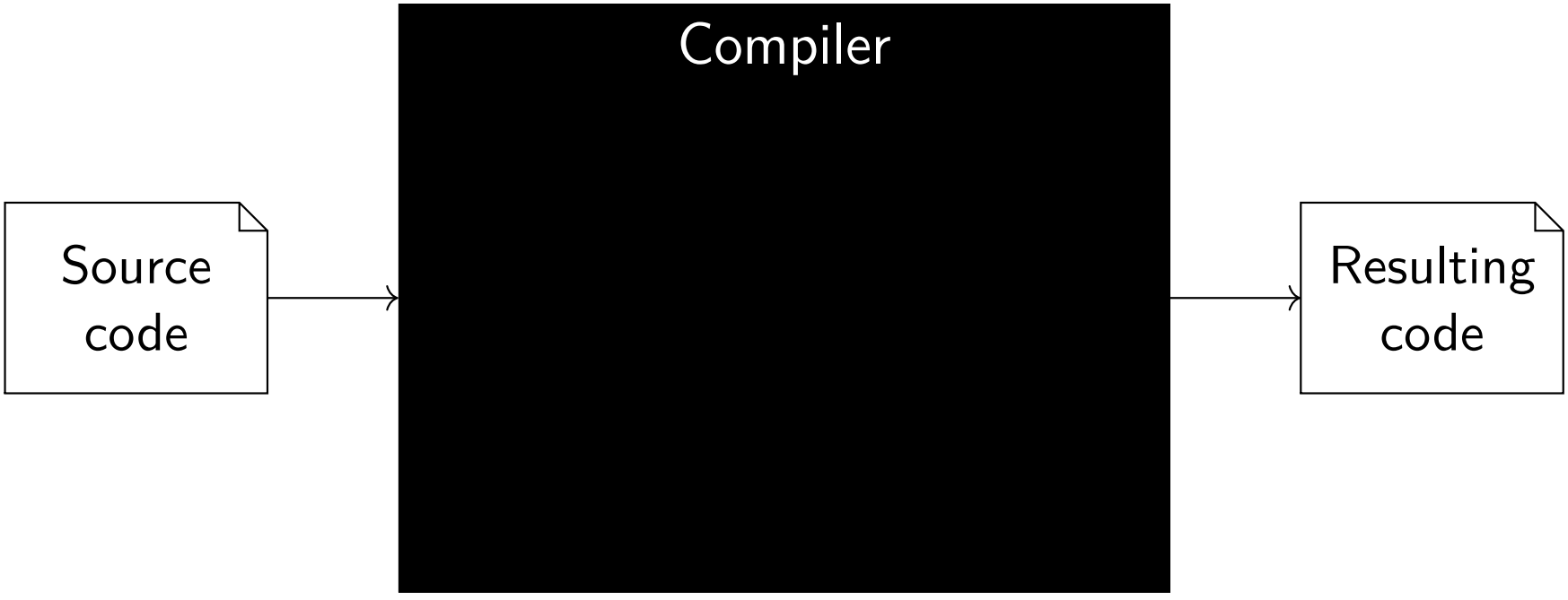
Compilation

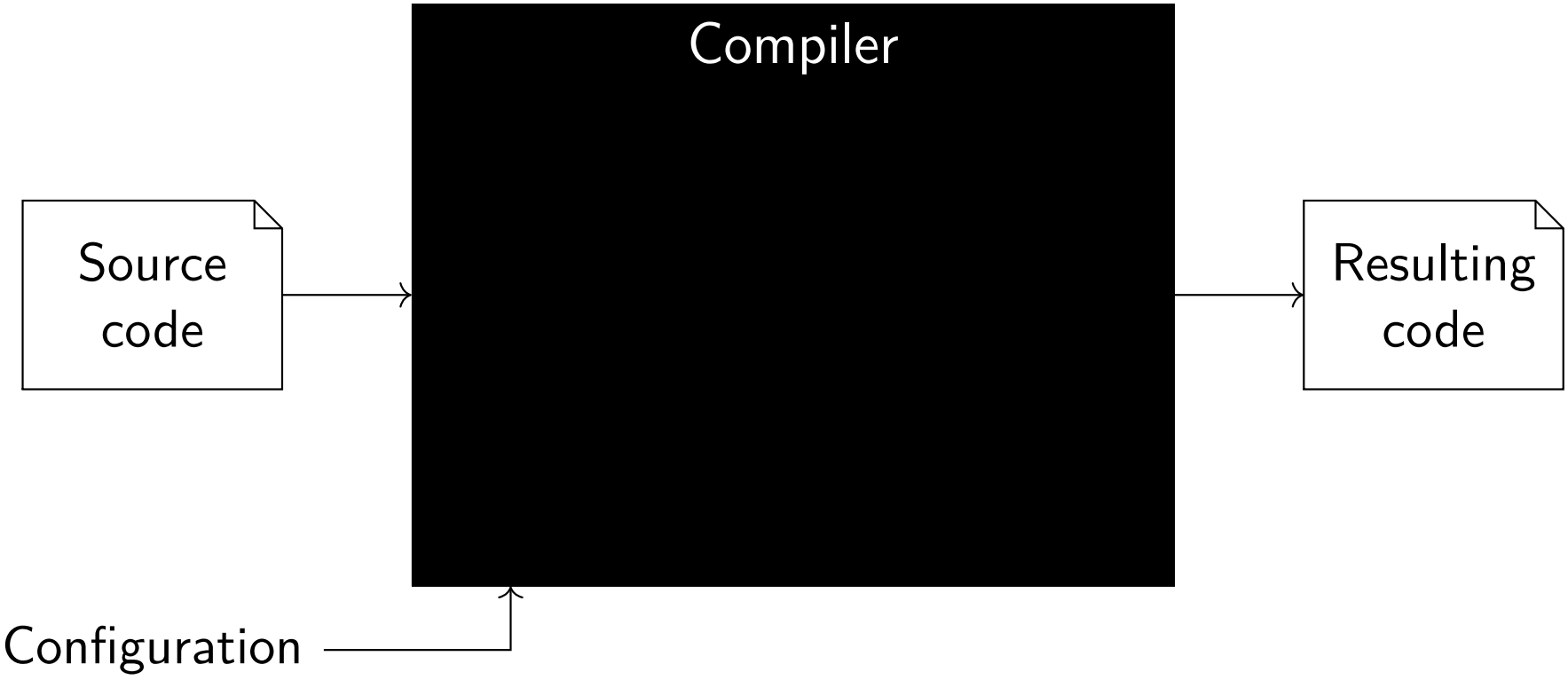
Work within P4S

F. Dagnat et al., fabien.dagnat@imt-atlantique.fr

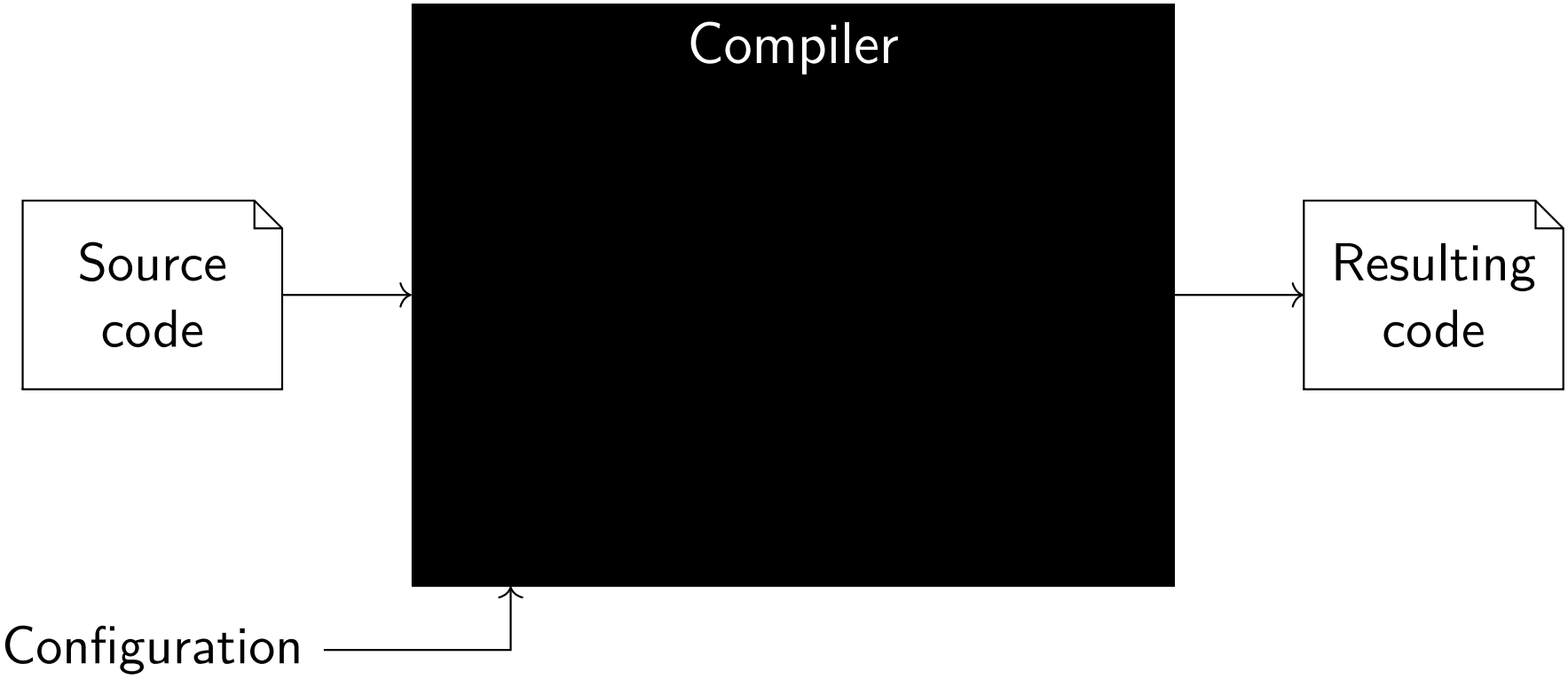
15/12/2022



- Source code: any programming language
- Compiler: `gcc`, `clang`, `javac`, `ocamlc`/`ocamlopt`...
- Resulting code: executable binary, *object*, *bytecode*

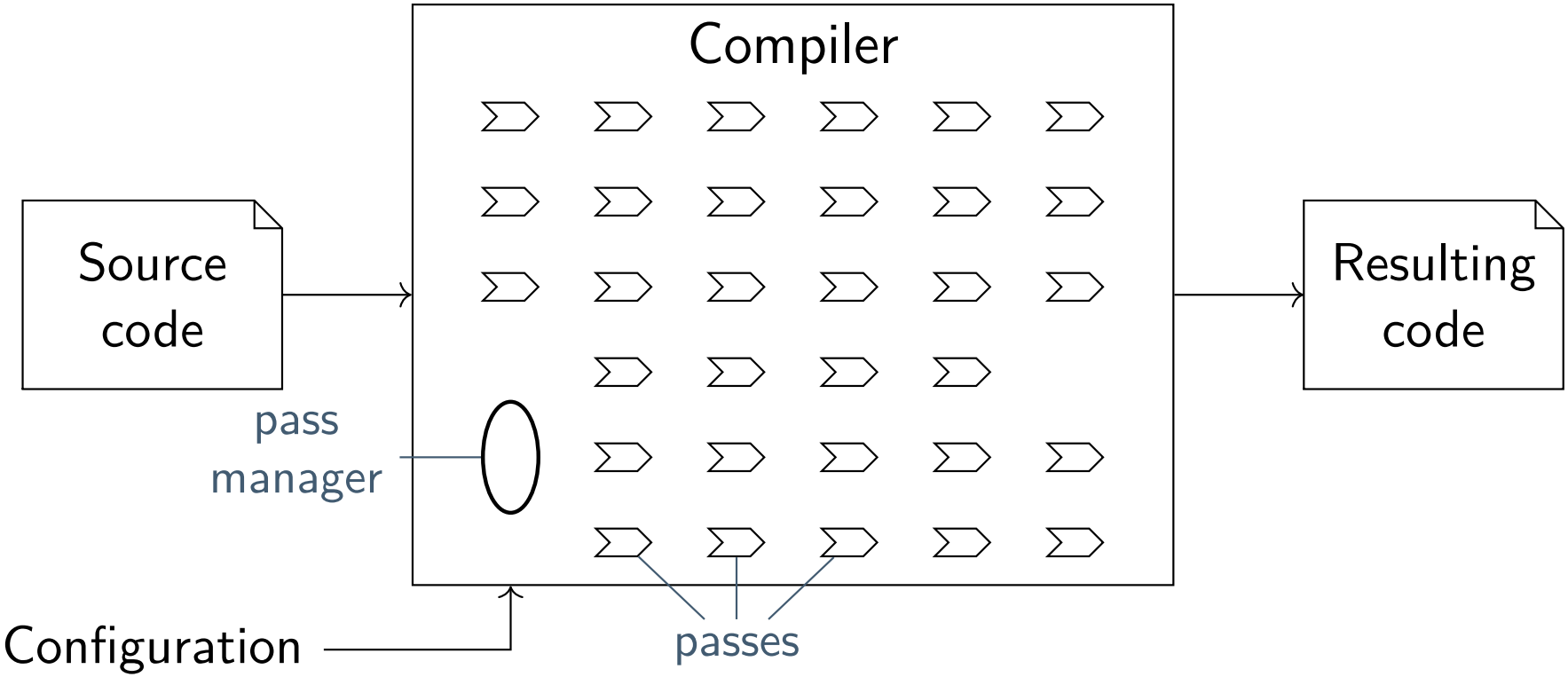


- Source code: any programming language
- Compiler: `gcc`, `clang`, `javac`, `ocamlc`/`ocamlopt`...
- Resulting code: executable binary, *object*, *bytecode*
- Configuration: command line options, source annotation...

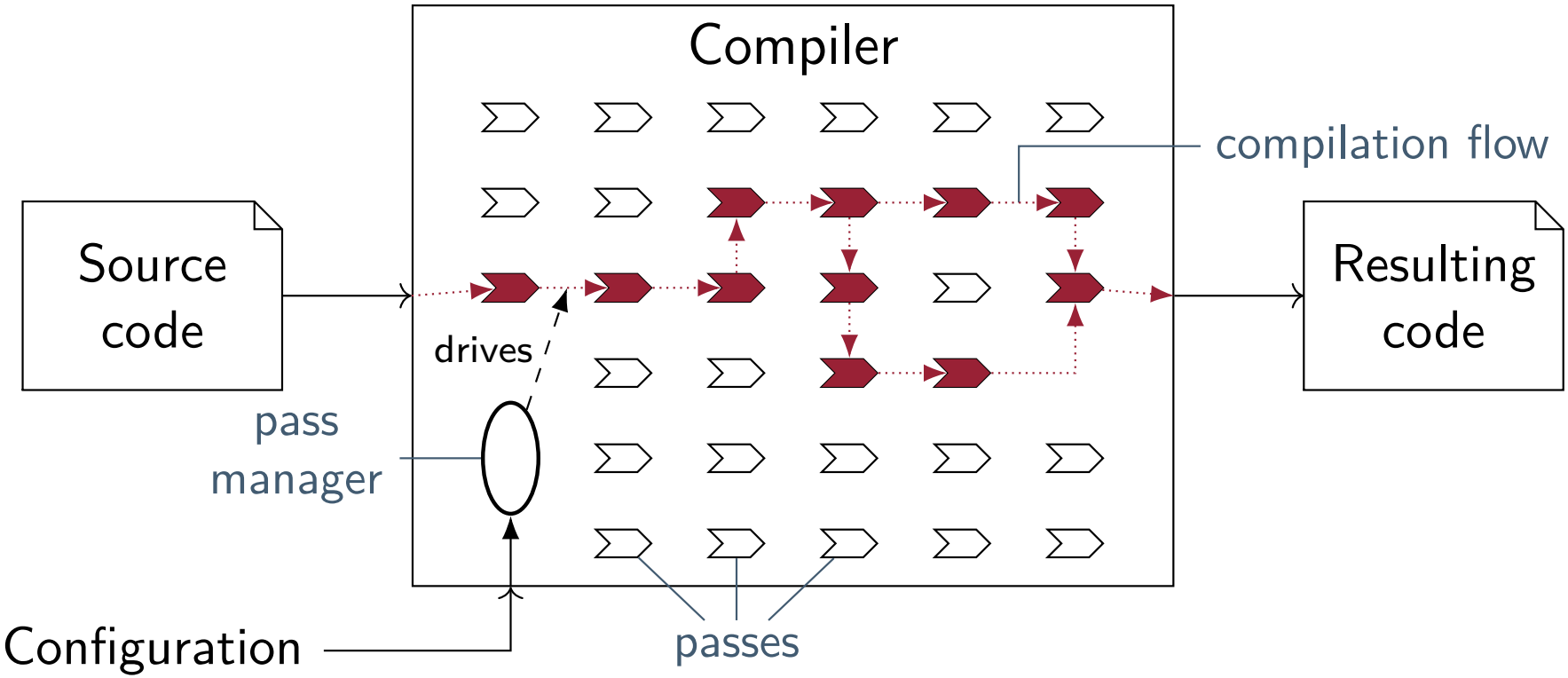


- "All" software are built by a compiler

$\Rightarrow$ Compiler are a critical piece of software



- LLVM contains 267 passes
- Mainly optimizing passes aiming at performance



- LLVM contains 267 passes
- Mainly optimizing passes aiming at performance
- Driven by 109 basic options and 9 combined options (`-O*`)

- Large complexity
  - History: from complex algorithms to complex combination
  - Difficult to know the real compilation flow
  - Difficult to control the interferences between passes

$\Rightarrow$ How to trust a compiler?

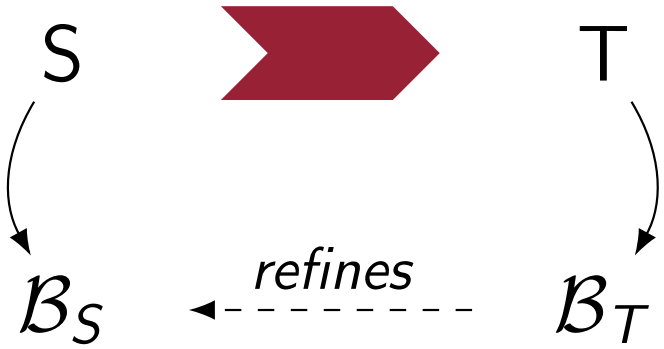
- New objectives for the executable code
  - Energy sobriety
    - Compilation for embedded systems but not the same requirements
  - Security (adding protection to the executable)
    - Flow integrity, obfuscation, constant time

$\Rightarrow$ How to integrate such new passes?

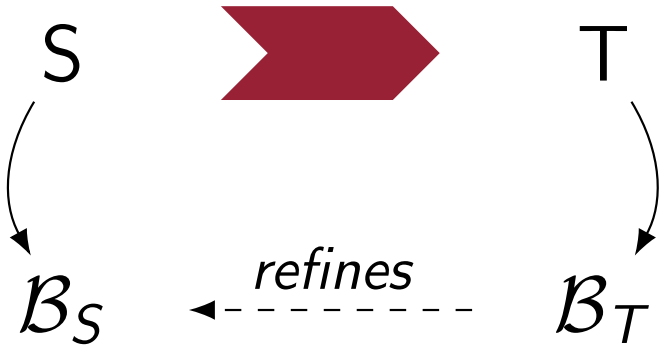
- More and more high-level and specific features
  - Object oriented, OpenMP, Machine Learning, ...
- More and more low-level specific hardware
  - High efficiency CPU, high perf. CPU, GPU, neural network units, ...

$\Rightarrow$ How to manage and combine so many different passes?

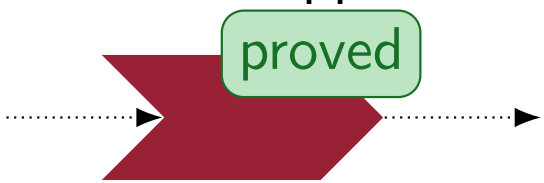
- There is a large and active community working on proofs of passes
- Correctness based on (observable) behaviors

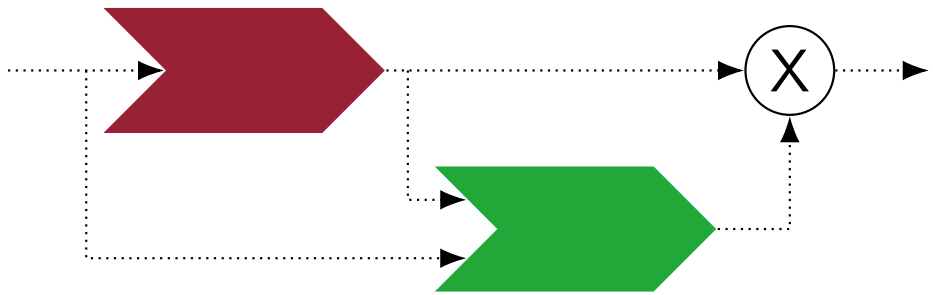


- There is a large and active community working on proofs of passes
- Correctness based on (observable) behaviors

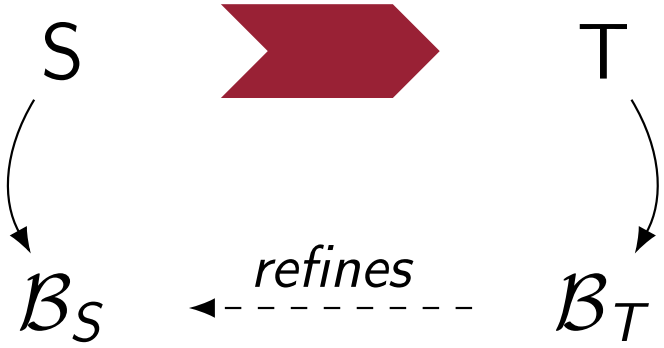


- Two main approaches pass proof or pass validation

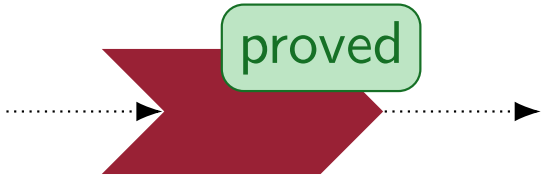


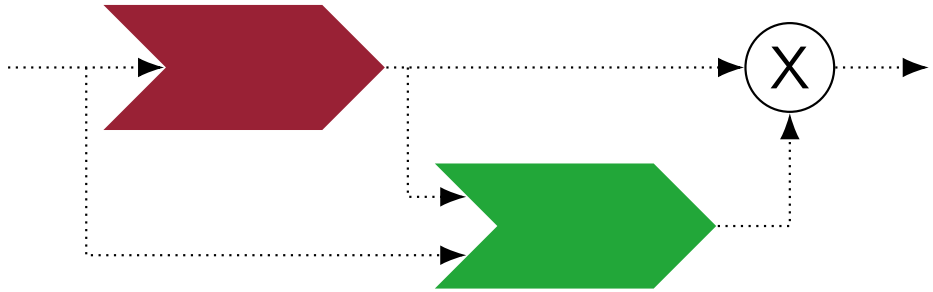


- There is a large and active community working on proofs of passes
- Correctness based on (observable) behaviors



- Two main approaches pass proof or pass validation





- Mainly for linear and static pipeline

- Complex compilation flows are not pipelines
  - *e.g.* for obfuscation
  - some optimization must be executed after some obfuscations
  - sometimes no optimization must occur between two obfuscation passes

$\Rightarrow$ This requires DSL for flow specification supporting verification

$\Rightarrow$ Contracts for the passes

- But how do we know which flow has been applied?

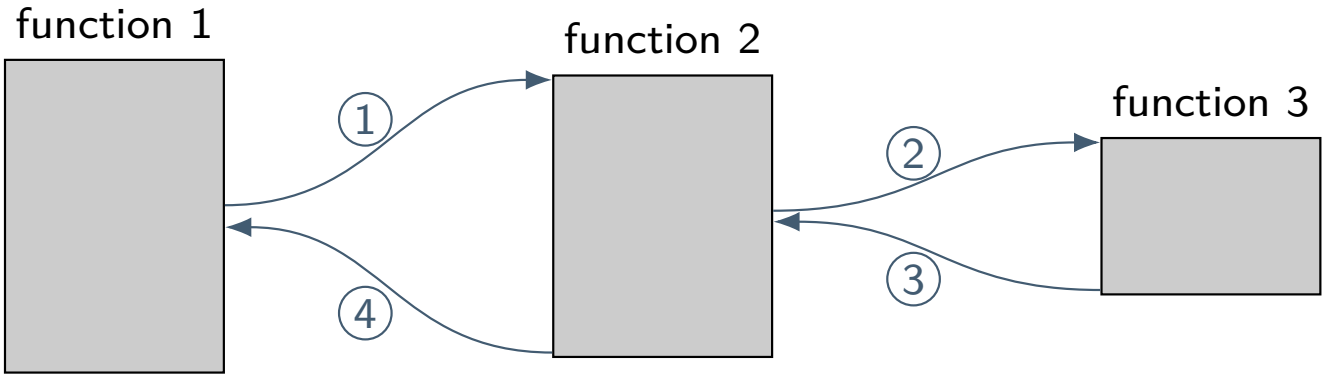
$\Rightarrow$ Traceability of the execution of a compilation pass

- This brings us to a form of *certification of compilation*

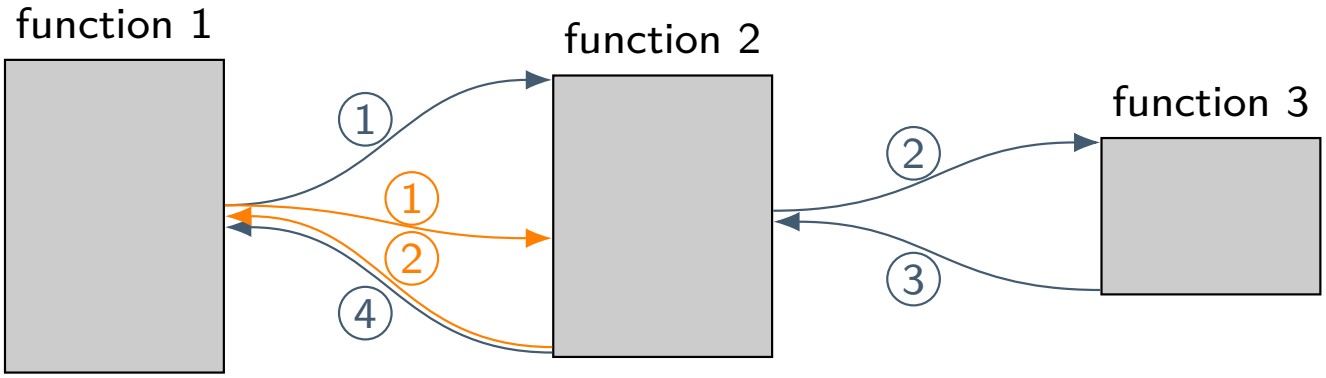
- Work of NS and BM later

- A pass is seen as a *pure black box* function
- A *program element* (a function) has a type recording a "set" of properties using *tags* and a presence qualifier (Pre, Abs, $\top$)
  - $\langle \textsc{ToProt} : \text{Pre}; \textsc{CFProt} : \text{Abs}; ...; \partial\top \rangle$
- A pass has a type
  - its preconditions are expressed by presence of *tags* on arguments
  - its effect is expressed by presence of *tags* on results
  - $\tau_1 \to \tau_2$ `with` $(\tau_1 <: \langle \textsc{CFProt} : \text{Abs}; \partial\top \rangle) \wedge \tau_2 = \tau_1 \oplus \textsc{CFProt}$
- A compilation script is a functional expression combining passes
- It is typed *gradually*
  - Types are inferred and checked statically
  - An unknown type is allowed
  - When checking an unknown type, a dynamic type test is added to the code

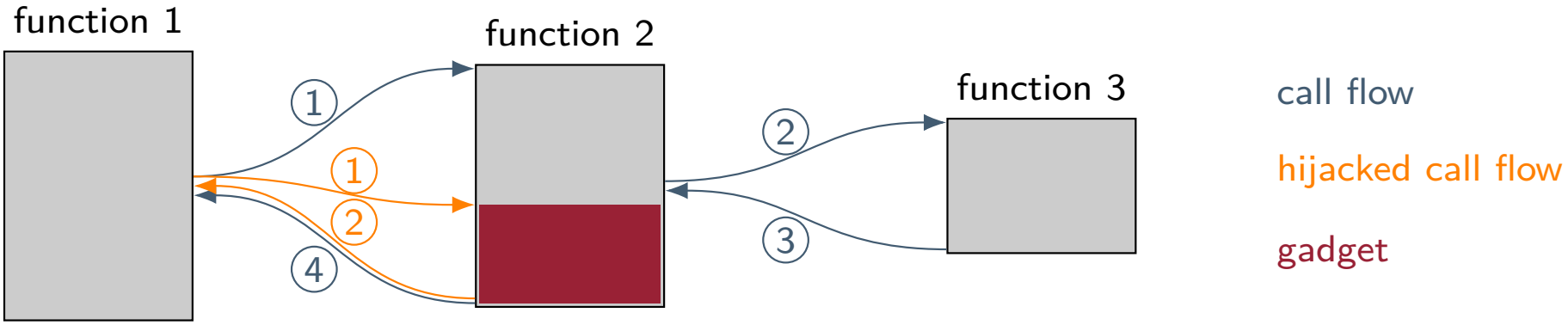
$\Rightarrow$ A DSL for compilation + verification of compilation flows



- A function call is compiled to a branching



- A function call is compiled to a branching
- An attacker can corrupt this branching
  - Using software ot hardware



- A function call is compiled to a branching
- An attacker can corrupt this branching
  - Using software ot hardware
- Allow to execute only the chosen instructions
- By chaining gadgets, the attacker can build any program
  - As soon as the binary is sufficiently big, it is Turing-complete
- Here, ROP but there is also BROP, JOP, ...

$\Rightarrow$ Makes it possible to execute a program even when the attacker cannot upload code to memory ($W \oplus X$)

- Context
  - An attacker hijacks a binary
  - Developers do not know how their sources are compiled

  $\Rightarrow$ How to help a developer *w.r.t* control flow hijack?
- Experimental setting
  - To build a large set of binaries in various format
  - To collect all the gadgets (of certain maximum size)
  - To measure the quantity and the quality of these gadgets
- Results
  - A big data set
  - The larger the binary is, the larger the number of distinct gadgets is
  - The transformations of the compiler has a high impact on the gadgets
  - The language/compiler can (probably) be inferred based on the gadgets found
  - A small change in the build environment cause a large change among the gadgets $\Rightarrow$ an existing CFH chain cannot be reused

- How to debug an obfuscated binary?
  - It is hard to understand such a binary
  - The bug may come from the source, any pass or a flow of passes

$\Rightarrow$ Trace the transformations during the compilation process

  - *Backward traceability*: from the binary to the source code
  - *Forward traceability*: from the source code to the binary
- Create a traceability framework
  - LLVM
  - but generic
- Current state, a prototype of
  - A tracer collecting events
  - The serialization of such a trace
  - A tool to query a trace

His presentation, later in 2023