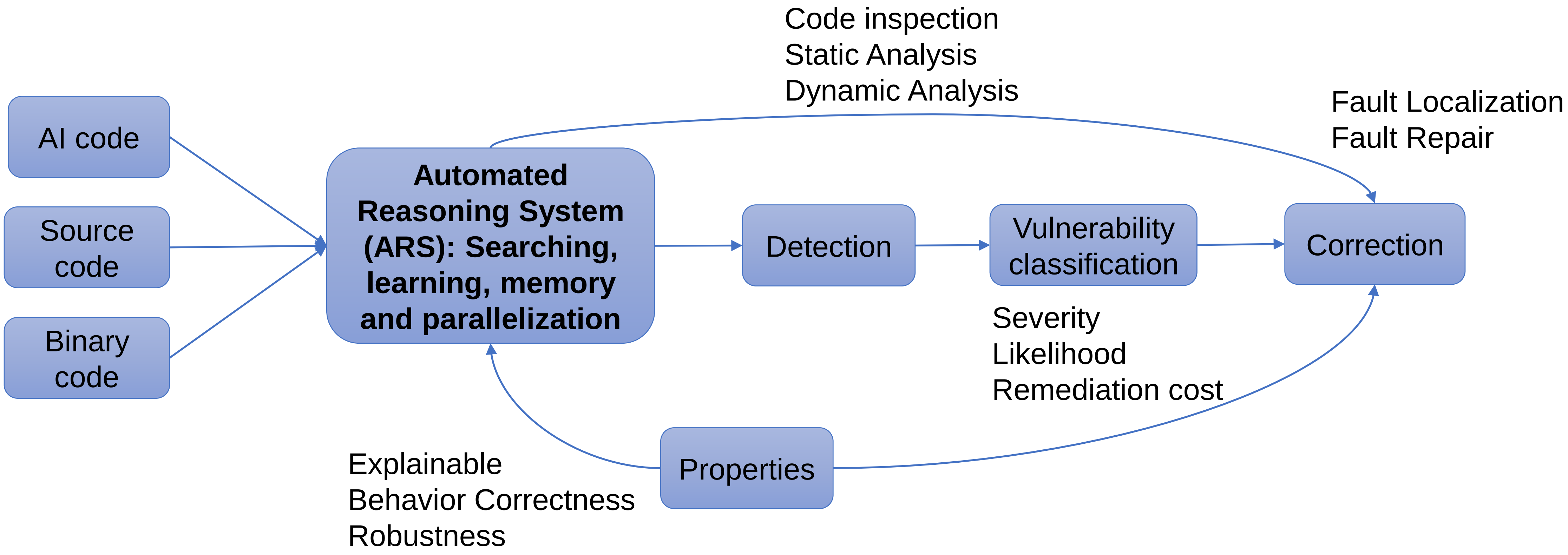
Braußе et al.: ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. ISSTA 2022: 773-776

Hybrid Verification Framework Vision



- Accentuate post-deployment safety
  - Reduce performance overheads by using “cheaper” hardware level protection
  - Reuse the information from static analysis to ensure only necessary more “expensive” safety checks are introduced
- Enhance pre-deployment analysis
  - Combine complementary techniques
  - Avoid producing a monolithic hybrid solution (e.g., concolic execution)

Research Mission: Automated Reasoning System for Safe & Secure SW and AI



Impact: Awards and Industrial Deployment

- **Distinguished Paper Award** at ICSE’11
- **Best Paper Award** at SBESC’15
- **Most Influential Paper Award** at ASE’23
- **39 awards** from the international competitions on software verification (SV-COMP) and testing (Test-Comp) 2012-2023 at **TACAS/FASE**
  - Bug Finding and Code Coverage
- **Intel** deploys **ESBMC** in production as one of its verification engines for **verifying firmware in C**
- **Nokia and ARM** have found **security vulnerabilities** in **software** written in **C/C++**
- **Funded by** EPSRC, Intel, Motorola, Samsung, Nokia, CNPq, FAPEAM, British Council, and Royal Society