Compilation, Code Transformations, and Executable Counterexamples for DIVINE

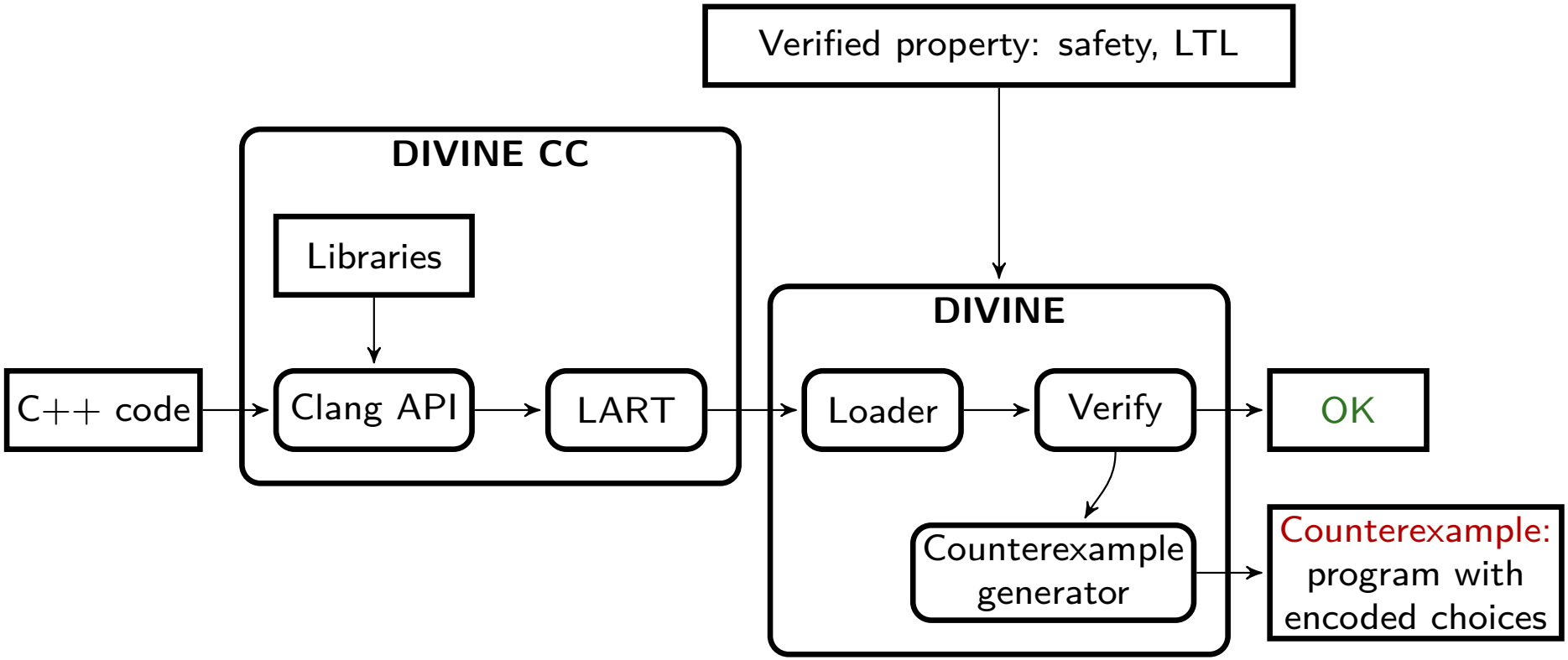
Vladimír Štill

Masaryk University

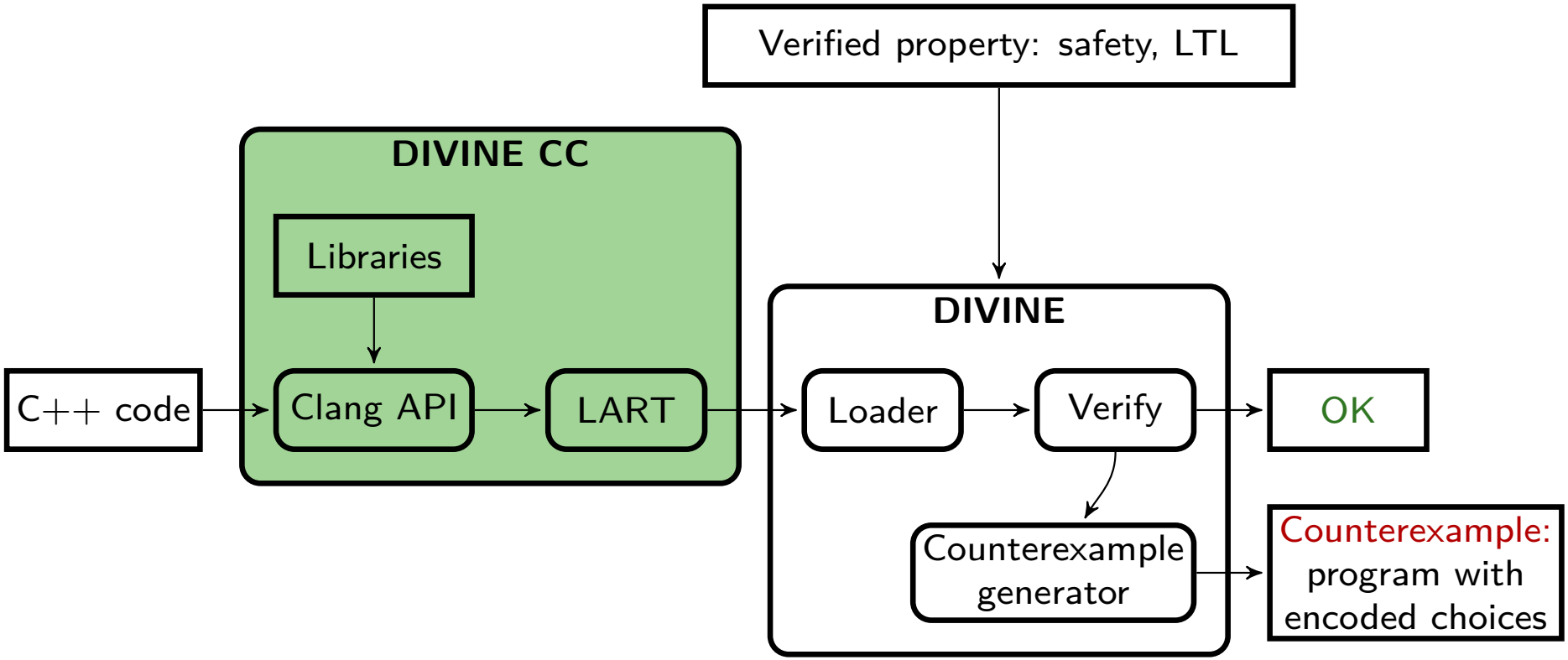
Brno, Czech Republic

15th June 2016

Verification of C++ programs



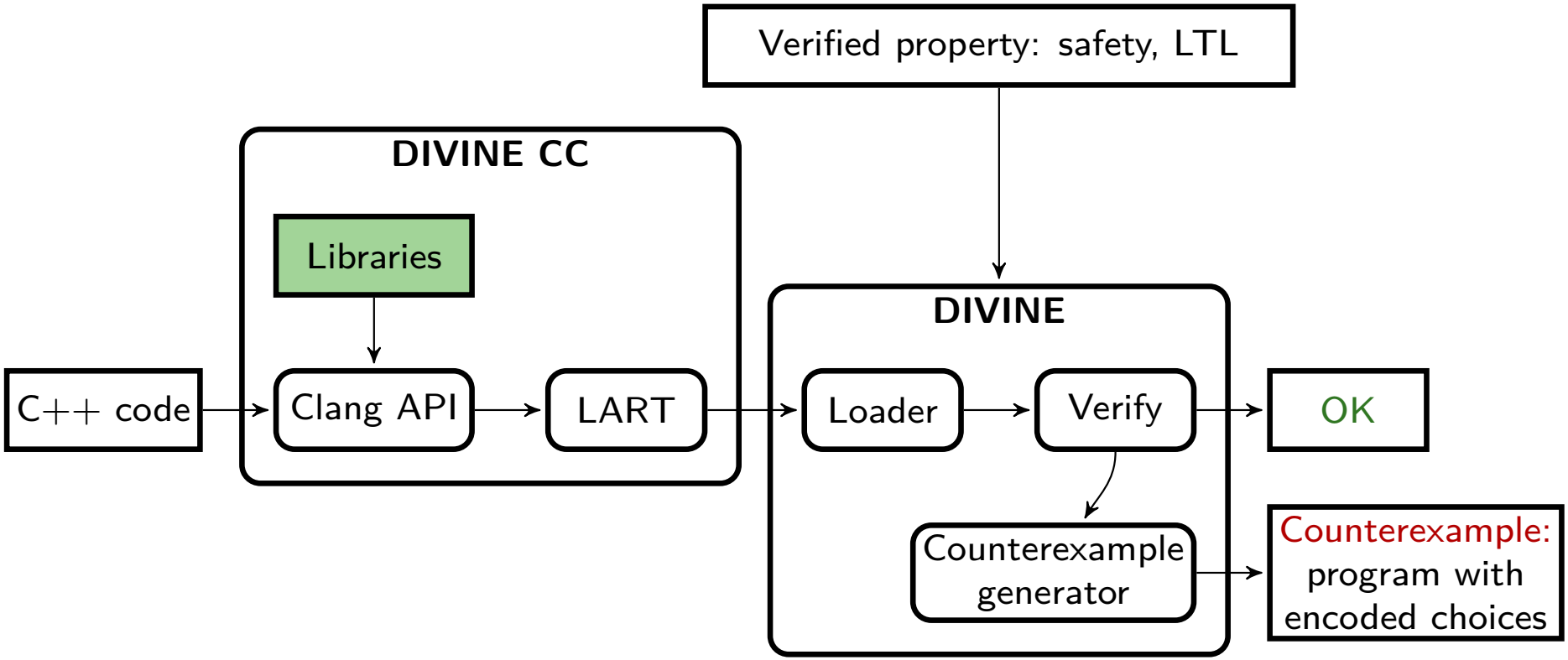
Verification of C++ programs



DIVINE compiler using Clang API and LLVM linker

- isolated from system libraries
- isolated from (most of) system defines

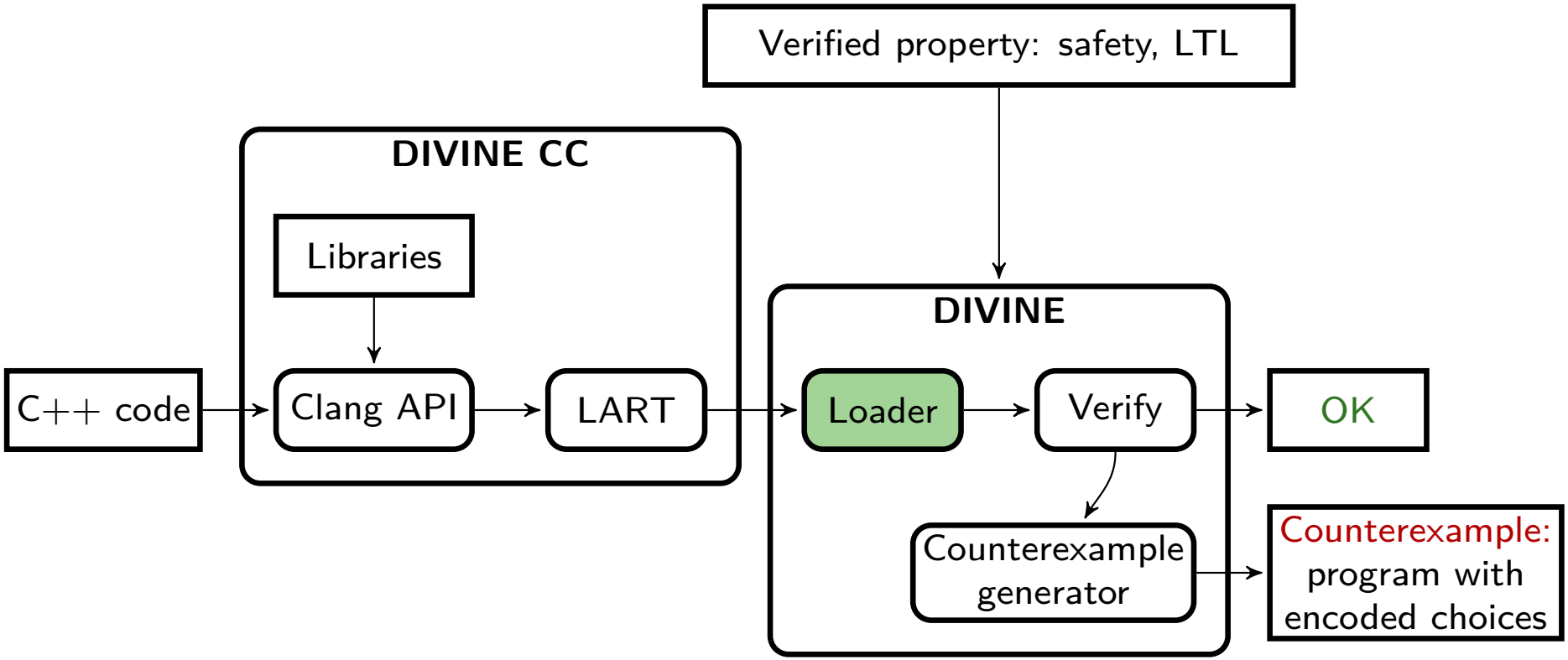
Verification of C++ programs



Libraries

- DIVINE has to have complete implementation of libraries
- PDClib for C99
- libc++ and libc++ ABI for C++14 (newly ported to DIVINE)
- pthreads, DIVINE runtime,. . .

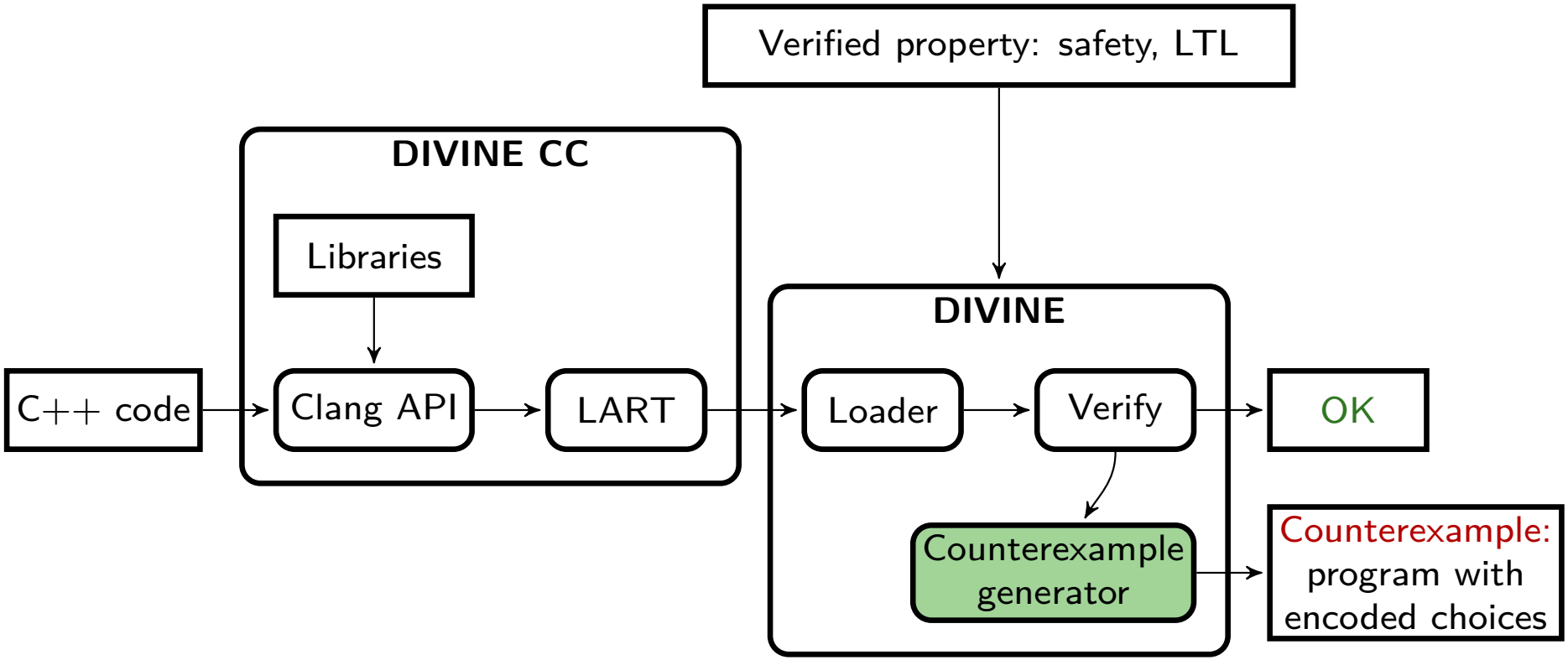
Verification of C++ programs



DIVINE Loader

- **LLVM instrumentation**: interrupt points, metadata, annotations
- command line arguments, file system
- conversion of LLVM to the internal representation

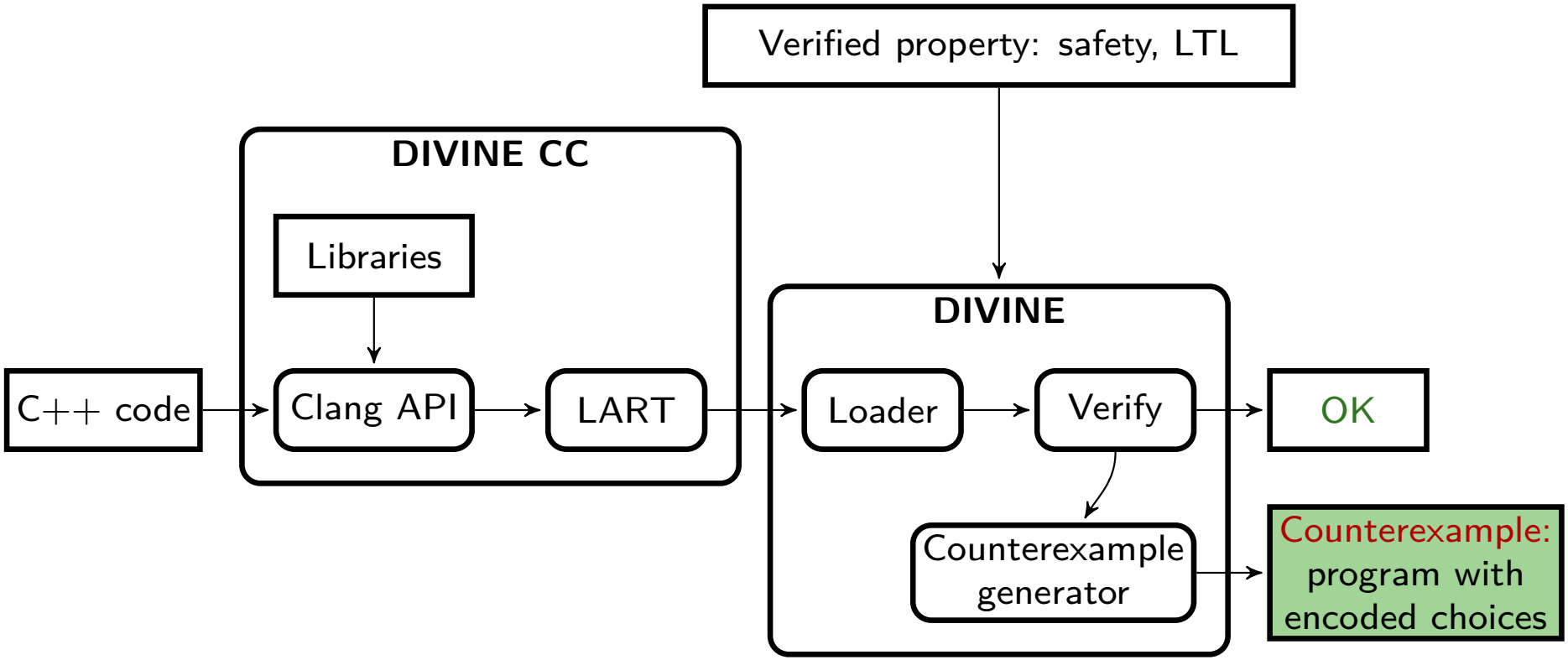
Verification of C++ programs



Counterexample generator

- algorithms will produce a counterexample as a list of choices
  - thread switches, nondeterminism
- embed choices into the program
- add implementation of DIVINE runtime

Verification of C++ programs



Counterexample

- can be executed as normal binary
- can be debugged using existing debuggers
- does not depend on system libraries

DIVINE CC

DIVINE 3 used a wrapper over system's Clang for compilation into LLVM

- depends on *the particular version* of Clang
  - the same version as used to build DIVINE
- can accidentally use system libraries

DIVINE CC

DIVINE 3 used a wrapper over system's Clang for compilation into LLVM

- depends on *the particular version* of Clang
  - the same version as used to build DIVINE
- can accidentally use system libraries

now DIVINE 4 uses fully integrated Clang API based compiler

- C++ API exports most of features of Clang
  - primarily intended for syntactic manipulation and refactoring tools and for Clang itself
- uses Clang's VFS for in-memory compilation and isolation
  - isolated from system headers (unless explicitly imported: `-I`)
- isolated from most of system defines (like `__unix__`)
- does not depend on any part of the system compiler

Instrumentations in DIVINE Loader

In DIVINE 4 the interpreter needs to be notified when

- a memory location visible to multiple threads is accessed
- a cycle in the state space can occur
  - safely approximated by cycles in the control flow

Instrumentations in DIVINE Loader

In DIVINE 4 the interpreter needs to be notified when

- a memory location visible to multiple threads is accessed
- a cycle in the state space can occur
  - safely approximated by cycles in the control flow

instrumenting the code allows more flexible interaction between DIVINE and static analysis in LART

- some interrupt points can be omitted
- it is easier to specify where the thread was interrupted
- it is possible to generate executable counterexamples

Instrumentations in DIVINE Loader

Loader also adds metadata for each function:

- function pointer
- number of arguments
- argument types
- frame size
- instruction metadata

Instrumentations in DIVINE Loader

Loader also adds metadata for each function:

- function pointer
- number of arguments
- argument types
- frame size
- instruction metadata

Annotations to facilitate debugging/counterexamples

- printed when a function is entered or left (including unwinding)

Executable Counterexamples

the bitcode already contains C and C++ library implementation

- native implementation of the DIVINE runtime has to be added
- simulates counterexample found by DIVINE
- no parallelism, instead manages and switches separate stacks
- support now planned only for `x86_64` on Linux
  - depends on calling conventions, syscalls
- work in progress

Future Work

- Finish executable counterexamples
- DIVINE CC which works as a drop-in replacement for Clang/GCC

  ```
  $ CC=divine.cc CXX=divine.cc make
  ```

- optimized instrumentations for the loader
  - statically detect some cases when interrupts cannot happen
    - local variables
    - cycles which must terminate
- calculate frame layout in the loader
  - includes allocation of LLVM registers into memory slots
  - can save memory

Students

Peter Hutta

- found suspected memory leak in `boost` parallel queue
  - not yet tracked to the code

Students

Peter Hutta

- found suspected memory leak in `boost` parallel queue
  - not yet tracked to the code

Jakub Kadaši

- works on DIVINE VFS in order to allow verification of `lvmetad`

Students

Peter Hutta

- found suspected memory leak in `boost` parallel queue
  - not yet tracked to the code

Jakub Kadaši

- works on DIVINE VFS in order to allow verification of `lvmetad`

Katarína Kejstová

- works on DIVINE VFS
- also on DIVINE 4: shadow maps – metadata for heap memory
  - pointer and value-initialization tracking

Students

Peter Hutta

- found suspected memory leak in `boost` parallel queue
  - not yet tracked to the code

Jakub Kadaši

- works on DIVINE VFS in order to allow verification of `lvmetad`

Katarína Kejstová

- works on DIVINE VFS
- also on DIVINE 4: shadow maps – metadata for heap memory
  - pointer and value-initialization tracking

Jan Mrázek

- will present Caching in Control-Explicit Data-Symbolic Model Checking
- also works on DIVINE 4 on DIVINE runtime