Using Off-the-Shelf Exception Support Components in C++ Verification

Vladimír Štill Petr Ročkai Jiří Barnat

Masaryk University

Brno, Czech Republic

26th July 2017

DIVINE – Verification of C++ Programs

DIVINE is a tool for testing and verification of C/C++ programs

- memory safety, assertion safety, parallelism errors
- easy error injection
- full support for C and C++, partial support for POSIX
- using clang/LLVM compiler infrastructure

DIVINE – Verification of C++ Programs

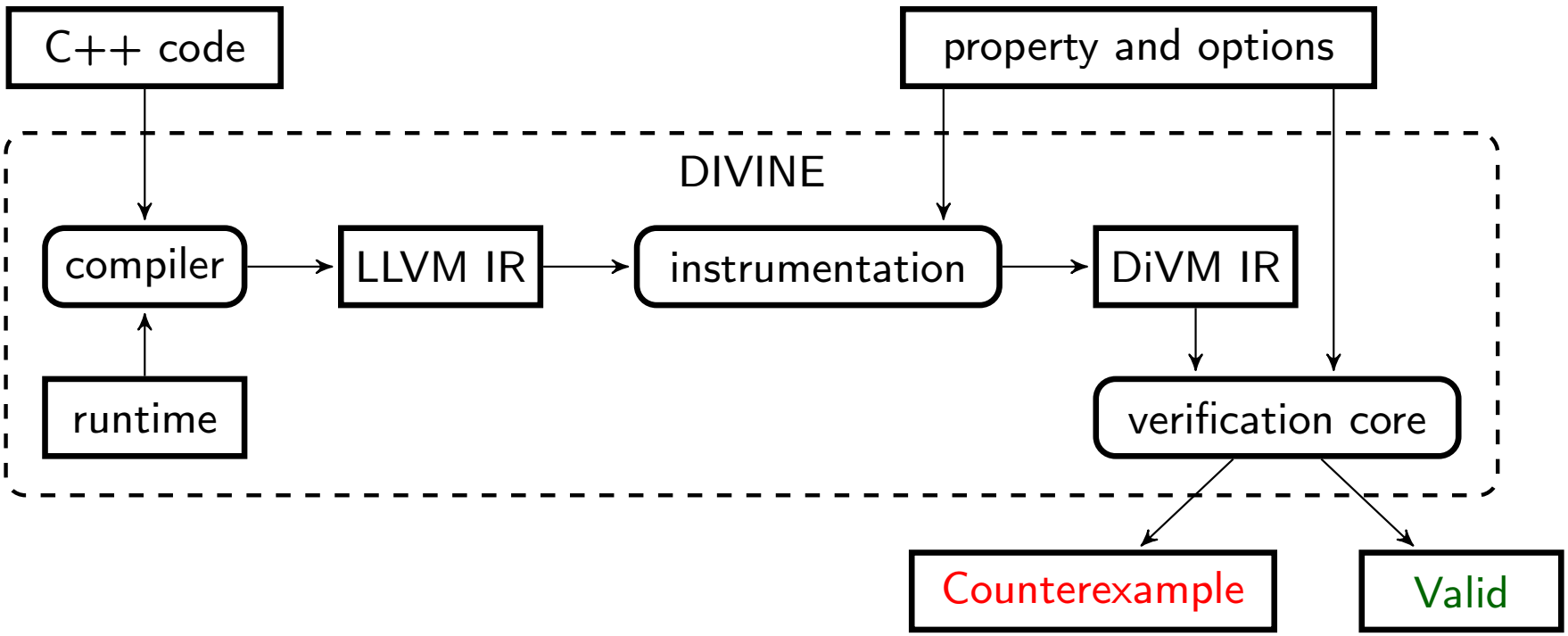
DIVINE is a tool for testing and verification of C/C++ programs

- memory safety, assertion safety, parallelism errors
- easy error injection
- full support for C and C++, partial support for POSIX
- using clang/LLVM compiler infrastructure

Contribution of this Work

- full support for C++ exceptions
- with minimal changes to the verification core of DIVINE
- re-using existing implementation of exception matching in the C++ runtime

DIVINE – Verification of C++ Programs



Motivation

C++ exceptions

- ubiquitous in real-world C++
- disabling exceptions can change behaviour (`new`)
- runtime support required, cannot be handled by the compiler itself

Motivation

C++ exceptions

- ubiquitous in real-world C++
- disabling exceptions can change behaviour (`new`)
- runtime support required, cannot be handled by the compiler itself

Off-the-Self Components

- using LLVM and clang helps a lot for C/C++ support
- DIVINE also re-uses C and C++ standard libraries
- more precise verification then with re-implementation of C++ support

Motivation

C++ exceptions

- ubiquitous in real-world C++
- disabling exceptions can change behaviour (`new`)
- runtime support required, cannot be handled by the compiler itself

Off-the-Self Components

- using LLVM and clang helps a lot for C/C++ support
- DIVINE also re-uses C and C++ standard libraries
- more precise verification then with re-implementation of C++ support

- exceptions support is complex
- re-implementation would risk imprecisions, would be large, or require changes to the verification core

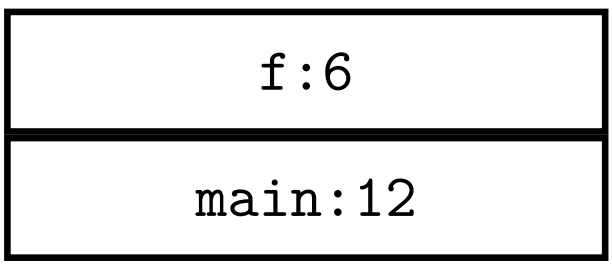
How Exceptions Work

```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();   <-------------------------
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```

main:12

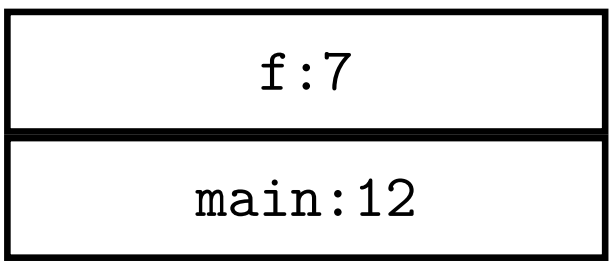
How Exceptions Work

```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;   <-------------------------
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```



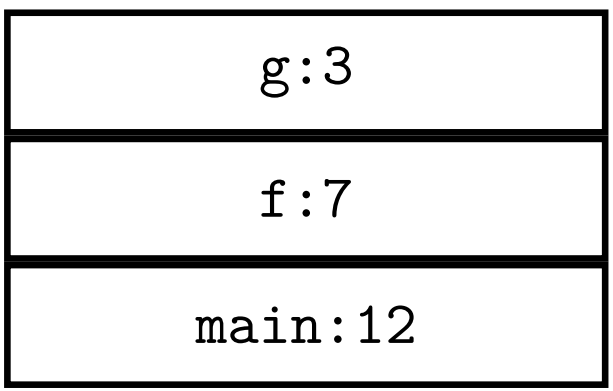
How Exceptions Work

```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();   <-------------------------
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```



How Exceptions Work

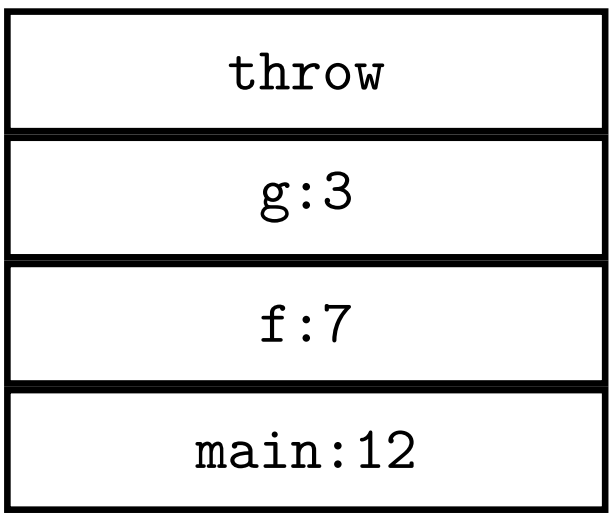
```
1  X::~X() { }
2  void g() {
3    throw std::exception();   ←
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```



How Exceptions Work

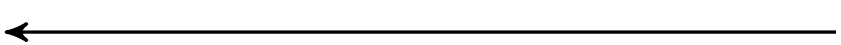
```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```

unwinding

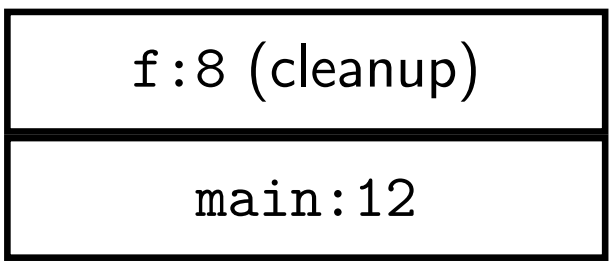


How Exceptions Work

```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```



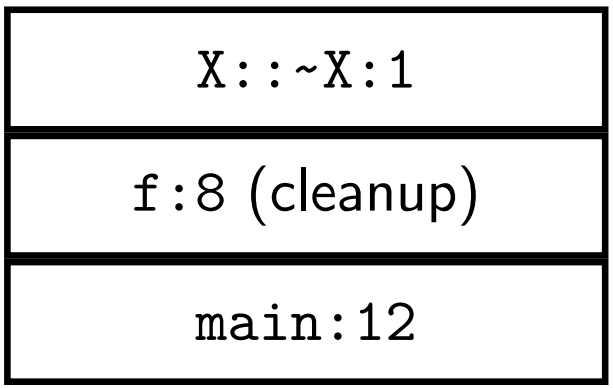
unwinding



How Exceptions Work

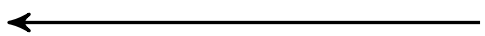
```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```

unwinding



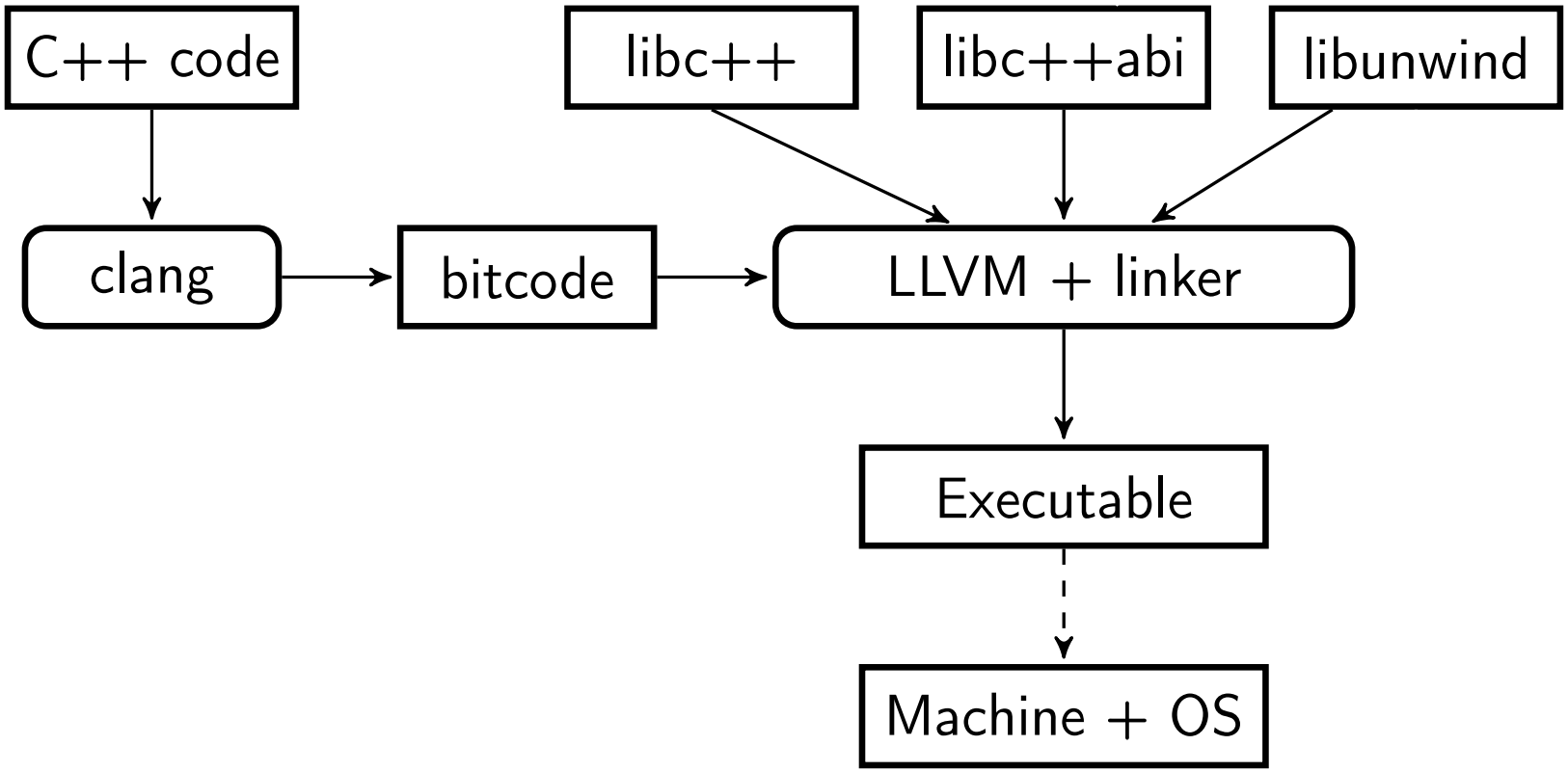
How Exceptions Work

```
1  X::~X() { }
2  void g() {
3    throw std::exception();
4  }
5  void f() {
6    X x;
7    g();
8  }
9
10 int main() {
11   try {
12     f();
13   } catch ( ... ) {
14     /* ... */
15   }
16 }
```



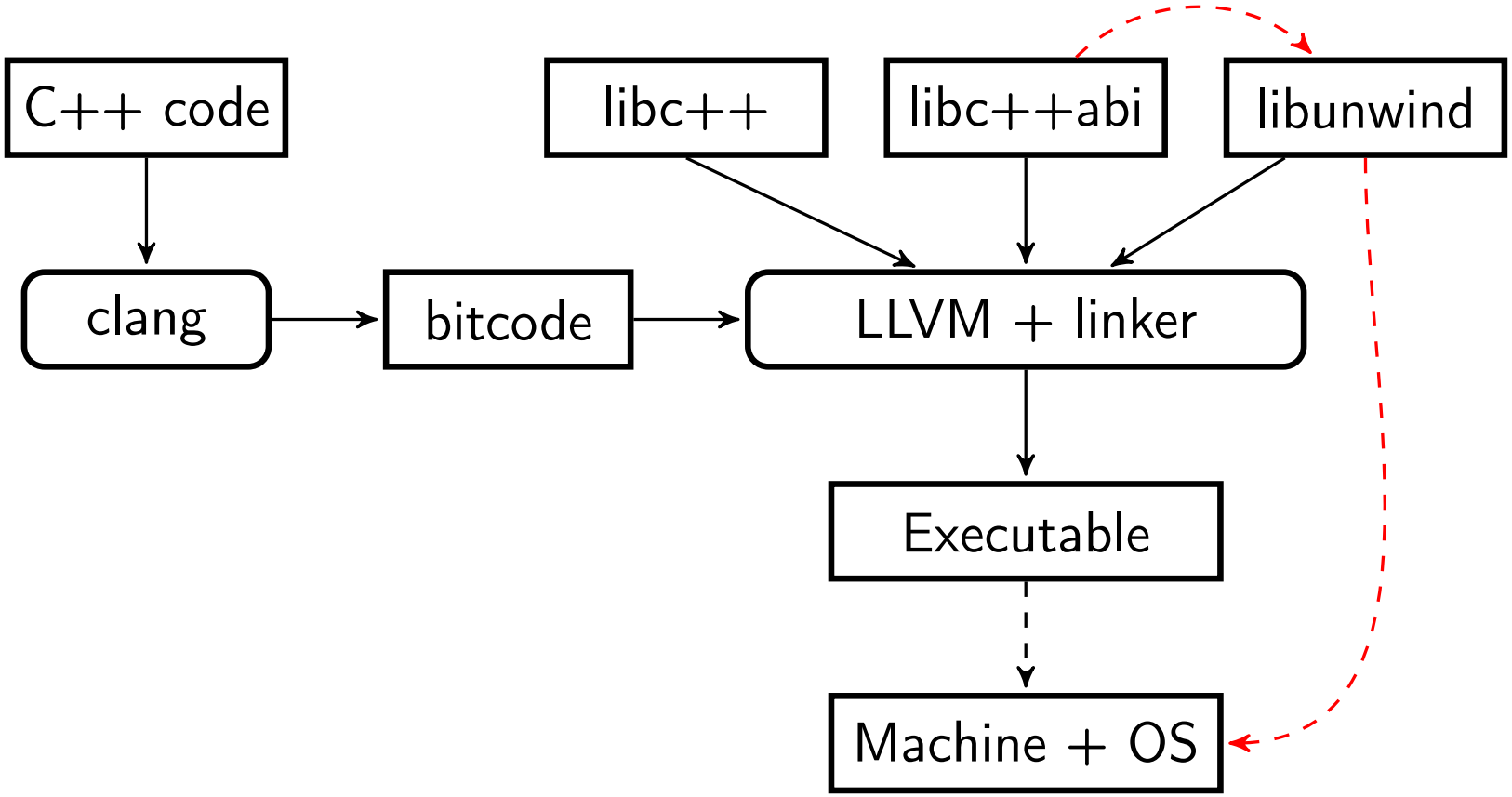
main:14

Running C++ Program



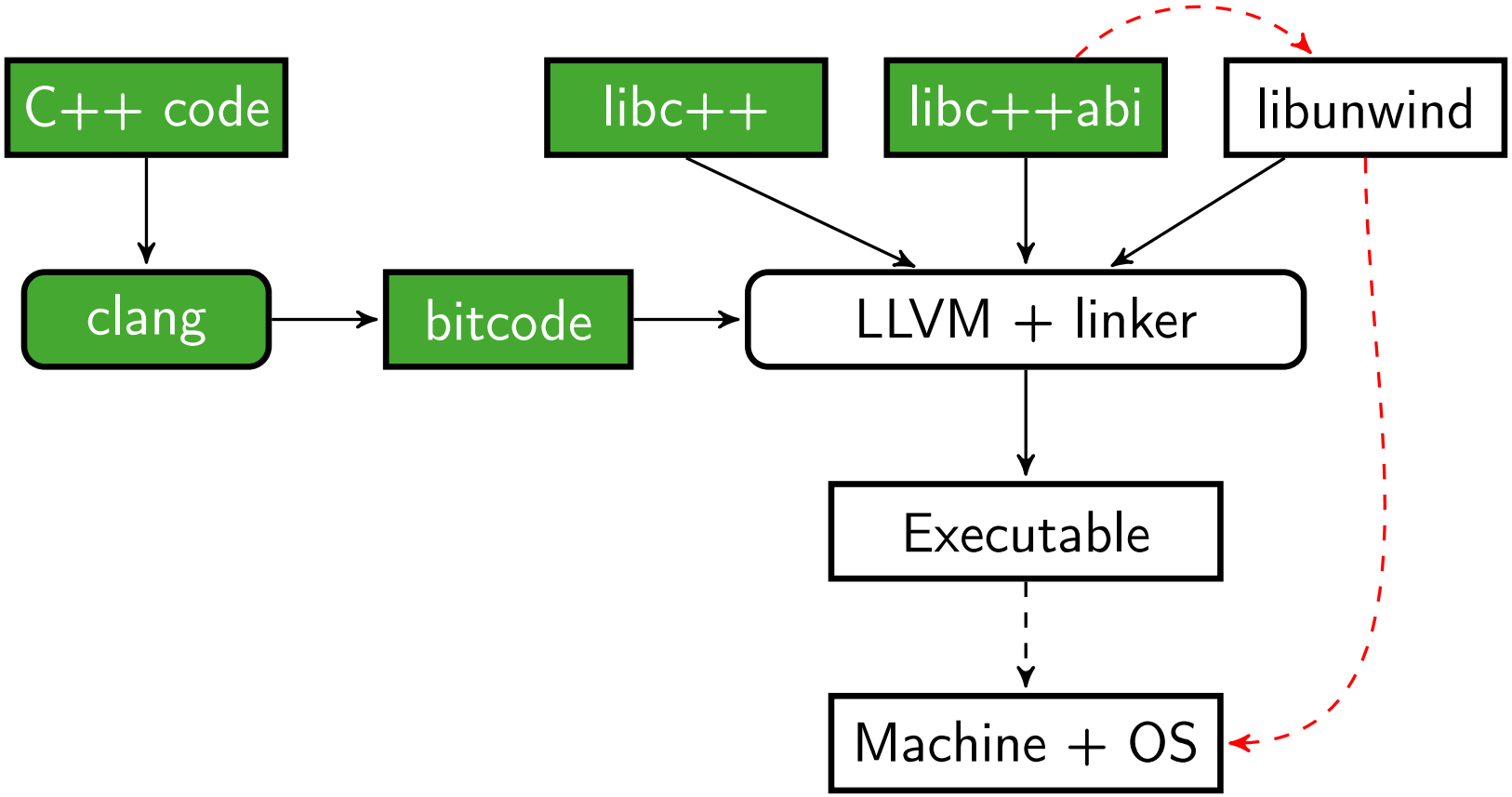
- the code is compiled and linked to the standard library (`libc++`), runtime library (`libc++abi`), and the unwinder (`libunwind`)

Running C++ Program



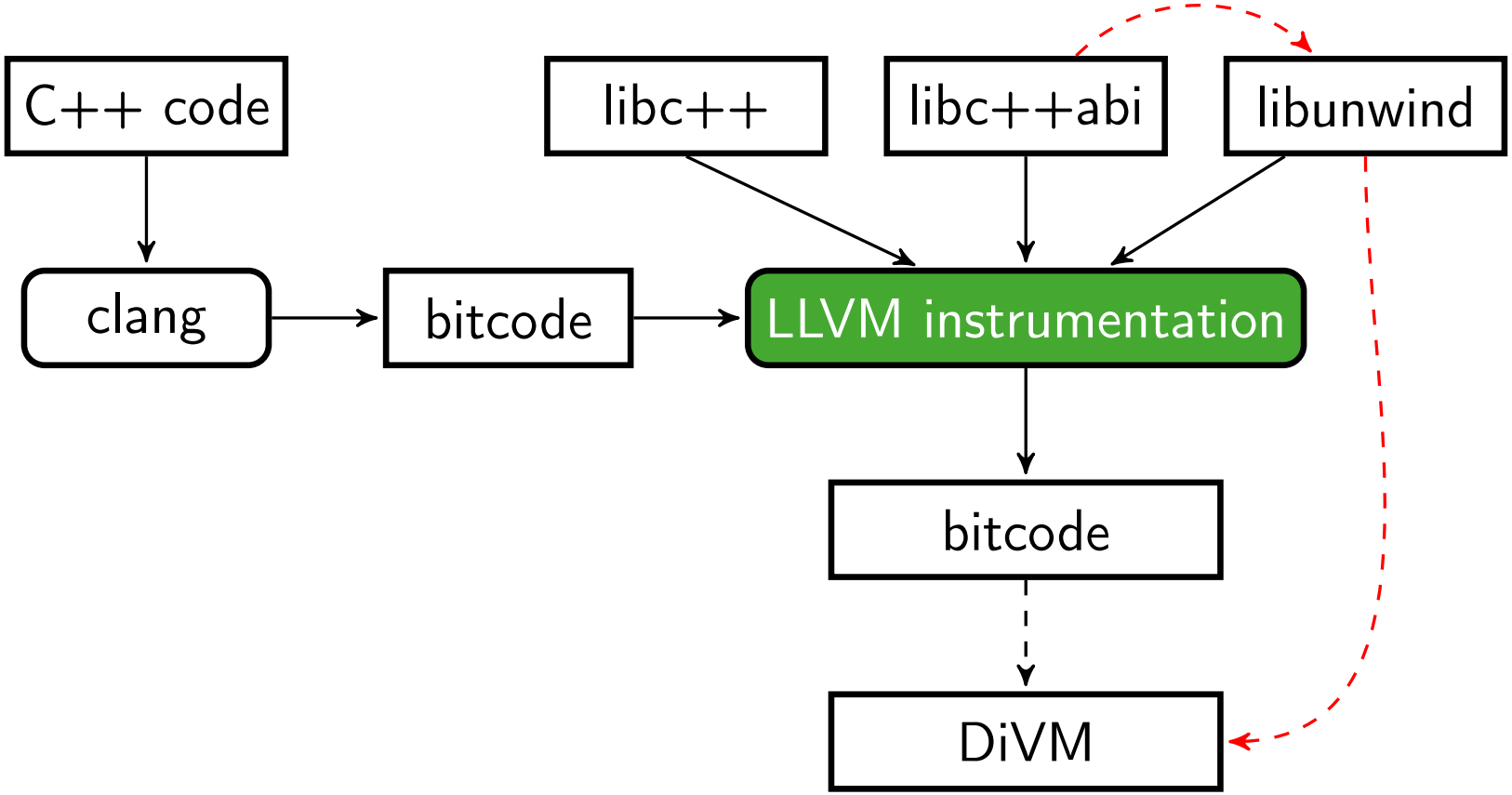
- the code is compiled and linked to the standard library (`libc++`), runtime library (`libc++abi`), and the unwinder (`libunwind`)
- the runtime library depends on the unwinder which depend on the machine and OS

Running C++ Program



- the code is compiled and linked to the standard library (`libc++`), runtime library (`libc++abi`), and the unwinder (`libunwind`)
- the runtime library depends on the unwinder which depend on the machine and OS
- green components are re-used in DIVINE

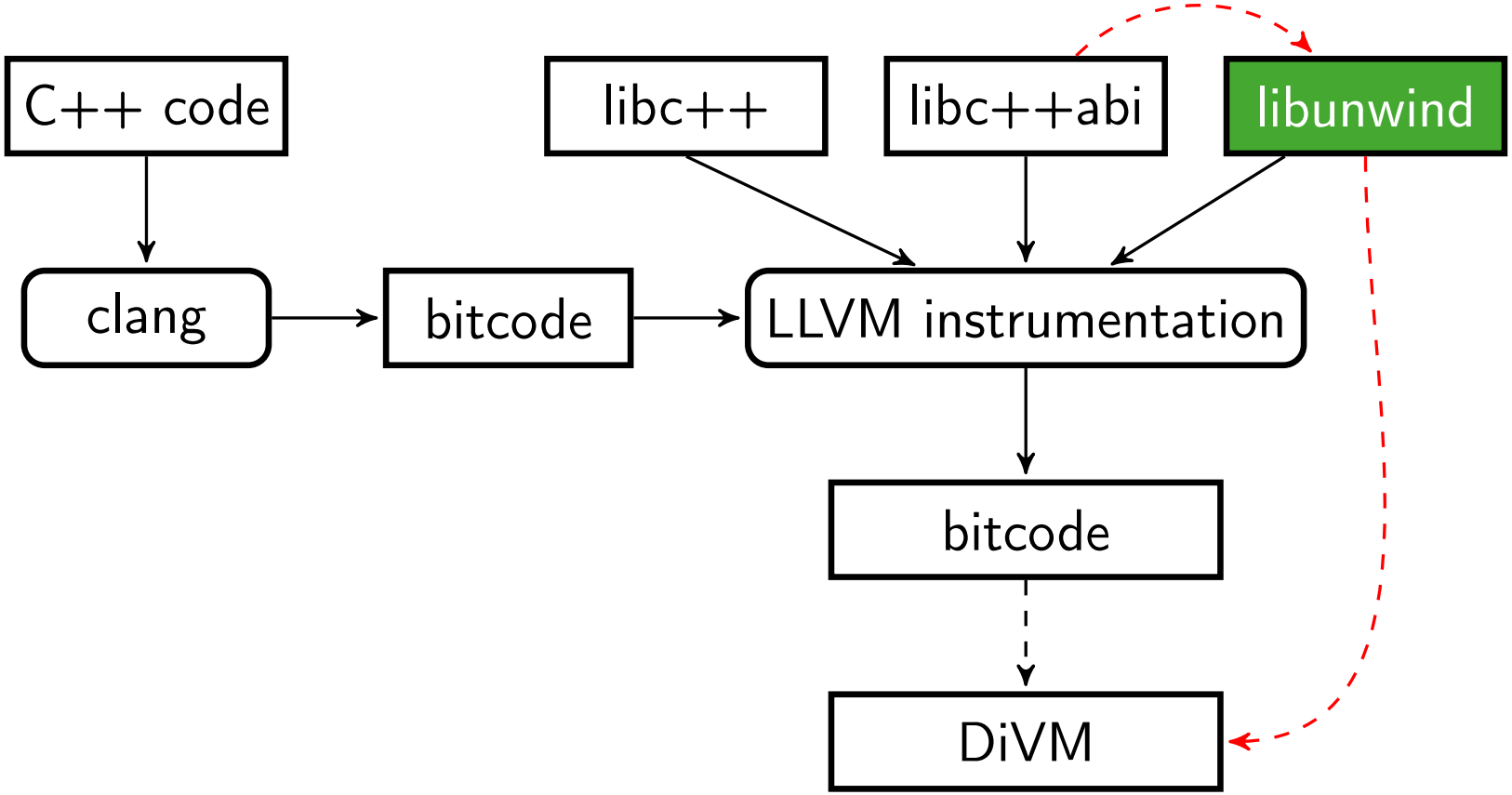
Analyzing C++ Program with DIVINE



DIVINE/DiVM-specific components

- LLVM-based preprocessing

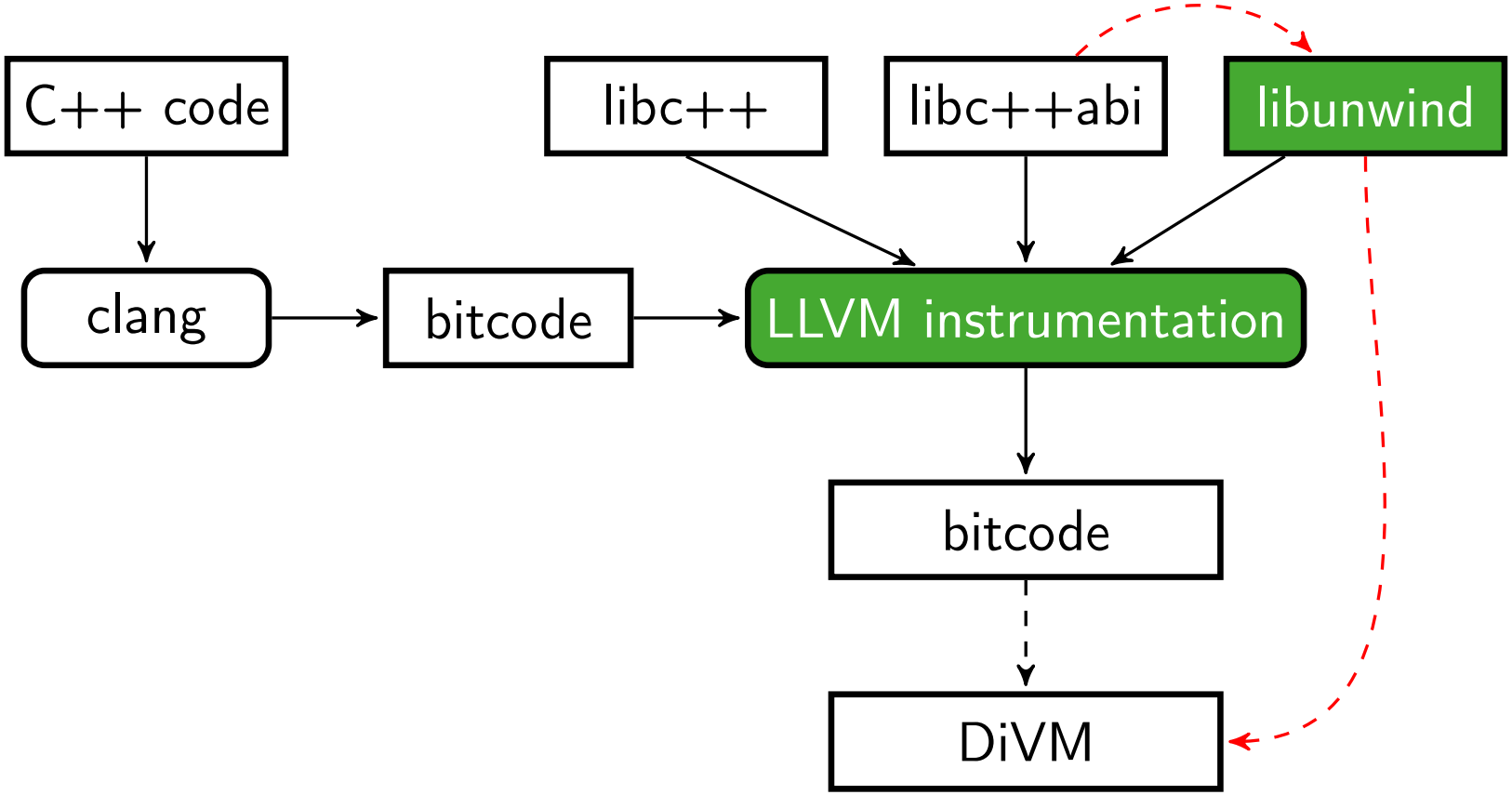
Analyzing C++ Program with DIVINE



DIVINE/DiVM-specific components

- LLVM-based preprocessing
- DiVM-based implementation of `libunwind`

Analyzing C++ Program with DIVINE



DIVINE/DiVM-specific components

- LLVM-based preprocessing
- DiVM-based implementation of `libunwind`
- approximately 700 lines of new modular C++ code

LLVM Transformation

- exceptions require metadata about stack frames, catch blocks and cleanups for destructors
  - normally describe the machine code
  - DIVINE needs metadata for LLVM bitcode

LLVM Transformation

- exceptions require metadata about stack frames, catch blocks and cleanups for destructors
  - normally describe the machine code
  - DIVINE needs metadata for LLVM bitcode
- metadata format depends on the implementation of the C++ runtime library

LLVM Transformation

- exceptions require metadata about stack frames, catch blocks and cleanups for destructors
  - normally describe the machine code
  - DIVINE needs metadata for LLVM bitcode
- metadata format depends on the implementation of the C++ runtime library
- output of the transformation is LLVM bitcode with additional metadata stored in global constants
- C++ specific encoding of catch and cleanup locations

The Unwinder (`libunwind`)

- used to manipulate the execution stack
- depends on the platform, calling conventions (e.g. Linux on `x86`)

The Unwinder (`libunwind`)

- used to manipulate the execution stack
- depends on the platform, calling conventions (e.g. Linux on `x86`)
- new unwinder for DiVM

The Unwinder (`libunwind`)

- used to manipulate the execution stack
- depends on the platform, calling conventions (e.g. Linux on `x86`)
- new unwinder for DiVM
- uses metadata from the transformation
- provides metadata for the `libc++abi` callbacks which search for the location to restore control flow to

The Unwinder (`libunwind`)

- used to manipulate the execution stack
- depends on the platform, calling conventions (e.g. Linux on `x86`)
- new unwinder for DiVM
- uses metadata from the transformation
- provides metadata for the `libc++abi` callbacks which search for the location to restore control flow to
- would also work with other languages

Evaluation & Conclusion

- reusable and modular implementation of C++ exceptions
- substantial improvement in verification fidelity

Evaluation & Conclusion

- reusable and modular implementation of C++ exceptions
- substantial improvement in verification fidelity
- minimal investment: $\sim$ 700 lines of code

Evaluation & Conclusion

- reusable and modular implementation of C++ exceptions
- substantial improvement in verification fidelity
- minimal investment: $\sim$ 700 lines of code
- minimal overhead: 2.6 % time overhead compared to an older style of implementation which required changes to the verification core

`divine.fi.muni.cz`
`paradise-fi/divine` on GitHub

more data & code:
`divine.fi.muni.cz/2017/exceptions`