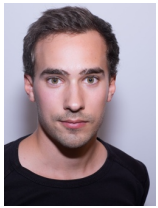
Real-World Software Verification with CPAchecker

Thomas Lemberger

LMU Munich, Germany



Context

- We focus on C programs.
- We use CPAchecker [1]
  `https://gitlab.com/sosy-lab/software/cpachecker/`
- We focus on the basic verification task
- Disclaimer: Not *everything* works well.
  - Basic multi-file verification
  - Not full C standard supported (missing, e.g.: `longjmp`, `atexit`, variadic method arguments).

CPA

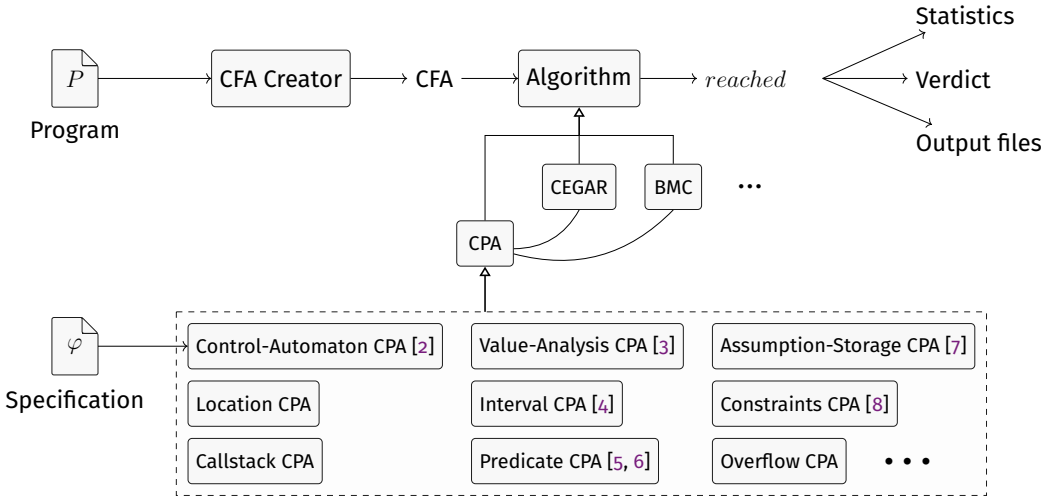
- Framework for Software Verification
  - Mostly written in Java
  - Open Source: Apache 2.0 License
    https://cpachecker.sosy-lab.org
  - 122 contributors
  - 386 346 lines of code
  - 32 531 commits
  - 104 years of effort (COCOMO model)
  - Has a well established, mature codebase maintained by a large development team

Source: https://openhub.net/p/cpachecker on 2023-09-11

CPA✔ : Features

- Input language C (experimental: Java, LLVM)
- Web frontend available:
  `https://vcloud.sosy-lab.org/cpachecker/webclient/run`
- Counterexample output with graphs
- Benchmarking infrastructure available
  (with large cluster of machines)
- Cross-platform: Linux, Mac, Windows

CPA


P
Program
CFA Creator
CFA
Algorithm
$reached$
Statistics
Verdict
Output files
CEGAR
BMC
• • •
CPA
φ
Specification
Control-Automaton CPA [2]
Value-Analysis CPA [3]
Assumption-Storage CPA [7]
Location CPA
Interval CPA [4]
Constraints CPA [8]
Callstack CPA
Predicate CPA [5, 6]
Overflow CPA
• • •

Algorithms

CPA✔ supports many algorithms:

- Configurable Program Analysis [9]
- Bounded Model Checking (BMC) [5]
- Counterexample-guided abstraction refinement (CEGAR) [5, 3]
- k-Induction [10]
- Interpolation-based Model Checking (IMC) [11]

- Test-Case Generation [12, 13]
- Conditional Model Checking [7]
- (Residual) Program Generation and Program Transformations [14, 15]
- ...

CPA✔ : Achievements

- Among world's best software verifiers:
  https://sv-comp.sosy-lab.org/2021/results/
- Continuous success in competition SV-COMP since 2012
  (103 medals: 27x gold, 39x silver, 37x bronze)
- Awarded Gödel medal
  by Kurt Gödel Society

- Used for Linux driver verification
  with dozens of real bugs found and fixed in Linux [16, 17]

CPA for Developers

Want to implement your own analysis?

- Easy, just write a CPA in Java
- CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \text{merge}, \text{stop}, \text{prec})$
- Implementations for 10 interfaces needed
- But for 8, we have default implementations
  $\rightarrow$ Minimal configuration:
  abstract state and
  abstract post operator

CPA✔ for Developers

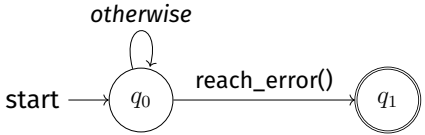
The CPA framework is flexible:

- Many components are provided as CPAs:
  - Location / program counter tracking
  - Callstack tracking
  - Specification input (as automata)
  - Pointer-aliasing information
- CPAs can be combined,
  so your analysis doesn't need to care about these things

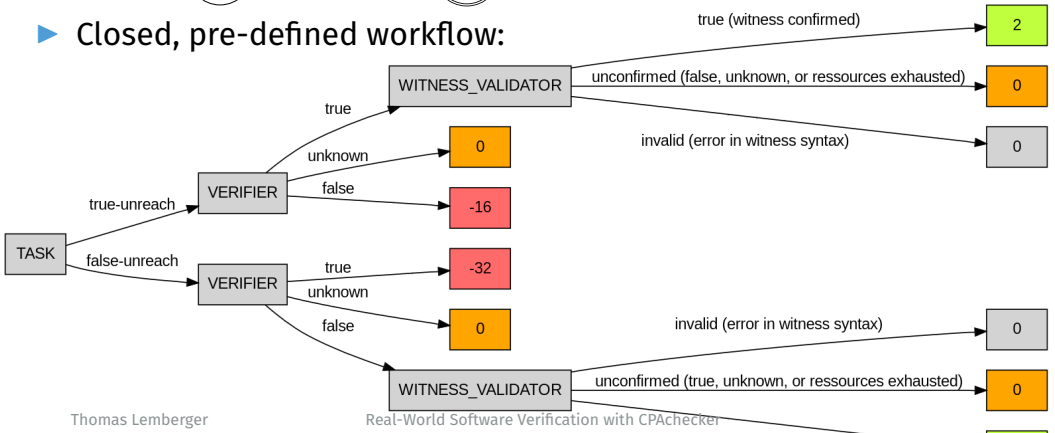
Application

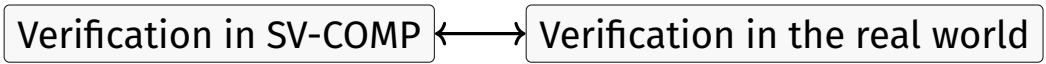
Verification in SV-COMP

- Fixed set of simple properties



- Closed, pre-defined workflow:




Verification in SV-COMP
Verification in the real world

Verification in the Real World

- We have custom properties.
- We want to continue analysis beyond the first result (despite false positives).
- We have to understand the verification result.
- (Which configuration/abstract domain to choose?)

Specs in CPA✔ : Control Automata

- Based on the BLAST query language [18].
- Multiple automata can be composed.

```
CONTROL AUTOMATON OverflowOrAssert

INITIAL STATE Init;
STATE USEFIRST Init:
 MATCH { $1 = sum($2, $3) } -> ASSUME { $1 == $2 + $3; } GOTO Init;
 CHECK("overflow")
  -> ERROR("no-overflow: integer overflow in $location");
 MATCH {__assert_fail($1, $2, $3, $4)}
  -> ERROR("assertion in $location: Condition $1 failed in $2, line $3");

END AUTOMATON
```

Program Properties

- Overflows
- Memory-related properties:

Invalid dereference

```
free(p);
// ...
*p;
```

Double free

```
free(p);
// ...
free(p);
```

⋮

Program Properties

- `assert.h` assertions

  `assert(x > 0);`

  → `if (x > 0) ; else __assert_fail("x > 0", ...);`

- Other types of error handling

```
// From coreutils cksum.c :
unsigned int length;
// ...
if (length + bytes_read < length) {
  errno = EOVERFLOW;
  return false;
}
```

More Complex Program Properties

Only fail if an overflow occurs in an expression that is stored in a variable that is used afterwards.

```
unsigned int bytes_read;
// ...
bytes_read = bytes_read - 1; // may underflow
```

```
STATE USEFIRST Init:
    CHECK(IS_TARGET_STATE) -> ERROR;
    MATCH { $1 = $? } && CHECK("overflow")
        -> MODIFY(TaintAnalysis, "taint($1)") GOTO Init;
```

More Complex Program Properties

Only fail if an overflow occurs in an expression that is stored in a variable that is used afterwards.

```
unsigned int bytes_read;
// ...
bytes_read = bytes_read - 1; // may underflow
```

```
STATE USEFIRST Init:
    CHECK(IS_TARGET_STATE) -> ERROR;
    MATCH { $1 = $? } && CHECK("overflow")
        -> MODIFY(TaintAnalysis, "taint($1)") GOTO Init;
```

- **To-Do**: Taint analysis

Ignore false positives

```
EVAL(location, "lineno") != 10044 && CHECK("overflow")
  -> ERROR("no-overflow: integer overflow in $location");
```

- Issue: Source-code changes

Ignore false positives

```
EVAL(location, "lineno") != 10044 && CHECK("overflow")
  -> ERROR("no-overflow: integer overflow in $location");
```

- Issue: Source-code changes
- **To-Do**: Store false positives in a robust way

What did the verifier do?

- The promised land: verification-result witnesses [2]
- Violation: abstract error path
- Proof: invariants
- Issue: Often stripped of information that verifier computed

```
main N12:
(assert (let ((.def_87 (= |main::x| (_ bv0 32)))).def_87))
```

But correctness witness without any invariant.

⇒ We turn to verifier-specific files

CPA✔ -specific files

```
abstractions.txt                  Counterexample.1.core.txt  functionCallsUsed.dot
ARG.dot                           Counterexample.1.dot       invariantPrecs.txt
ARGRefinements.dot                Counterexample.1.html      invariants.txt
ARGSimplified.dot                 Counterexample.1.smt2      predmap.txt
ARG.svg                           Counterexample.1.spc       Statistics.txt
cfa                               Counterexample.1.txt       UsedConfiguration.propert
cfa.dot                           Counterexample.1.witness.dot VariableClassification.
cfaPixel.svg                      coverage.info              VariableTypeMapping.txt
Counterexample.1.assignment.txt CPALog.txt                   witness.graphml
Counterexample.1.c                functionCalls.dot
```

CPA -specific files: Statistics.txt

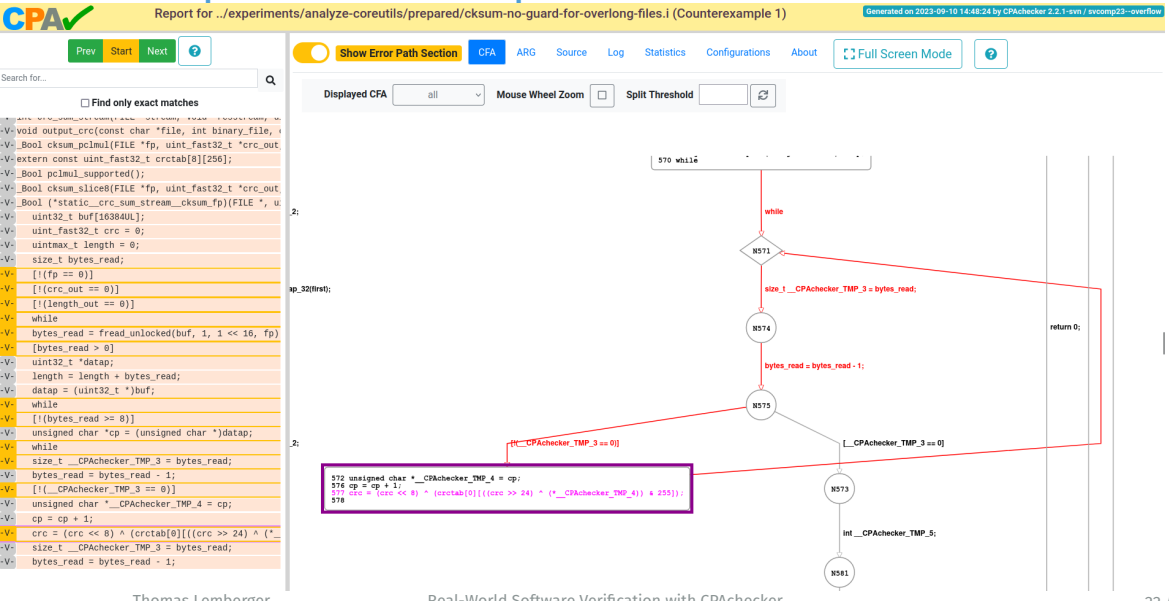
```
...
Time for analysis setup:       1.892s
  Time for loading CPAs:       0.437s
  Time for loading parser:     0.187s
  Time for CFA construction:   1.235s
    Time for parsing file(s):  0.439s
    Time for AST to CFA:       0.490s
    Time for CFA sanity check: 0.037s
    Time for post-processing:  0.201s
    Time for CFA export:       0.697s
    ...
```

CPA✔ -specific files: Statistics.txt

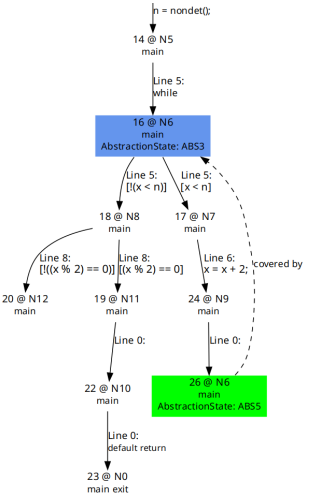
```
...
Time for analysis setup:       1.892s
  Time for loading CPAs:       0.437s
  Time for loading parser:     0.187s
  Time for CFA construction:   1.235s
    Time for parsing file(s):  0.439s
    Time for AST to CFA:       0.490s
    Time for CFA sanity check: 0.037s
    Time for post-processing:  0.201s
    Time for CFA export:       0.697s
    ...
```

- **To-Do**: Drill-down

CPA✔ -specific files: Report.html



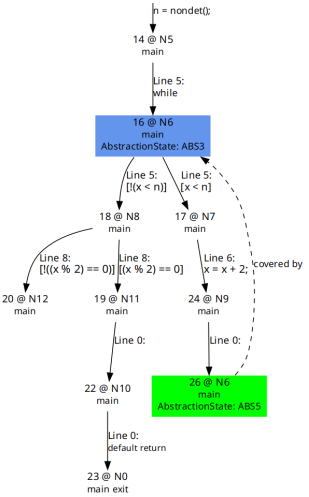
abstractions.txt




```
0 (3) @1:
(assert true)

3 (5) @6:
(assert (let ((.def_83 (bvurem |main::x| (_ bv2
    32))))(let ((.def_86 (bvult |main::x| .def
    _83)))(let ((.def_105 (not .def_86)))(let ((.
    def_91 (= .def_83 (_ bv0 32))))(let ((.def_88
     (bvurem .def_83 (_ bv2 32))))(let ((.def_89
    (= .def_83 .def_88)))(let ((.def_104 (and .
    def_89 .def_91)))(let ((.def_106 (and .def
    _104 .def_105)))(let ((.def_84 (bvult .def_83
     (_ bv2 32))))(let ((.def_107 (and .def_84 .
    def_106))).def_107))))))))))
```

abstractions.txt




```
0 (3) @1:
(assert true)

3 (5) @6:
(assert (let ((.def_83 (bvurem |main::x| (_ bv2
    32))))(let ((.def_86 (bvult |main::x| .def
    _83)))(let ((.def_105 (not .def_86)))(let ((.
    def_91 (= .def_83 (_ bv0 32))))(let ((.def_88
     (bvurem .def_83 (_ bv2 32))))(let ((.def_89
    (= .def_83 .def_88)))(let ((.def_104 (and .
    def_89 .def_91)))(let ((.def_106 (and .def
    _104 .def_105)))(let ((.def_84 (bvult .def_83
     (_ bv2 32))))(let ((.def_107 (and .def_84 .
    def_106))).def_107))))))))))
```

- **To-Do**: Readable SMTlib

Counterexample.1.assignment.txt

- Model of the counterexample

```
int main(void) {
    unsigned int x = 0;
    unsigned short n = nondet();
    while (x < n) {
        x += 2;
    }
    if (x > 5) reach_error();
}
```

```
main::n@3: 6
main::x@2: 0
main::x@3: 2
main::x@4: 4
main::x@5: 6
nondet!2@: 6
```

Conclusion

- **CPA✔** provides a nice framework (specification + analyses) to implement real-world analyses
- Challenges shift from analysis to specification and verification artifacts
- **Call for collaboration**: Please approach me about ideas and projects!

References I

[1] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). doi:10.1007/978-3-642-22110-1_16

[2] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). doi:10.1145/3477579

[3] Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). doi:10.1007/978-3-642-37057-1_11

[4] Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE (2023)

References II

[5] Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). doi:10.1007/s10817-017-9432-6

[6] Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010), https://www.sosy-lab.org/research/pub/2010-FMCAD.Predicate_Abstraction_with_Adjustable-Block_Encoding.pdf

[7] Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). doi:10.1145/2393596.2393664

References III

[8] Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Proc. ISoLA. pp. 195–211. LNCS 9952, Springer (2016). doi:10.1007/978-3-319-47166-2_14

[9] Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). doi:10.1007/978-3-540-73368-3_51

[10] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). doi:10.1007/978-3-319-21690-4_42

References IV

[11] Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). doi:10.48550/arXiv.2208.05046

[12] Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). doi:10.1007/s10009-020-00587-8

[13] Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). doi:10.1007/978-3-319-92994-1_1

[14] Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). doi:10.1145/3180155.3180259

References V

[15] Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: Cegar-pt: A tool for abstraction by program transformation. In: Proc. ASE (2023)

[16] Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). doi:10.1007/978-3-642-11486-1_14

[17] Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA. pp. 1–6. LNCS 7610, Springer (2012). doi:10.1007/978-3-642-34032-1_1

[18] Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The blast query language for software verification. In: Proc. SAS. pp. 2–18. LNCS 3148, Springer (2004). doi:10.1007/978-3-540-27864-1_2