# VERIFICATION OPTIMIZATION THROUGH GLOBAL COMMON SUBEXPRESSION ELIMINATION

Menezes, R. S., Tihanyi, N. Jain, R., Levin, A., de Freitas, R. Cordeiro, L.

# OUR TOOL

# OUR TOOL

- ESBMC is a software verifier capable of ensuring safety properties of programs.

# OUR TOOL

- ESBMC is a software verifier capable of ensuring safety properties of programs.
- It is used by industrial partners for the verification of firmware.

# OUR TOOL

- ESBMC is a software verifier capable of ensuring safety properties of programs.
- It is used by industrial partners for the verification of firmware.
- The approach consists in translating a bounded trace of the program into an SMT formula.

# OUR TOOL

- ESBMC is a software verifier capable of ensuring safety properties of programs.
- It is used by industrial partners for the verification of firmware.
- The approach consists in translating a bounded trace of the program into an SMT formula.
- https://github.com/esbmc/esbmc

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

$$(a_0 \geq -128 \wedge a_0 \leq 127) \wedge (b_0 \geq -128 \wedge b_0 \leq 127)$$

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

$$(a_0 \geq -128 \wedge a_0 \leq 127) \wedge (b_0 \geq -128 \wedge b_0 \leq 127)$$
$$\wedge (b_1 = b_0 \times 2)$$

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

$$(a_0 \geq -128 \wedge a_0 \leq 127) \wedge (b_0 \geq -128 \wedge b_0 \leq 127)$$
$$\wedge (b_1 = b_0 \times 2) \wedge [b_2 = \phi(a_0 > 10 \Rightarrow b_1, a_0 \leq 10 \Rightarrow b_0)]$$

# HOW IT WORKS

```
1 int main() {
2   char a,b;
3   if(a > 10)
4     b *= 2;
5   assert(a + b != 42);
6 }
```

$$(a_0 \geq -128 \wedge a_0 \leq 127) \wedge (b_0 \geq -128 \wedge b_0 \leq 127)$$
$$\wedge (b_1 = b_0 \times 2) \wedge [b_2 = \phi(a_0 > 10 \Rightarrow b_1, a_0 \leq 10 \Rightarrow b_0)]$$
$$\wedge (a_0 + b_2 = 42)$$

# WHAT MAKES IT DIFFICULT

# WHAT MAKES IT DIFFICULT

- Tracking pointers.

# WHAT MAKES IT DIFFICULT

- Tracking pointers.
- We compute points-to by evaluating the symbolic expressions and recursively matching addresses.

# WHAT MAKES IT DIFFICULT

- Tracking pointers.
- We compute points-to by evaluating the symbolic expressions and recursively matching addresses.
- This can be even trickier pointers with metadata.

# WHAT MAKES IT DIFFICULT

- Tracking pointers.
- We compute points-to by evaluating the symbolic expressions and recursively matching addresses.
- This can be even trickier pointers with metadata.

```
1 struct my_ptr {
2   int64_t size : 8;
3   int64_t addr : 56;
4 };
```

# WHAT MAKES IT DIFFICULT

```
 1 int foo() {
 2   struct my_ptr ptr;
 3   if(*) {
 4       ptr.addr = malloc(120);
 5       ptr.size = 120;
 6     }
 7   else {
 8     ptr.addr = malloc(150);
 9     ptr.size = 150;
10   }
11   int64_t my_addr = *((int64_t*)&ptr);
12   void *the_addr = (void*) (my_addr & 0x0FFFFFFF);
13   ...
14 }
```

# WHAT MAKES IT DIFFICULT

```
 1 int foo() {
 2   struct my_ptr ptr;
 3   if(*) {
 4       ptr.addr = malloc(120);
 5       ptr.size = 120;
 6     }
 7   else {
 8     ptr.addr = malloc(150);
 9     ptr.size = 150;
10   }
11   int64_t my_addr = *((int64_t*)&ptr);
12   void *the_addr = (void*) (my_addr & 0x0FFFFFFF);
13   ...
14 }
```

# WHAT MAKES IT DIFFICULT

```
 1 int foo() {
 2   struct my_ptr ptr;
 3   if(*) {
 4       ptr.addr = malloc(120);
 5       ptr.size = 120;
 6     }
 7   else {
 8     ptr.addr = malloc(150);
 9     ptr.size = 150;
10   }
11   int64_t my_addr = *((int64_t*)&ptr);
12   void *the_addr = (void*) (my_addr & 0x0FFFFFFF);
13   ...
14 }
```

# WHAT MAKES IT DIFFICULT

```
 1 int foo() {
 2   struct my_ptr ptr;
 3   if(*) {
 4       ptr.addr = malloc(120);
 5       ptr.size = 120;
 6     }
 7   else {
 8     ptr.addr = malloc(150);
 9     ptr.size = 150;
10   }
11   int64_t my_addr = *((int64_t*)&ptr);
12   void *the_addr = (void*) (my_addr & 0x0FFFFFFF);
13   ...
14 }
```

# OPTIMIZING

# OPTIMIZING

- ESBMC relies on multiple checks and analysis to compute pointer aliases.

# OPTIMIZING

- ESBMC relies on multiple checks and analysis to compute pointer aliases.
- We can accelerate it and other operations by applying compiler optimizations.

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   tbl->Map[EntryIndex].Auxiliary.Flags &= 1;
3   tbl->Map[EntryIndex].Wc++;
4   tbl->Map[EntryIndex].V = 42;
5 }
```

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   tbl->Map[EntryIndex].Auxiliary.Flags &= 1;
3   tbl->Map[EntryIndex].Wc++;
4   tbl->Map[EntryIndex].V = 42;
5 }
```

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   tbl->Map[EntryIndex].Auxiliary.Flags &= 1;
3   tbl->Map[EntryIndex].Wc++;
4   tbl->Map[EntryIndex].V = 42;
5 }
```

- The repetitive dereferences makes ESBMC compute the same symbolic pointer operations over and over.

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   tbl->Map[EntryIndex].Auxiliary.Flags &= 1;
3   tbl->Map[EntryIndex].Wc++;
4   tbl->Map[EntryIndex].V = 42;
5 }
```

- The repetitive dereferences makes ESBMC compute the same symbolic pointer operations over and over.
- Invoking the `write` function over 1000 entries results in a 160s symbolic execution and memory exhaustion at solving (after 10min).

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   entry *pEntry = &(tbl->Map[EntryIndex]);
3   pEntry->Auxiliary.Flags &= 1;
4   pEntry->Wc++;
5   pEntry->V = 42;
6 }
```

# EXAMPLE

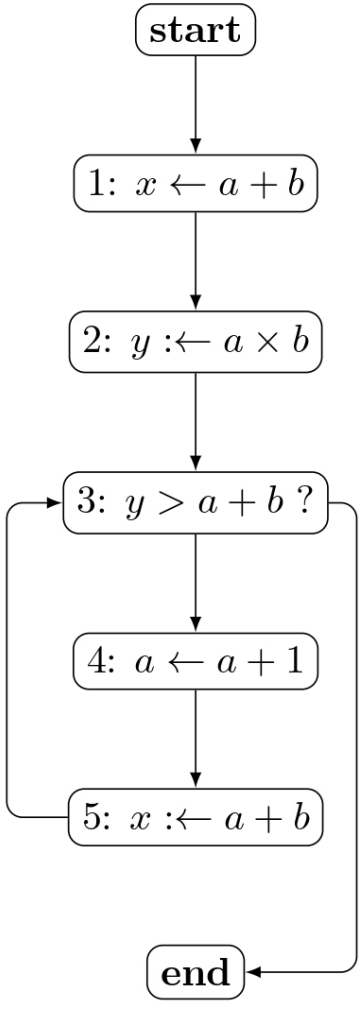
```
1 void write(table *tbl, unsigned EntryIndex) {
2   entry *pEntry = &(tbl->Map[EntryIndex]);
3   pEntry->Auxiliary.Flags &= 1;
4   pEntry->Wc++;
5   pEntry->V = 42;
6 }
```

# EXAMPLE

```
1 void write(table *tbl, unsigned EntryIndex) {
2   entry *pEntry = &(tbl->Map[EntryIndex]);
3   pEntry->Auxiliary.Flags &= 1;
4   pEntry->Wc++;
5   pEntry->V = 42;
6 }
```

- By constructing an alias the analysis now takes less than 1s.

# AVAILABLE EXPRESSIONS




**Domain:** $\{a + b, a \times b, a + 1\}$

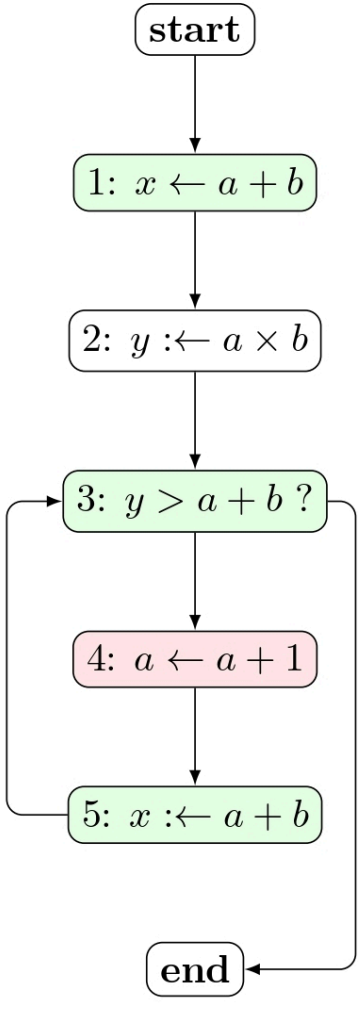
| Vertex | Gen(v) | Kill(v) |
|---|---|---|
| 1 | $\{a + b\}$ | $\emptyset$ |
| 2 | $\{a \times b\}$ | $\emptyset$ |
| 3 | $\{a + b\}$ | $\emptyset$ |
| 4 | $\emptyset$ | $\{a+b, a \times b, a+1\}$ |
| 5 | $\{a + b\}$ | $\emptyset$ |

$$\texttt{In}(v) = \begin{cases} \mathbf{I}, \texttt{v} = \texttt{start} \\ \sqcap_{x \in \texttt{pred}(v)} \texttt{Out}(x) \end{cases}$$

$$\texttt{Out}(v) = \texttt{In}(v) \setminus \texttt{Kill}(v) \cup \texttt{Gen}(v)$$

| Vertex | In(v) | Out(v) |
|---|---|---|
| 1 | $\emptyset$ | $\{a + b\}$ |
| 2 | $\{a + b\}$ | $\{a + b, a \times b\}$ |
| 3 | $\{a + b\}$ | $\{a + b\}$ |
| 4 | $\{a + b\}$ | $\emptyset$ |
| 5 | $\emptyset$ | $\{a + b\}$ |

# AVAILABLE EXPRESSIONS




**Domain:** $\{a + b, a \times b, a + 1\}$

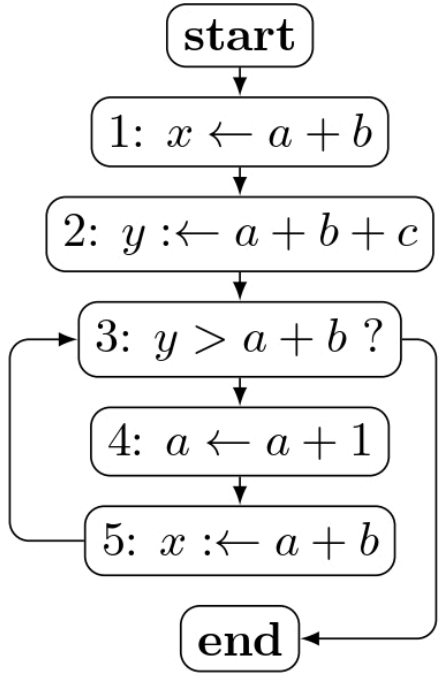
| Vertex | Gen(v) | Kill(v) |
|---|---|---|
| 1 | $\{a + b\}$ | $\emptyset$ |
| 2 | $\{a \times b\}$ | $\emptyset$ |
| 3 | $\{a + b\}$ | $\emptyset$ |
| 4 | $\emptyset$ | $\{a+b, a \times b, a+1\}$ |
| 5 | $\{a + b\}$ | $\emptyset$ |

$$\texttt{In}(v) = \begin{cases} \mathbf{I}, \texttt{v} = \texttt{start} \\ \sqcap_{x \in \texttt{pred}(v)} \texttt{Out}(x) \end{cases}$$

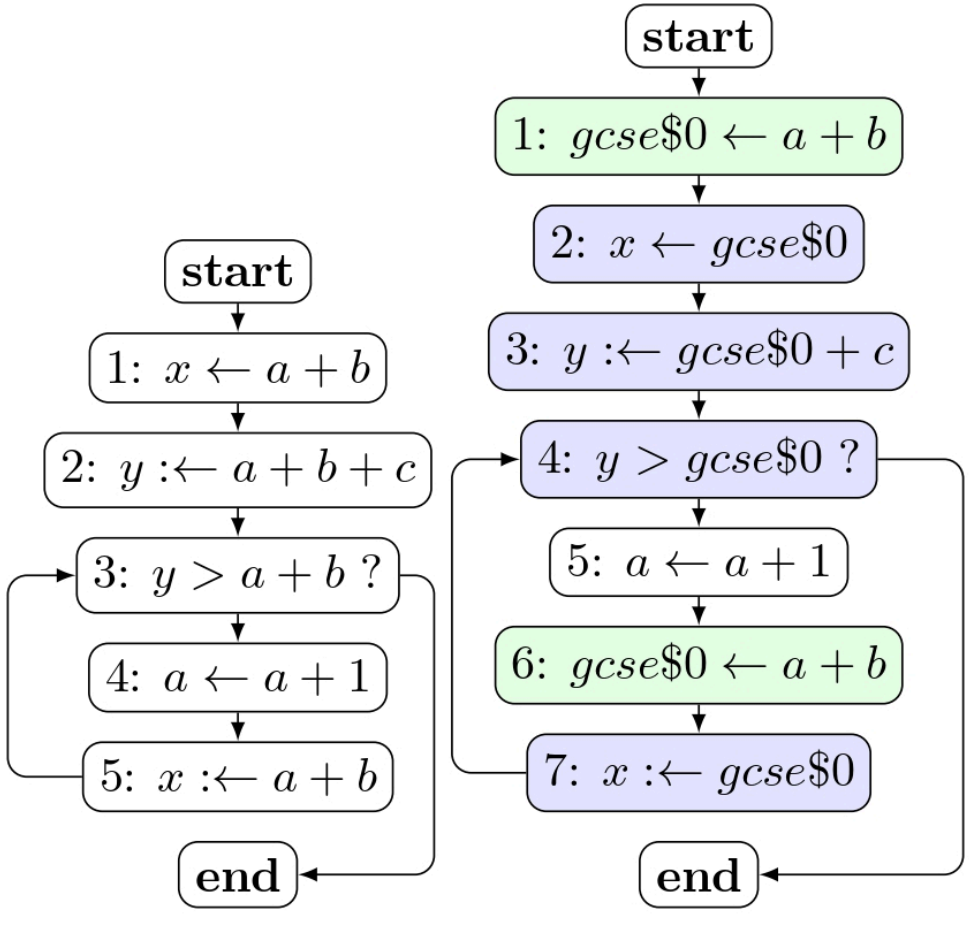
$$\texttt{Out}(v) = \texttt{In}(v) \setminus \texttt{Kill}(v) \cup \texttt{Gen}(v)$$

| Vertex | In(v) | Out(v) |
|---|---|---|
| 1 | $\emptyset$ | $\{a + b\}$ |
| 2 | $\{a + b\}$ | $\{a + b, a \times b\}$ |
| 3 | $\{a + b\}$ | $\{a + b\}$ |
| 4 | $\{a + b\}$ | $\emptyset$ |
| 5 | $\emptyset$ | $\{a + b\}$ |

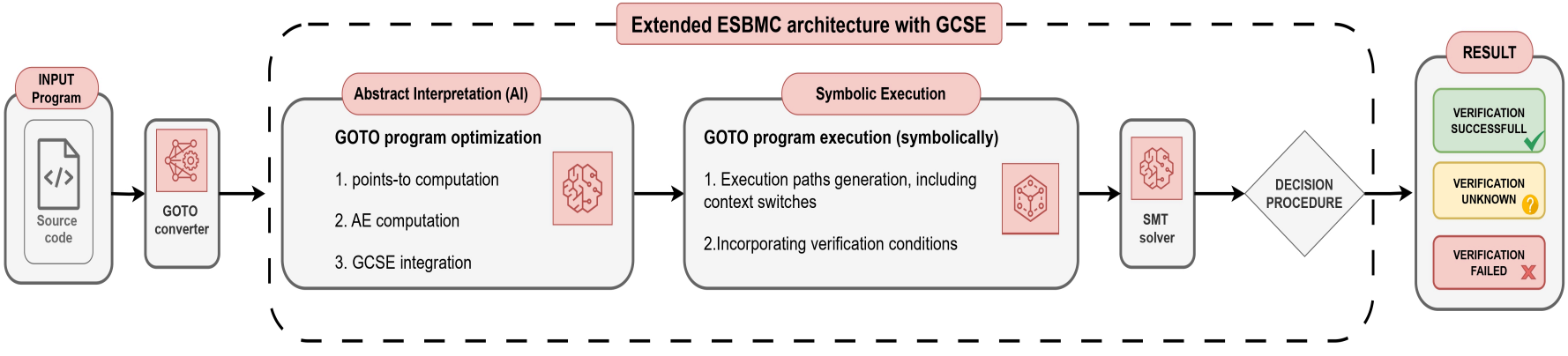
# GCSE


start
1: $x \leftarrow a + b$
2: $y :\leftarrow a + b + c$
3: $y > a + b$ ?
4: $a \leftarrow a + 1$
5: $x :\leftarrow a + b$
end

# GCSE


start
1: $x \leftarrow a + b$
2: $y :\leftarrow a + b + c$
3: $y > a + b$ ?
4: $a \leftarrow a + 1$
5: $x :\leftarrow a + b$
end
start
1: $gcse\$0 \leftarrow a + b$
2: $x \leftarrow gcse\$0$
3: $y :\leftarrow gcse\$0 + c$
4: $y > gcse\$0$ ?
5: $a \leftarrow a + 1$
6: $gcse\$0 \leftarrow a + b$
7: $x :\leftarrow gcse\$0$
end

# VO-GCSE


Extended ESBMC architecture with GCSE
INPUT Program
Source code
GOTO converter
Abstract Interpretation (AI)
GOTO program optimization
1. points-to computation
2. AE computation
3. GCSE integration
Symbolic Execution
GOTO program execution (symbolically)
1. Execution paths generation, including context switches
2.Incorporating verification conditions
SMT solver
DECISION PROCEDURE
RESULT
VERIFICATION SUCCESSFULL
VERIFICATION UNKNOWN
VERIFICATION FAILED

# RESULTS

# RESULTS

- We tested our approach on SV-Comp benchmarks for Reachsafety and Memory categories.

# RESULTS

- We tested our approach on SV-Comp benchmarks for Reachsafety and Memory categories.
- VO-GCSE delivers significant performance improvements in verification time— up to 52%—in memory-related verification tasks.

# RESULTS

- We tested our approach on SV-Comp benchmarks for Reachsafety and Memory categories.
- VO-GCSE delivers significant performance improvements in verification time— up to 52%—in memory-related verification tasks.
- VO-GCSE struggles in categories such as Hardware and ECA. Mainly due to the overhead of the Abstract Interpreter.

# NEXT STEPS

# NEXT STEPS

- We aim to extend ESBMC to be able to use modern compiler techniques, including: phi-nodes, IFDS, data-flow.

# NEXT STEPS

- We aim to extend ESBMC to be able to use modern compiler techniques, including: phi-nodes, IFDS, data-flow.
- Improve the Abstract Interpreter.

# NEXT STEPS

- We aim to extend ESBMC to be able to use modern compiler techniques, including: phi-nodes, IFDS, data-flow.
- Improve the Abstract Interpreter.
- Improve the points-to analysis.

# NEXT STEPS

- We aim to extend ESBMC to be able to use modern compiler techniques, including: phi-nodes, IFDS, data-flow.
- Improve the Abstract Interpreter.
- Improve the points-to analysis.
- Explore other compiler techniques that can be applied to software verification (e.g., loop transformations, strength weakening).

# THANK YOU