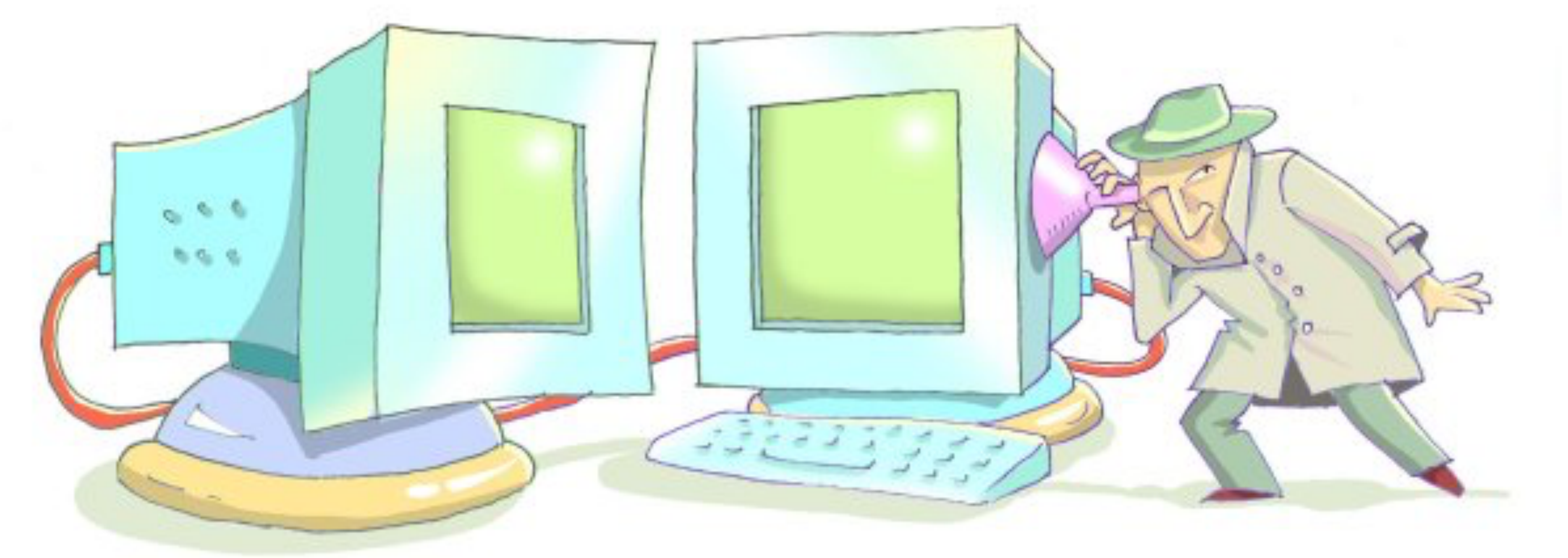
Formal Verification of a Constant-Time Preserving C Compiler

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin,
Vincent Laporte, David Pichardie and Alix Trieu

Cache timing attacks against cryptographic implementations

- Common side-channel: cache timing attacks
- Exploit the latency between cache hits and misses
- Attackers can recover cryptographic keys
  - Tromer et al (2010), Gullasch et al (2011) show efficient attacks on AES implementations
- Based on the use of look-up tables
  - Access to memory addresses that depend on the key



Constant-time programming

A programming discipline for crypto programmers

- Constant-time programs should not
  - branch on secrets
  - perform memory accesses that depend on secrets

```
if (secret)
then do1()
else do2()
```

not constant-time

- This is a strictly stronger property than « time execution does not depend on secrets » !
- There are constant-time implementations of many cryptographic algorithms: AES, DES, RSA, etc

```
a[secret] = …
```

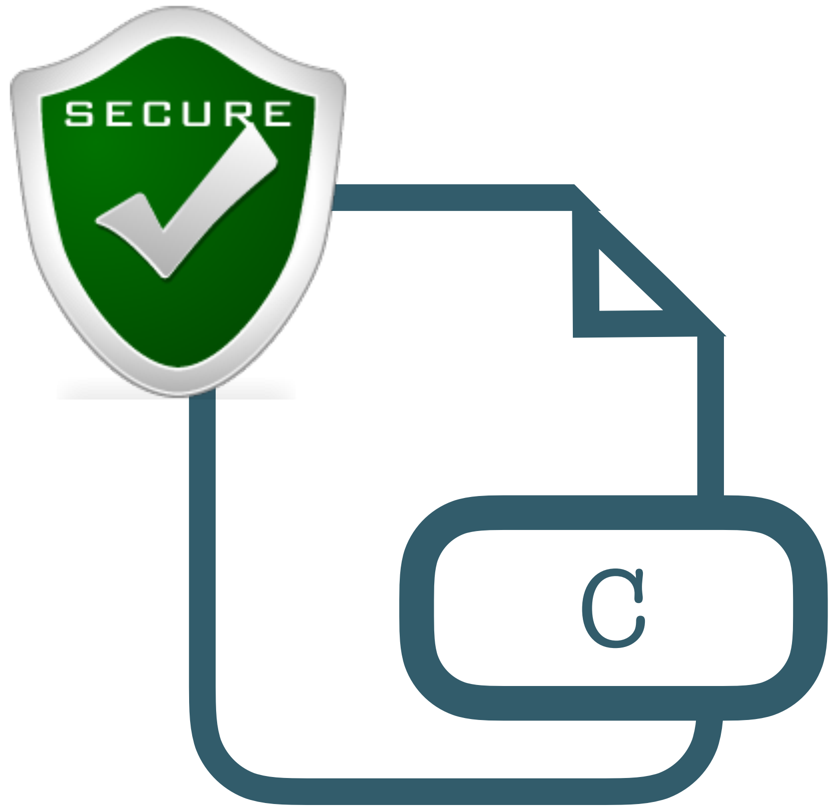
not constant-time

Cryptographic constant-time verification

- Several verification tools have been built and used for checking that popular libraries are constant-time [Almeida16, Rodrigues16]
- But checking low-level implementations is not ideal
  - it makes the analysis work harder (e.g. alias analysis)
  - it makes the results of the analysis difficult to understand for programmers

Our Research Program

Our Research Program

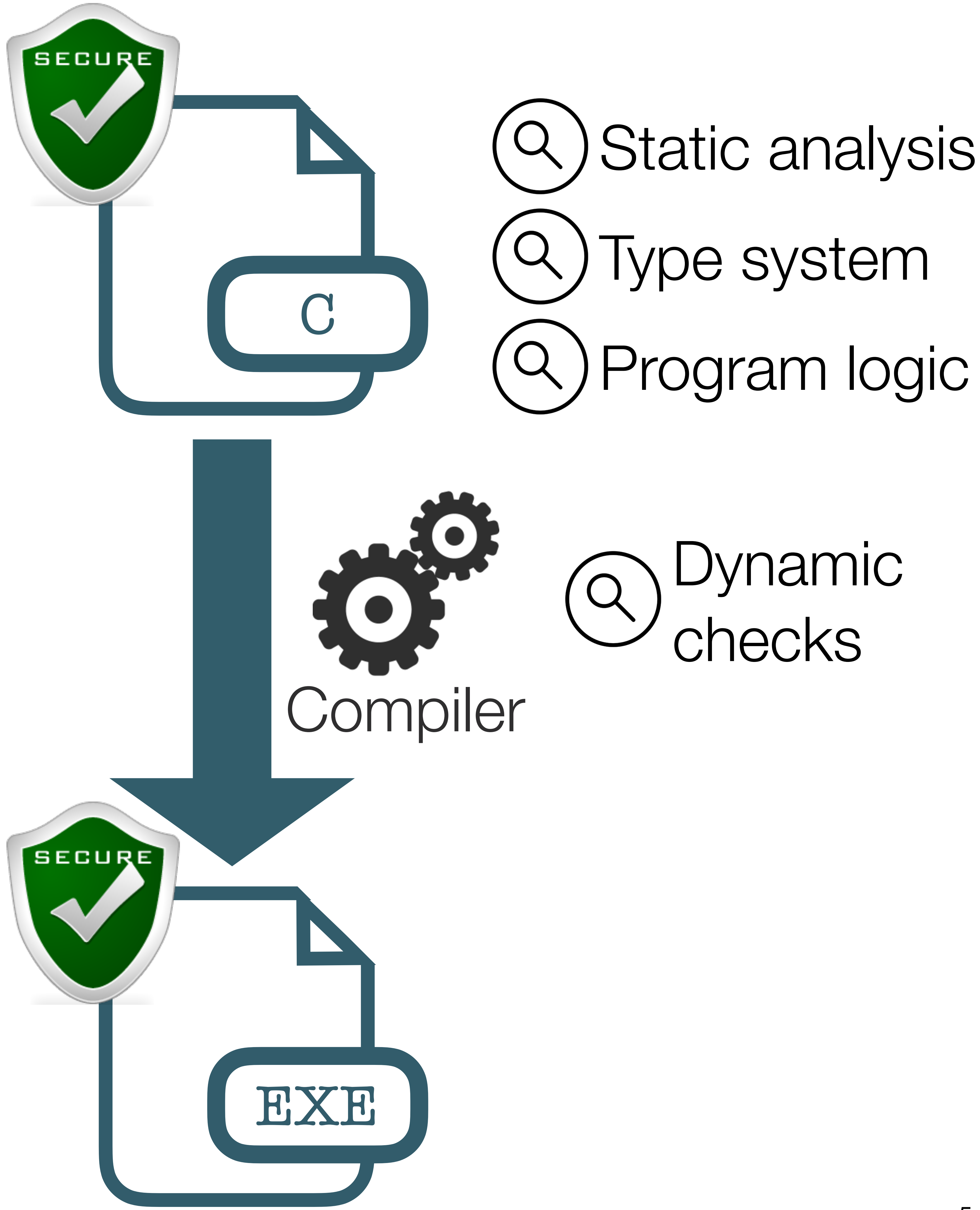


- Static analysis
- Type system
- Program logic

- Build secure programming abstractions at source level (C-like)

Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure



Static analysis

Type system

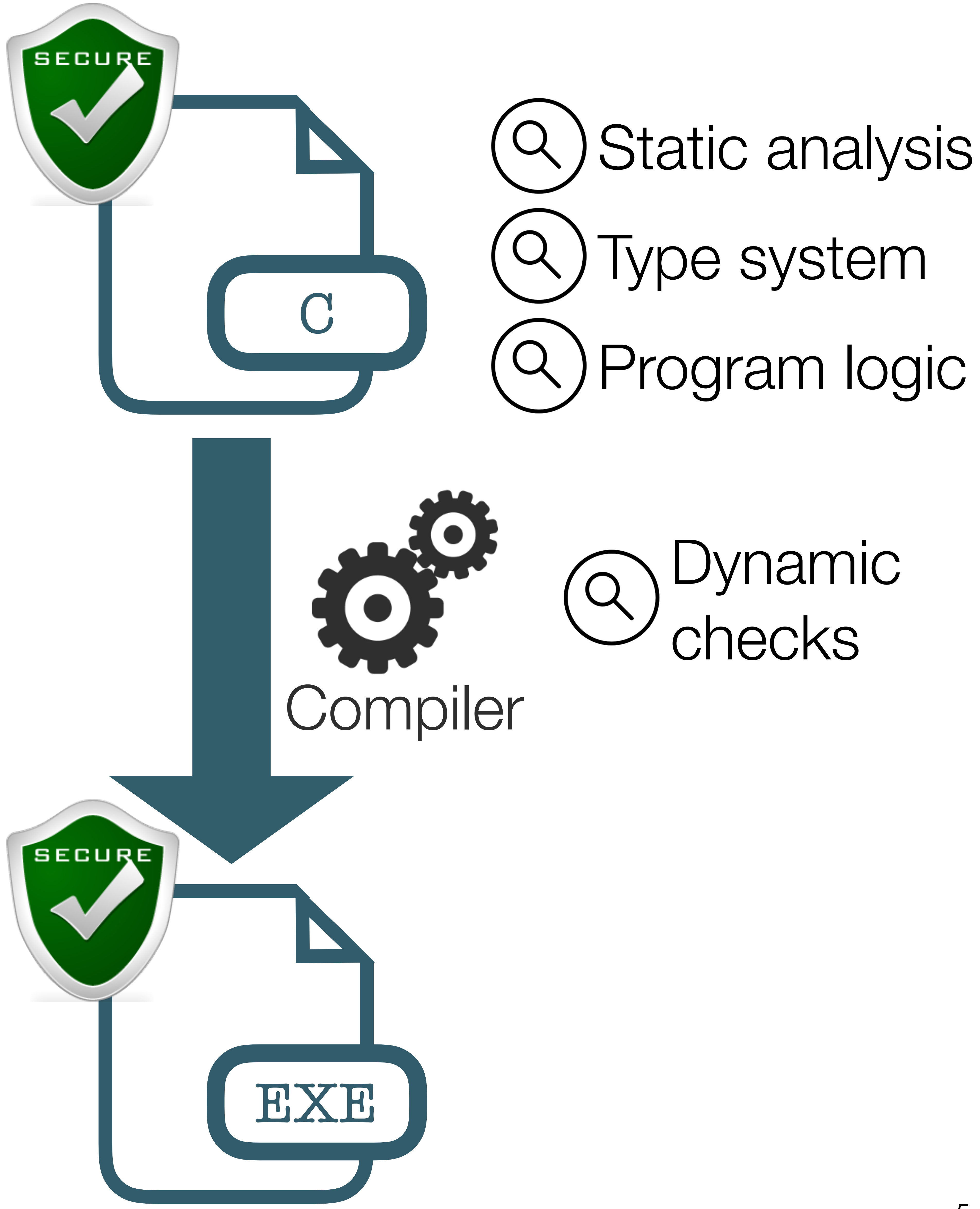
Program logic

Compiler

Dynamic checks

Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs

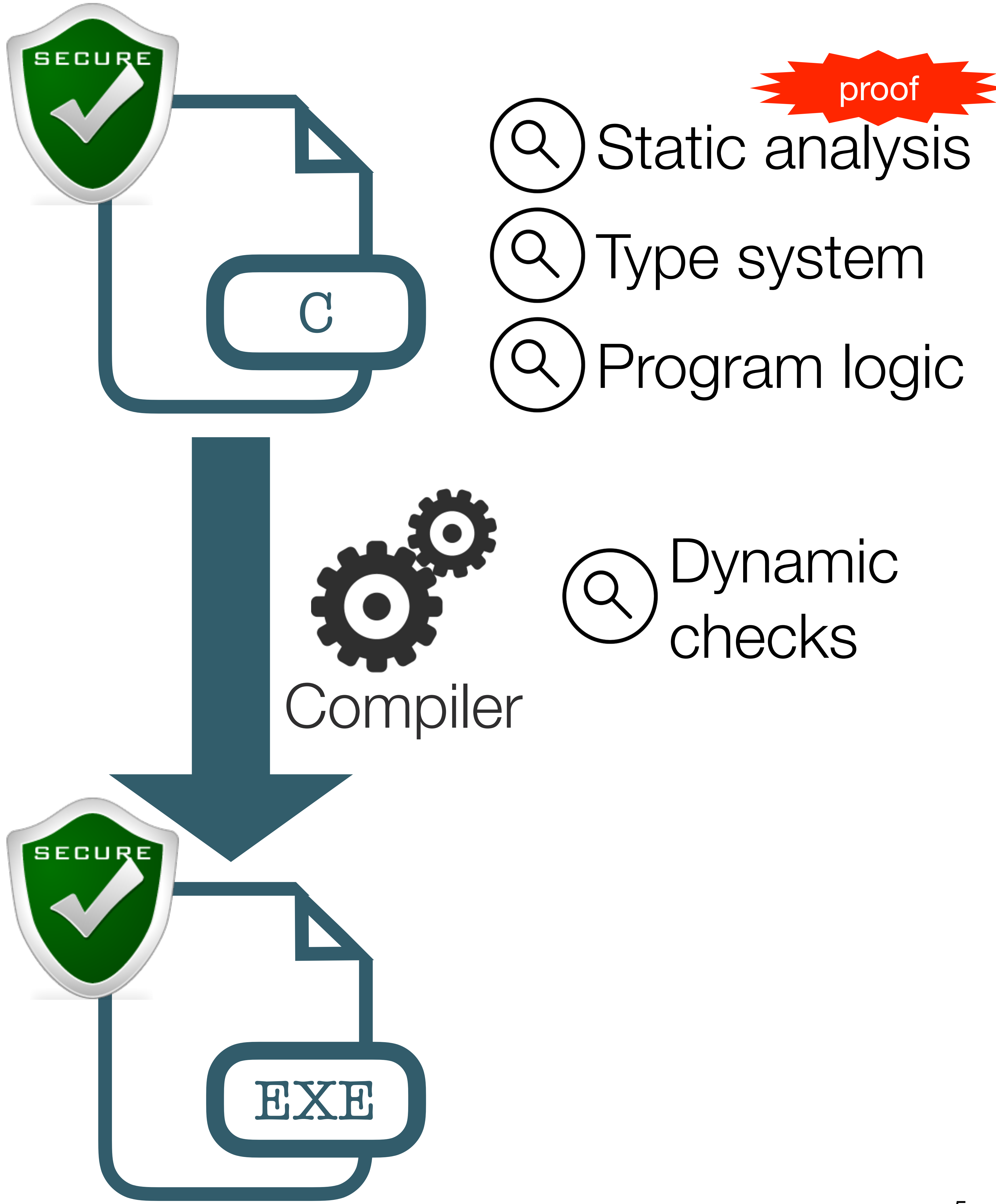


Our Research Program

- Build secure programming abstractions at source level (C-like)

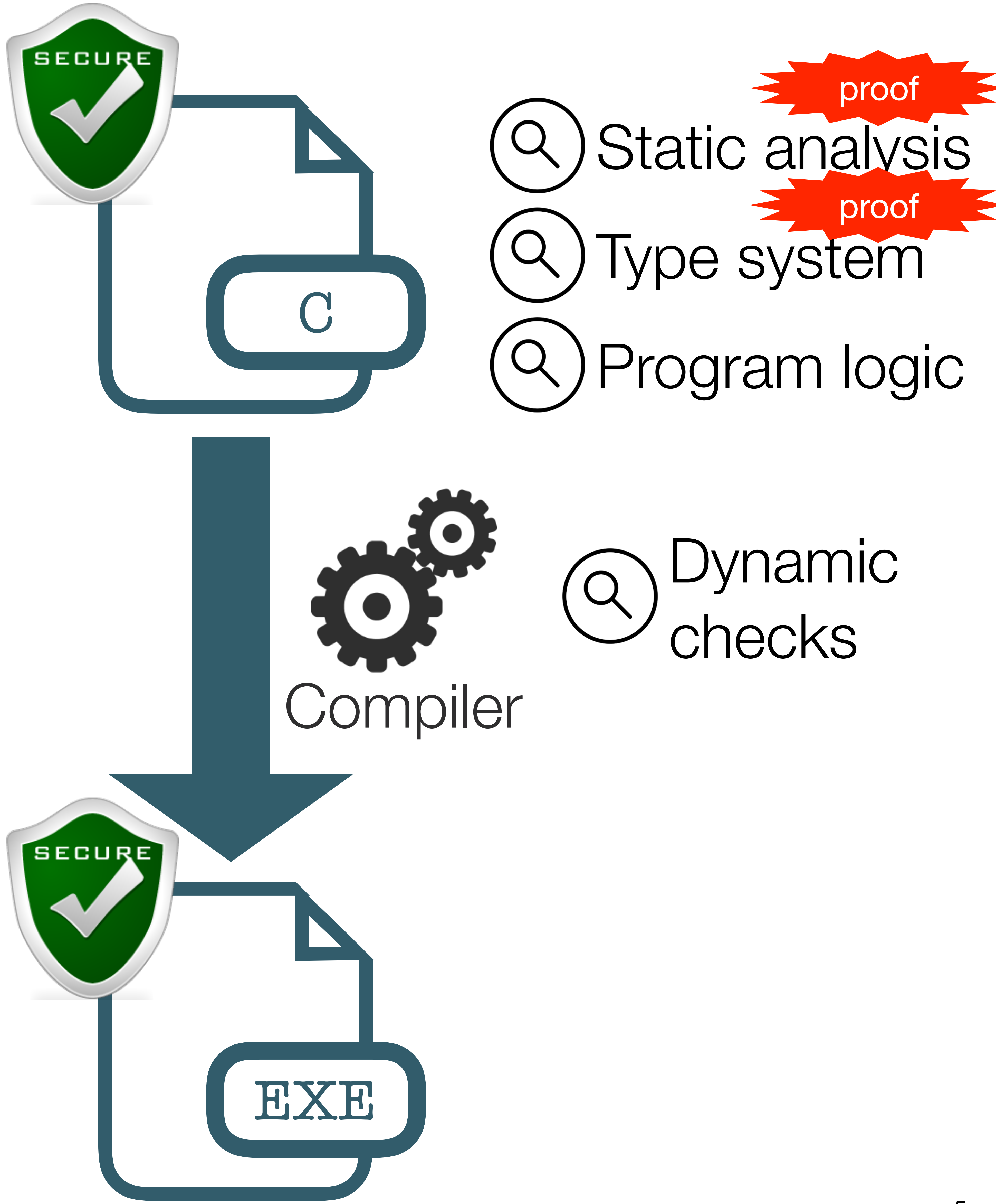
- Make sure the compiler will generate executables that are as secure

- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs



Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs



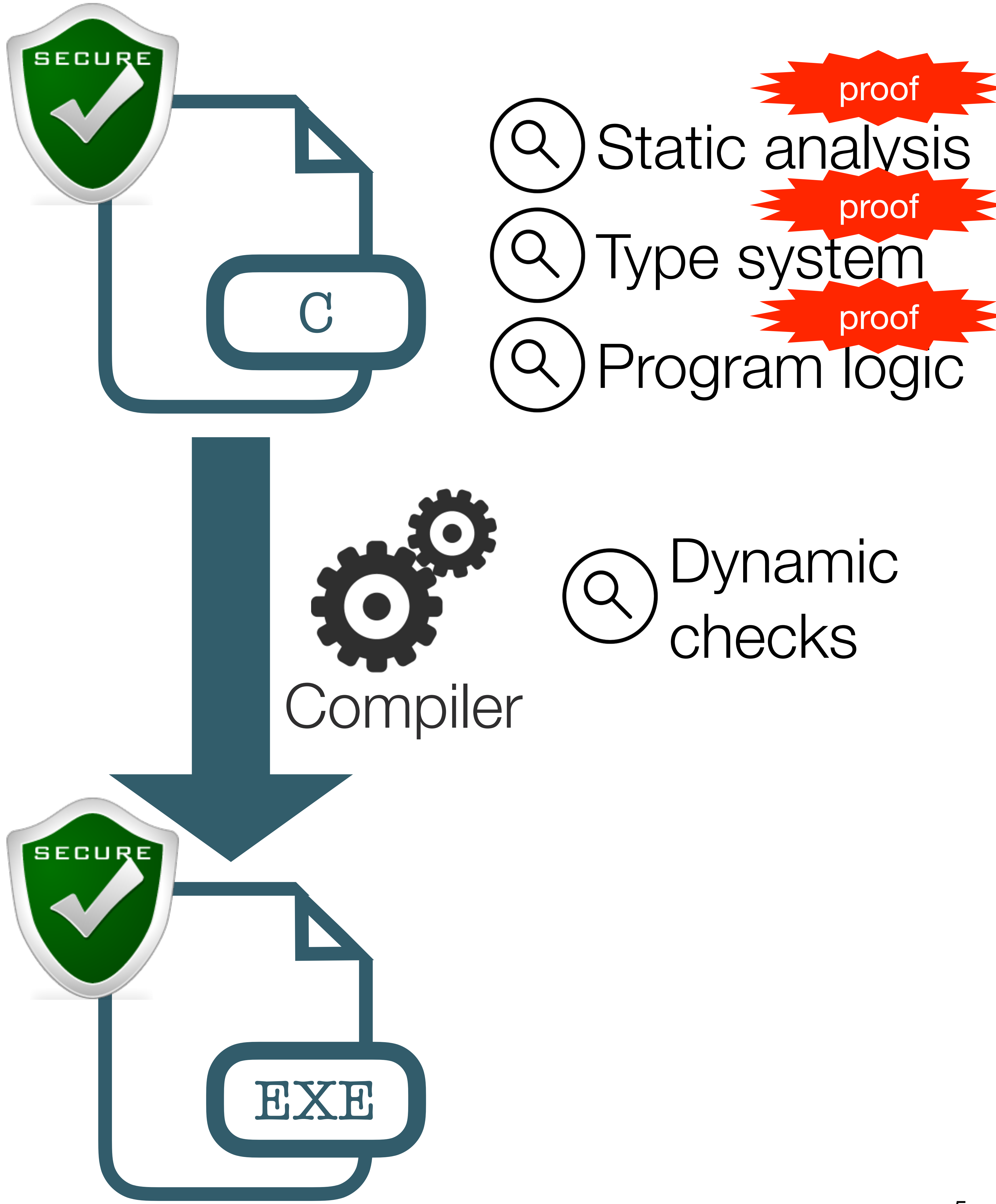
proof
Static analysis
proof
Type system
Program logic

Compiler

Dynamic checks

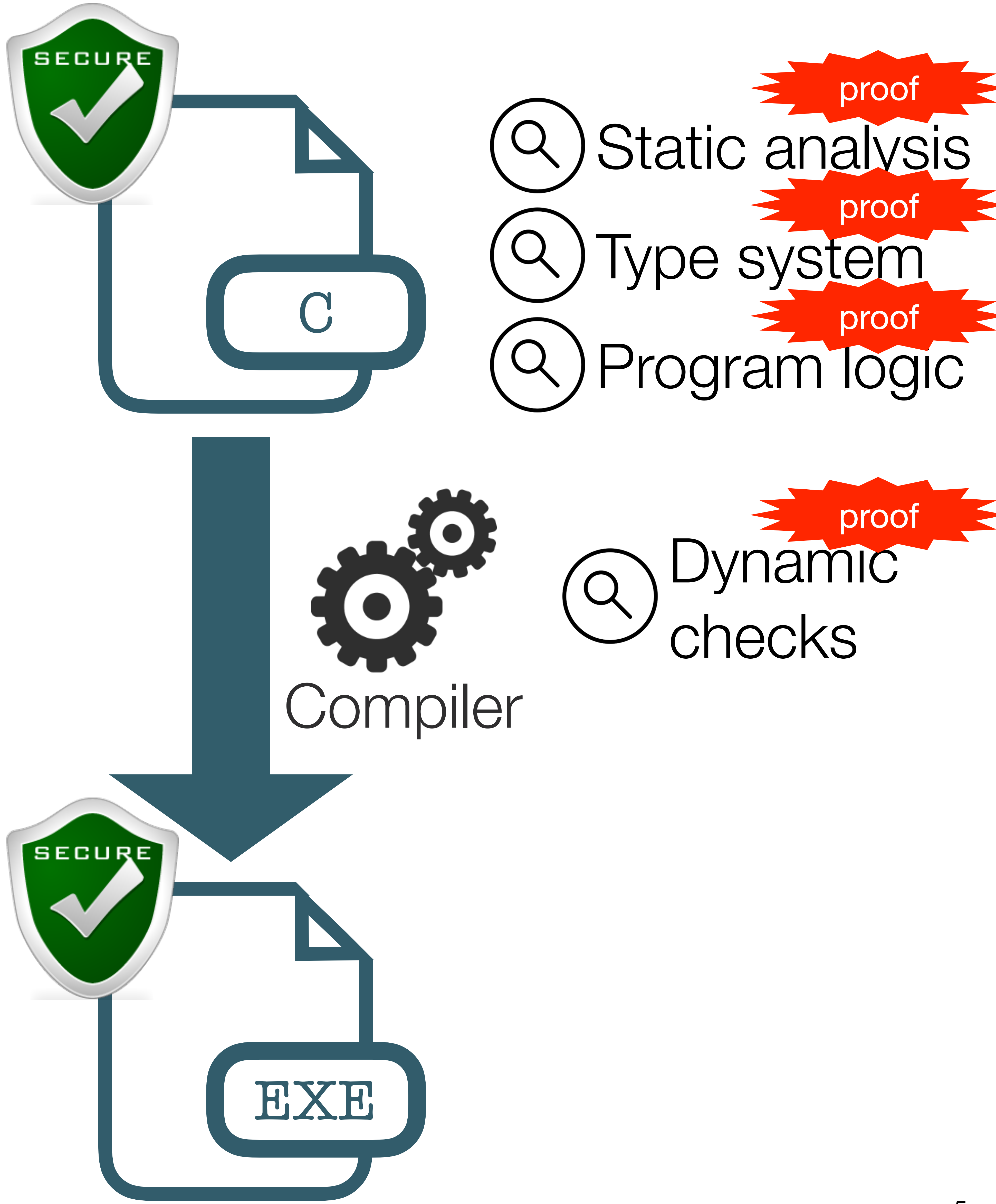
Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs



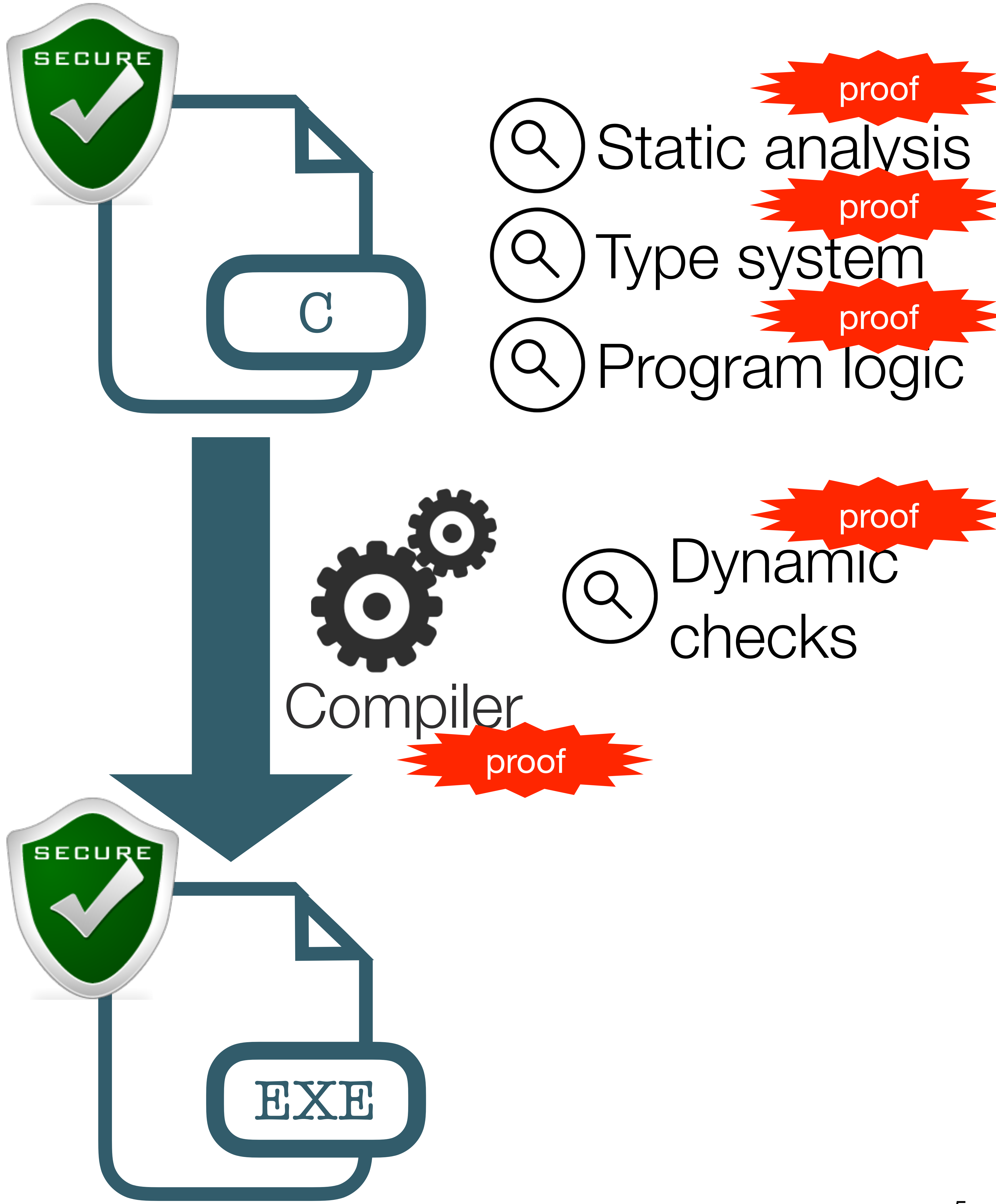
Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs



Our Research Program

- Build secure programming abstractions at source level (C-like)
- Make sure the compiler will generate executables that are as secure
- Reduce as much as possible the TCB (Trusted Computing Base) with formal proofs



Cryptographic constant-time verification

- In [ESORICS'17] we provide a verification tool at C source level
  - it tracks taints in memory and checks the constant-time property
  - it is based on the Verasco C abstract interpreter [POPL'15]

S. Blazy, D. Pichardie, A. Trieu.
Verifying Constant-Time Implementations by Abstract Interpretation.
ESORICS 2017 & Journal of Computer Security 2019.

J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie.
A formally-verified C static analyzer.
POPL'15.

- In this work [POPL'20]
  - we prove the CompCert compiler preserves the constant-time property

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, A. Trieu.
Formal verification of a constant-time preserving C compiler.
POPL'20

$$\forall p, \mathrm{ConstantTime}(p) \Rightarrow \mathrm{ConstantTime}(\mathrm{compile}(p))$$

Verified Compilation

- Proving semantic properties on non-toy compilers requires a machine-checked proof

- CompCert [Leroy06] is a milestone in this area
  - a moderately optimizing compiler for C
  - programmed and verified with the Coq proof assistant
  - now being used in commercial settings and for software certification [Kästner18]

- CompCert theorems show
  - it preserves memory safety ✓
  - it preserves observable behaviors ✓
  - but they says nothing about side channels attacks ?

This work

- Makes precise what secure compilation means for cryptographic constant-time
- Provides a machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
- Explains how to turn a pre-exisiting formally-verified compiler into a formally-verified secure compiler
- Provides a proof toolkit for proving security preservation with simulation diagrams

Some background on CompCert

Background: verifying a compiler

CompCert, a moderately optimizing C compiler usable for critical embedded software

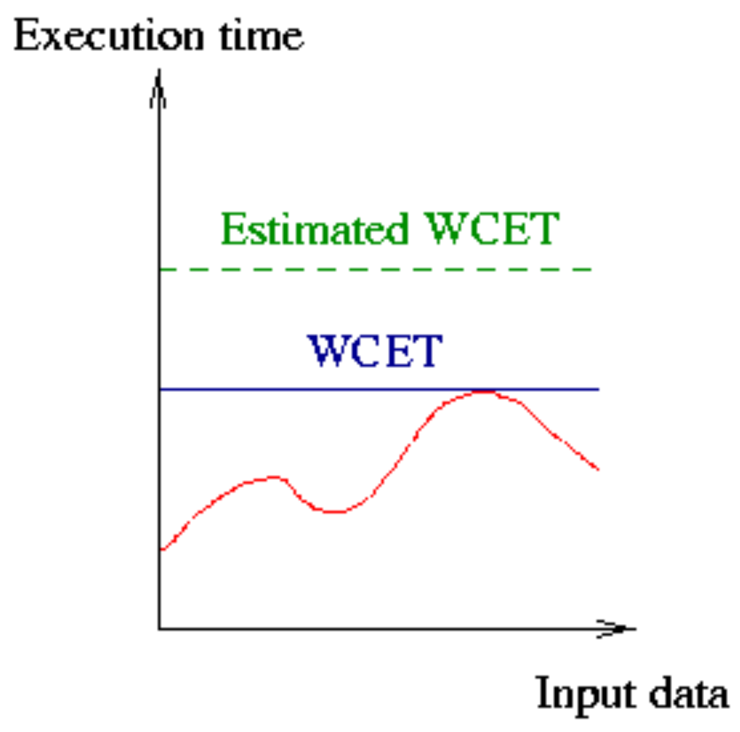
= compiler + proof that the compiler does not introduce bugs

Using the Coq proof assistant, X. Leroy proves the following semantic preservation property:

> For all source programs S and compiler-generated code C, if
> the compiler generates machine code C from source S,
> without reporting a compilation error,
> then «C behaves like S».

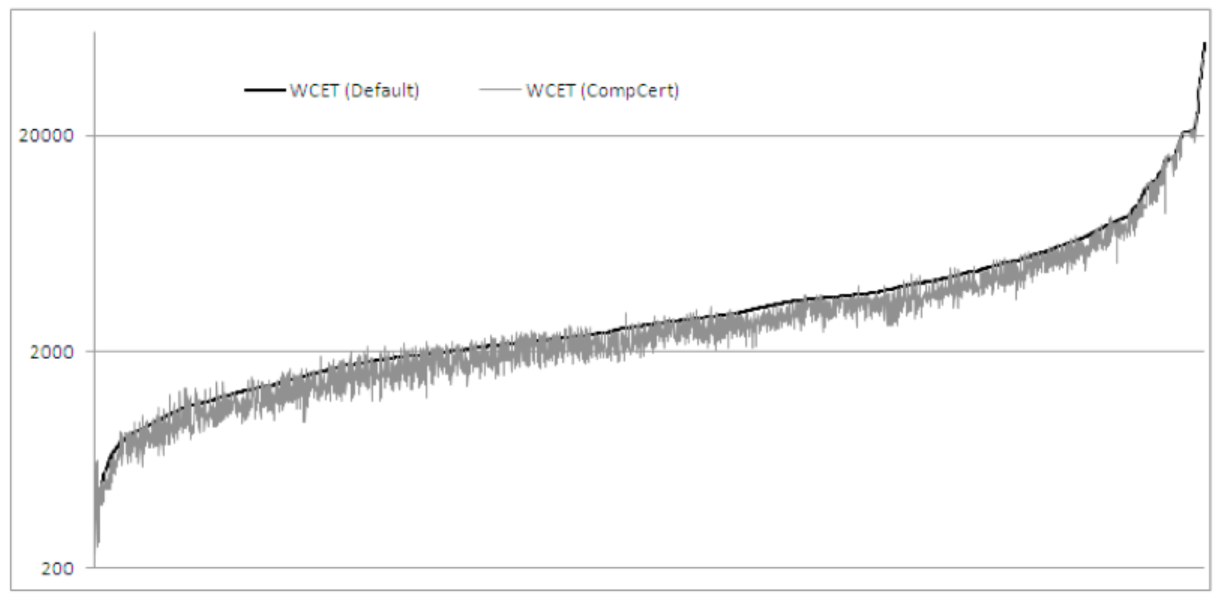
Compiler written from scratch, along with its proof; not trying to prove an existing compiler

Compcert meets the industrial world



Fly-by-wire software, for recent Airbus planes

- control-command code generated from block diagrams (3600 files, 3.96 MB of assembly code)
- minimalistic OS



Results

- Estimated WCET for each file
- Average improvement per file: 14%
- Compiled with CompCert 2.3, May 2014

Conformance to the certification process (DO-178)

- Trade-off between traceability guarantees and efficiency of the generated code

Co rld

Fly-by-wire softwar

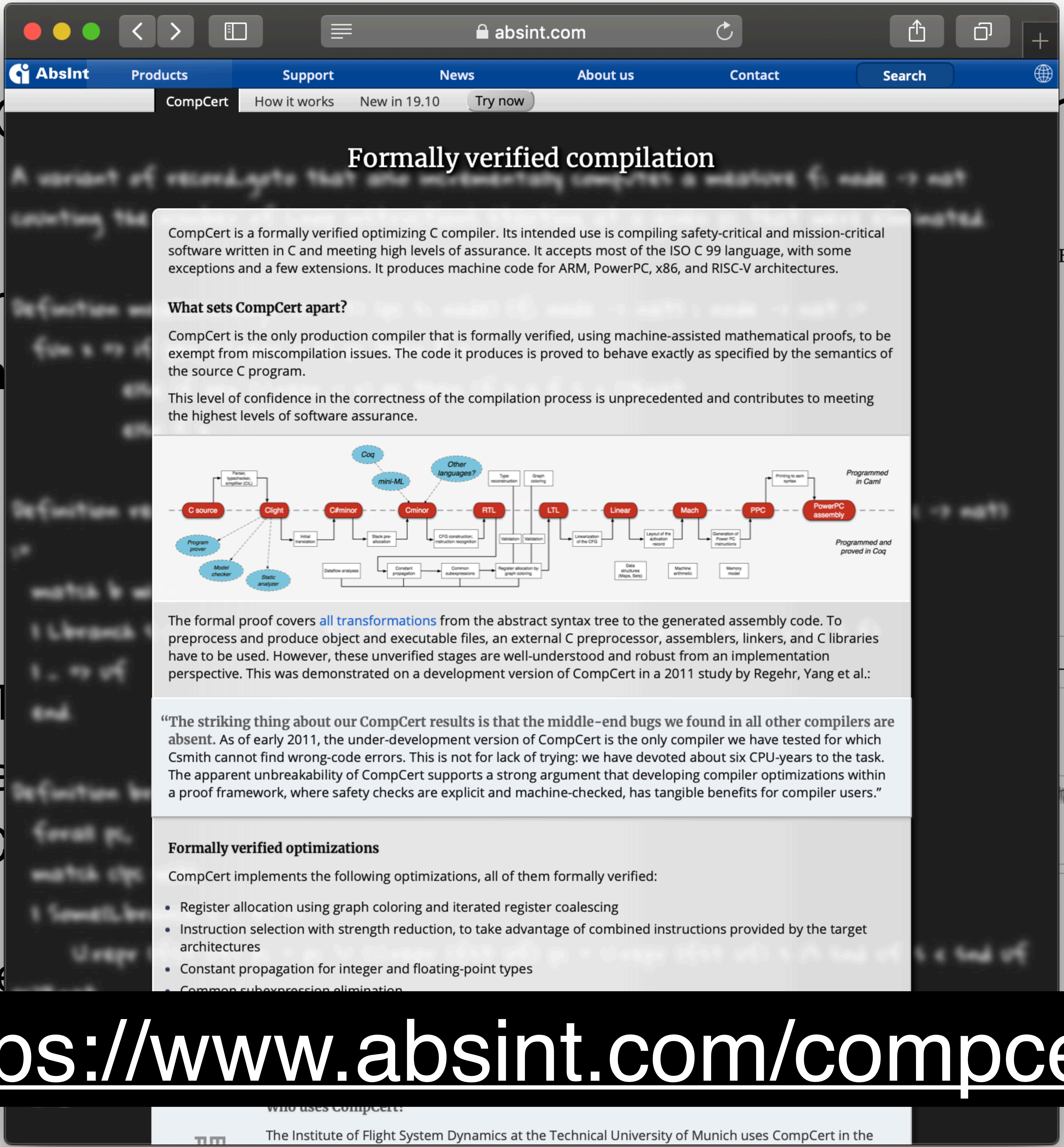
- control-comman
  (3600 files, 3.96
- minimalistic OS

Results

- Estimated WCET
- Average improve
- Compiled with C

Conformance to the

- Trade-off b ted code



Formally verified compilation

CompCert is a formally verified optimizing C compiler. Its intended use is compiling safety-critical and mission-critical software written in C and meeting high levels of assurance. It accepts most of the ISO C 99 language, with some exceptions and a few extensions. It produces machine code for ARM, PowerPC, x86, and RISC-V architectures.

What sets CompCert apart?

CompCert is the only production compiler that is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. The code it produces is proved to behave exactly as specified by the semantics of the source C program.

This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance.

The formal proof covers all transformations from the abstract syntax tree to the generated assembly code. To preprocess and produce object and executable files, an external C preprocessor, assemblers, linkers, and C libraries have to be used. However, these unverified stages are well-understood and robust from an implementation perspective. This was demonstrated on a development version of CompCert in a 2011 study by Regehr, Yang et al.:

> "The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."

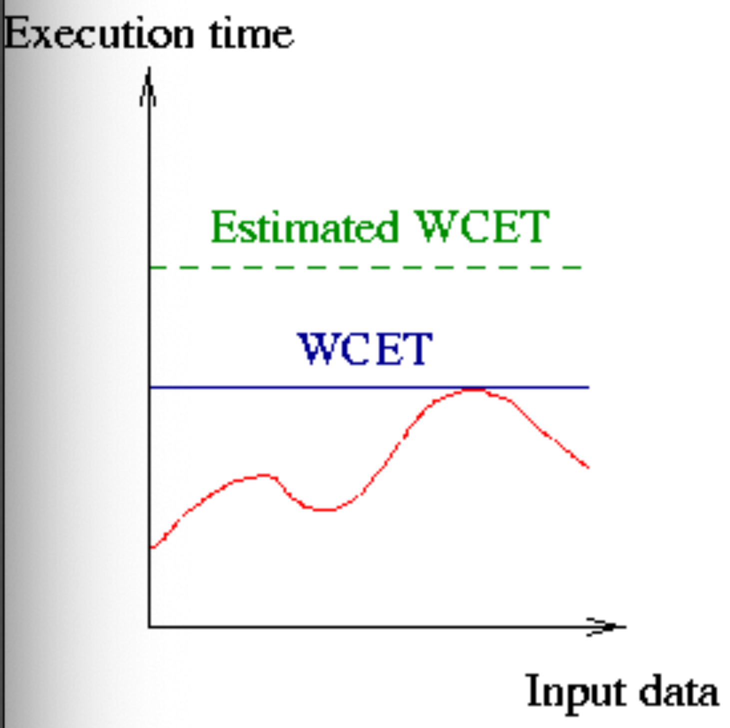
Formally verified optimizations

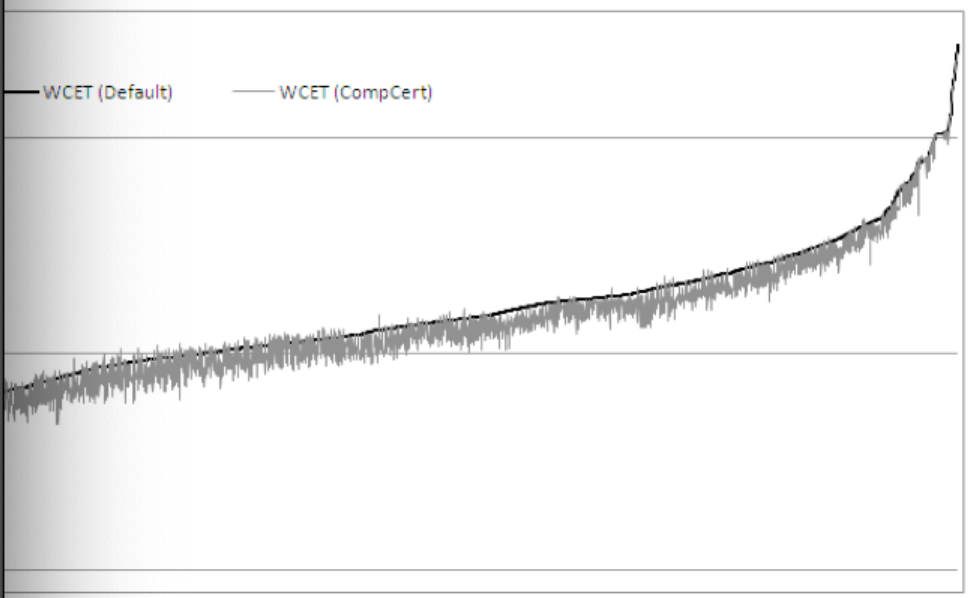
CompCert implements the following optimizations, all of them formally verified:

- Register allocation using graph coloring and iterated register coalescing
- Instruction selection with strength reduction, to take advantage of combined instructions provided by the target architectures
- Constant propagation for integer and floating-point types
- Common subexpression elimination

Who uses CompCert?

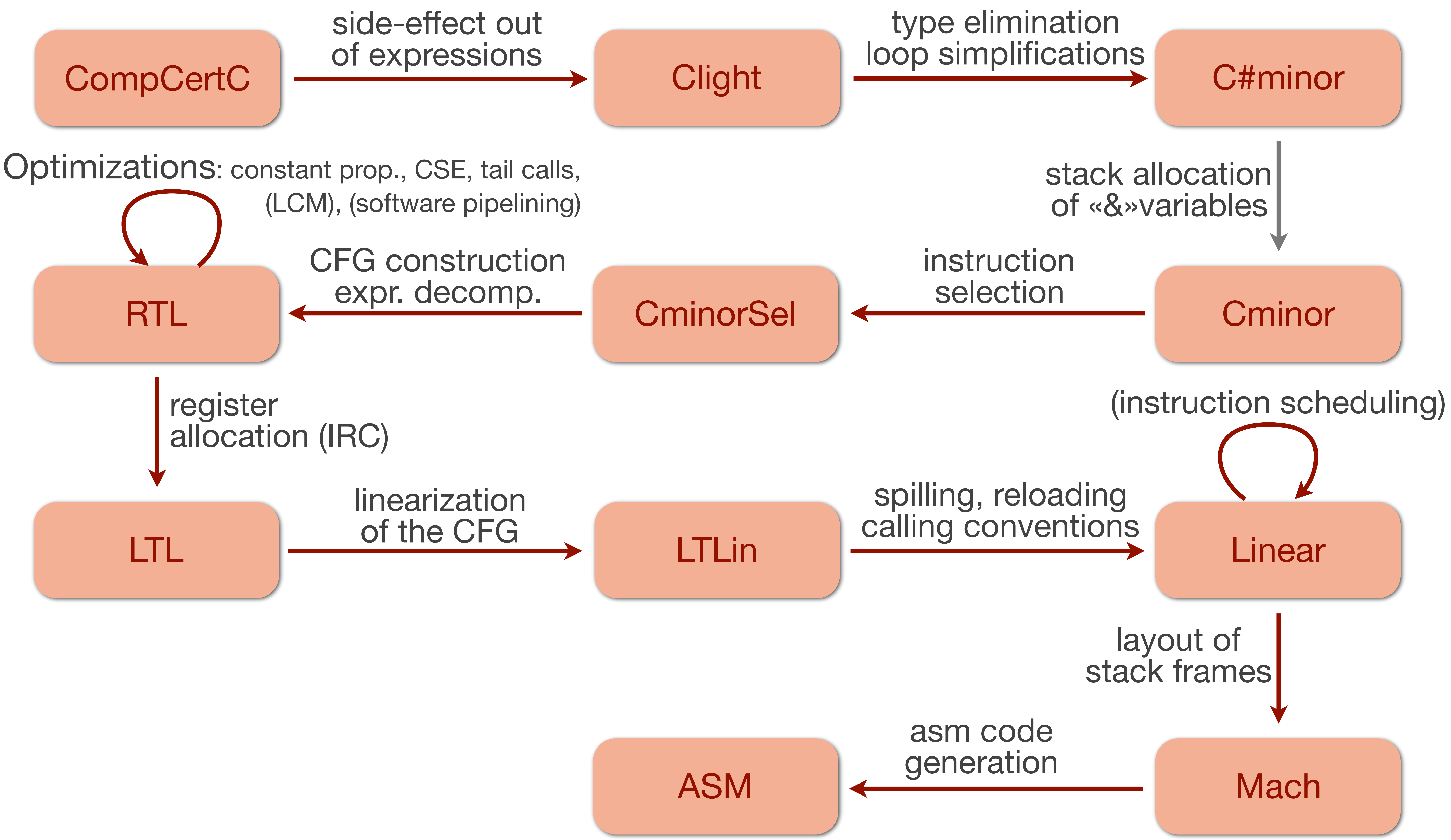
The Institute of Flight System Dynamics at the Technical University of Munich uses CompCert in the





https://www.absint.com/compcert/

CompCert: 1 compiler, 11 languages…



CompCertC

side-effect out of expressions

Clight

type elimination
loop simplifications

C#minor

stack allocation of «&»variables

Cminor

instruction selection

CminorSel

CFG construction
expr. decomp.

RTL

Optimizations: constant prop., CSE, tail calls, (LCM), (software pipelining)

register allocation (IRC)

LTL

linearization of the CFG

LTLin

spilling, reloading
calling conventions

Linear

(instruction scheduling)

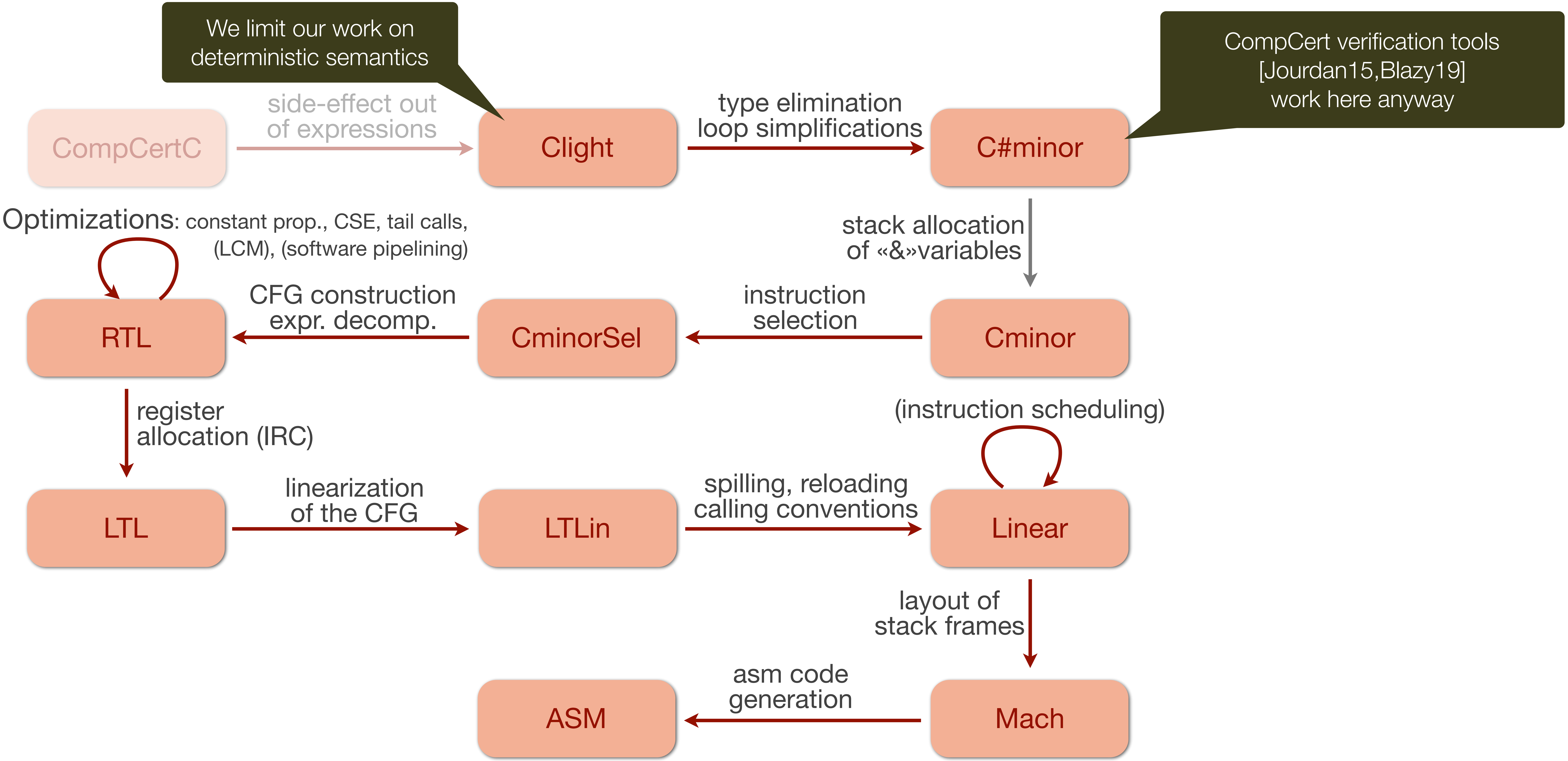
layout of stack frames

Mach

asm code generation

ASM

CompCert: 1 compiler, 11 languages…



We limit our work on deterministic semantics

CompCert verification tools [Jourdan15,Blazy19] work here anyway

CompCertC

side-effect out of expressions

Clight

type elimination loop simplifications

C#minor

stack allocation of «&»variables

Cminor

instruction selection

CminorSel

CFG construction expr. decomp.

RTL

Optimizations: constant prop., CSE, tail calls, (LCM), (software pipelining)

register allocation (IRC)

LTL

linearization of the CFG

LTLin

spilling, reloading calling conventions

Linear

(instruction scheduling)

layout of stack frames

Mach

asm code generation

ASM

CompCert: … and 17 preservations proofs

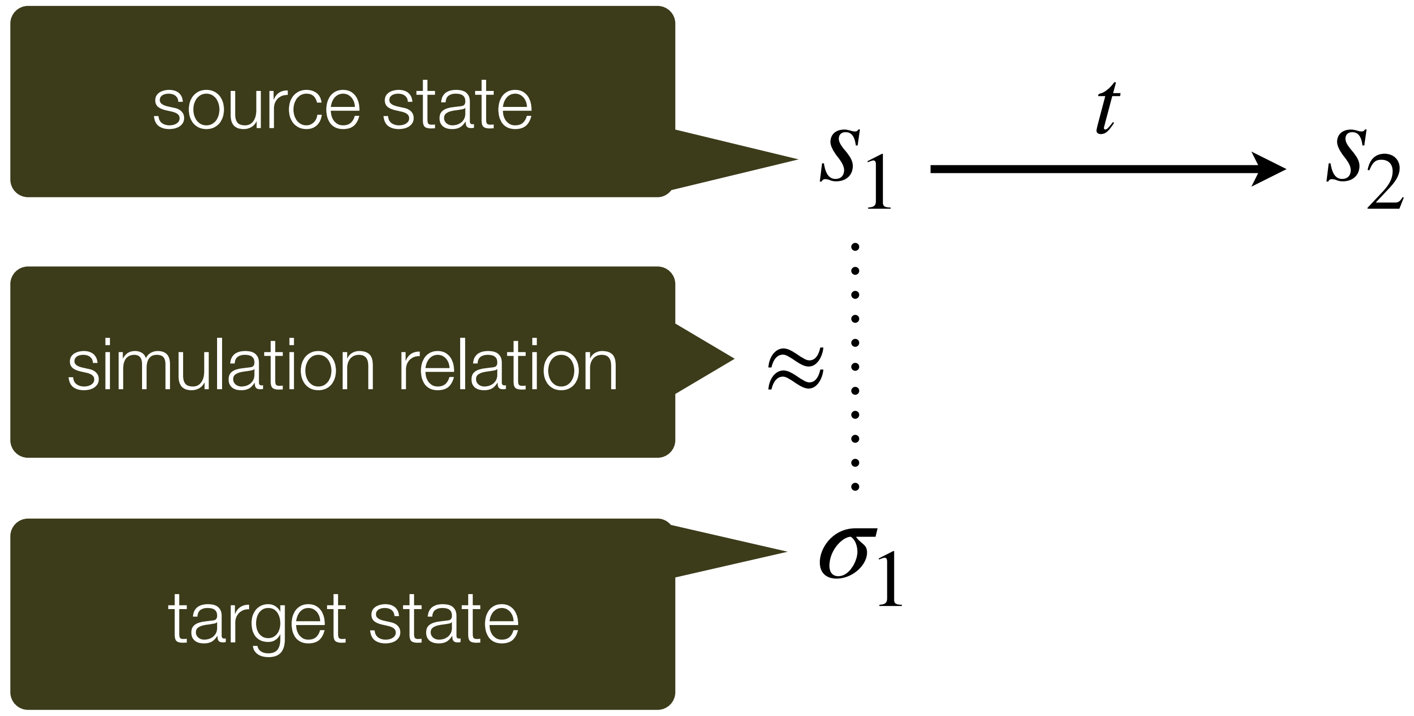
Compiler pass	Explanation on the pass
Cshmgen	Type elaboration, simplification of control
Cminorgen	Stack allocation
Selection	Recognition of operators and addr. modes
RTLgen	Generation of CFG and 3-address code
Tailcall	Tailcall recognition
Inlining	Function inlining
Renumber	Renumbering CFG nodes
ConstProp	Constant propagation
CSE	Common subexpression elimination
Deadcode	Redundancy elimination
Allocation	Register allocation
Tunneling	Branch tunneling
Linearize	Linearization of CFG
CleanupLabels	Removal of unreferenced labels
Debugvar	Synthesis of debugging information
Stacking	Laying out stack frames
Asmgen	Emission of assembly code

CompCert preservation proof methodology

- Each langage is given an **operational semantics $s \xrightarrow{t} s'$** that models a small step transition from a state s to a state s' by emitting a trace of external events t.
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.

CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



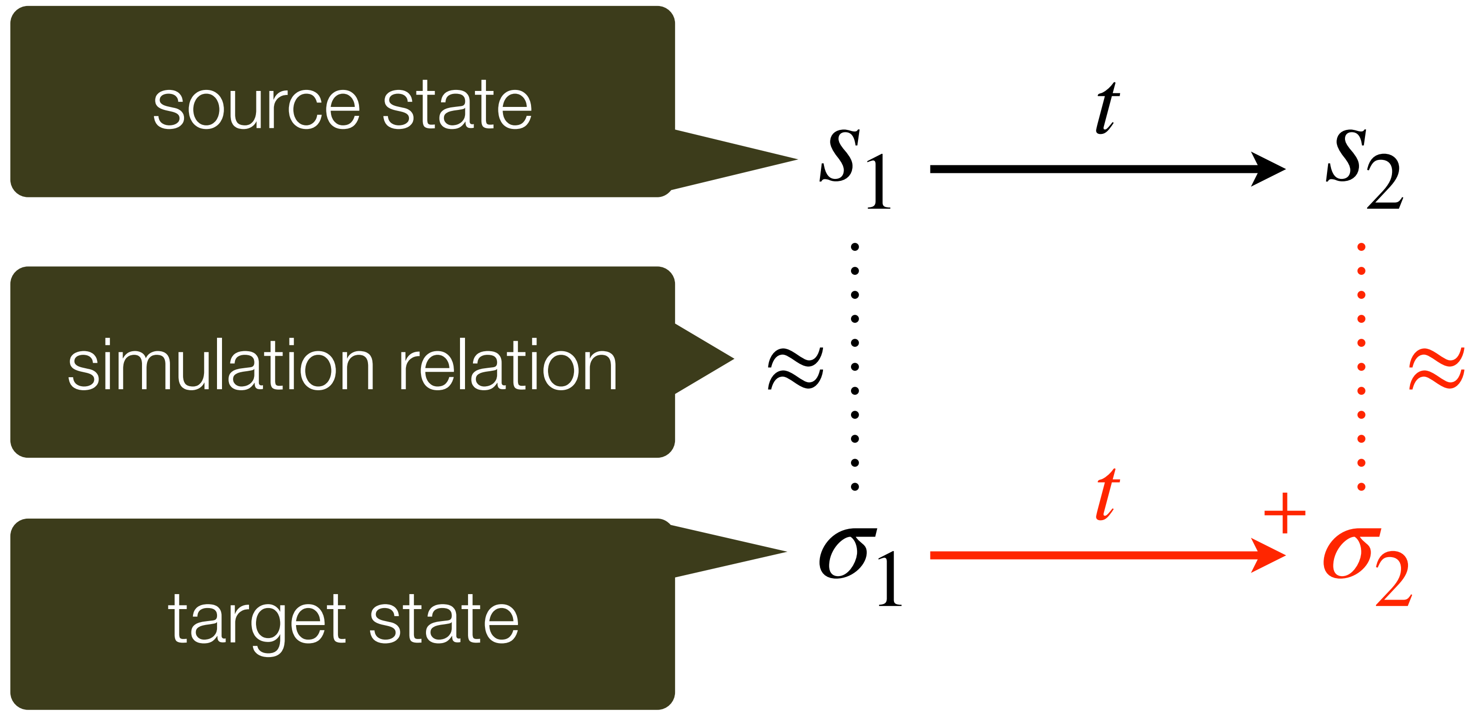
source state

simulation relation

target state

CompCert preservation proof methodology

- Each langage is given an **operational semantics $s \xrightarrow{t} s'$** that models a small step transition from a state s to a state s' by emitting a trace of external events t.
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



source state

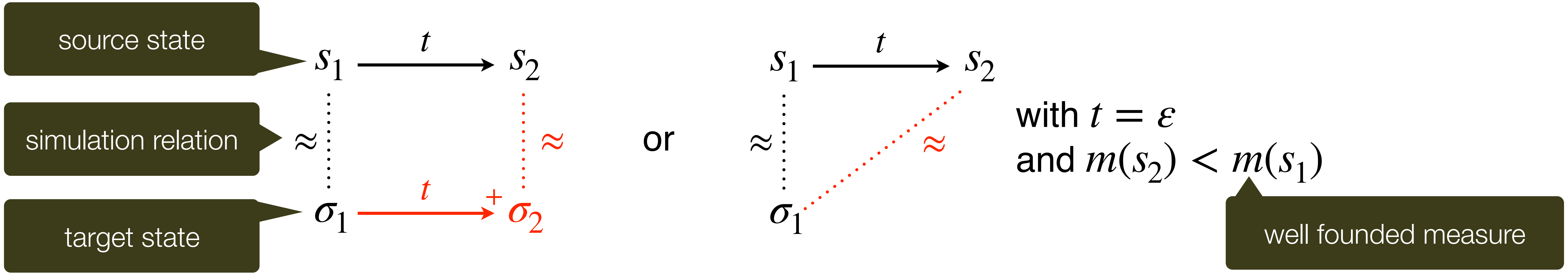
simulation relation

target state

$$\begin{tikzcd} s_1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s_2 \arrow[d, dash, dotted, "\approx"] \\ \sigma_1 \arrow[r, "t"] & {}^{+}\sigma_2 \end{tikzcd}$$

CompCert preservation proof methodology

- Each langage is given an **operational semantics** $s \xrightarrow{t} s'$ that models a small step transition from a state s to a state s' by emitting a trace of external events t.
- From this stems a notion of **program behavior** (event trace) for complete (possibly infinite) executions.
- Behavior preservation is proved via backward and forward simulation, but thanks to langage determinism, **forward simulation** is enough.



source state

simulation relation

target state

$$\begin{tikzcd} s_1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s_2 \arrow[d, dash, dotted, "\approx"] \\ \sigma_1 \arrow[r, "t", "+"{near end}] & \sigma_2 \end{tikzcd}$$

or

$$\begin{tikzcd} s_1 \arrow[r, "t"] \arrow[d, dash, dotted, "\approx"'] & s_2 \\ \sigma_1 \arrow[ur, dash, dotted, "\approx"'] & \end{tikzcd}$$

with $t = \varepsilon$
and $m(s_2) < m(s_1)$

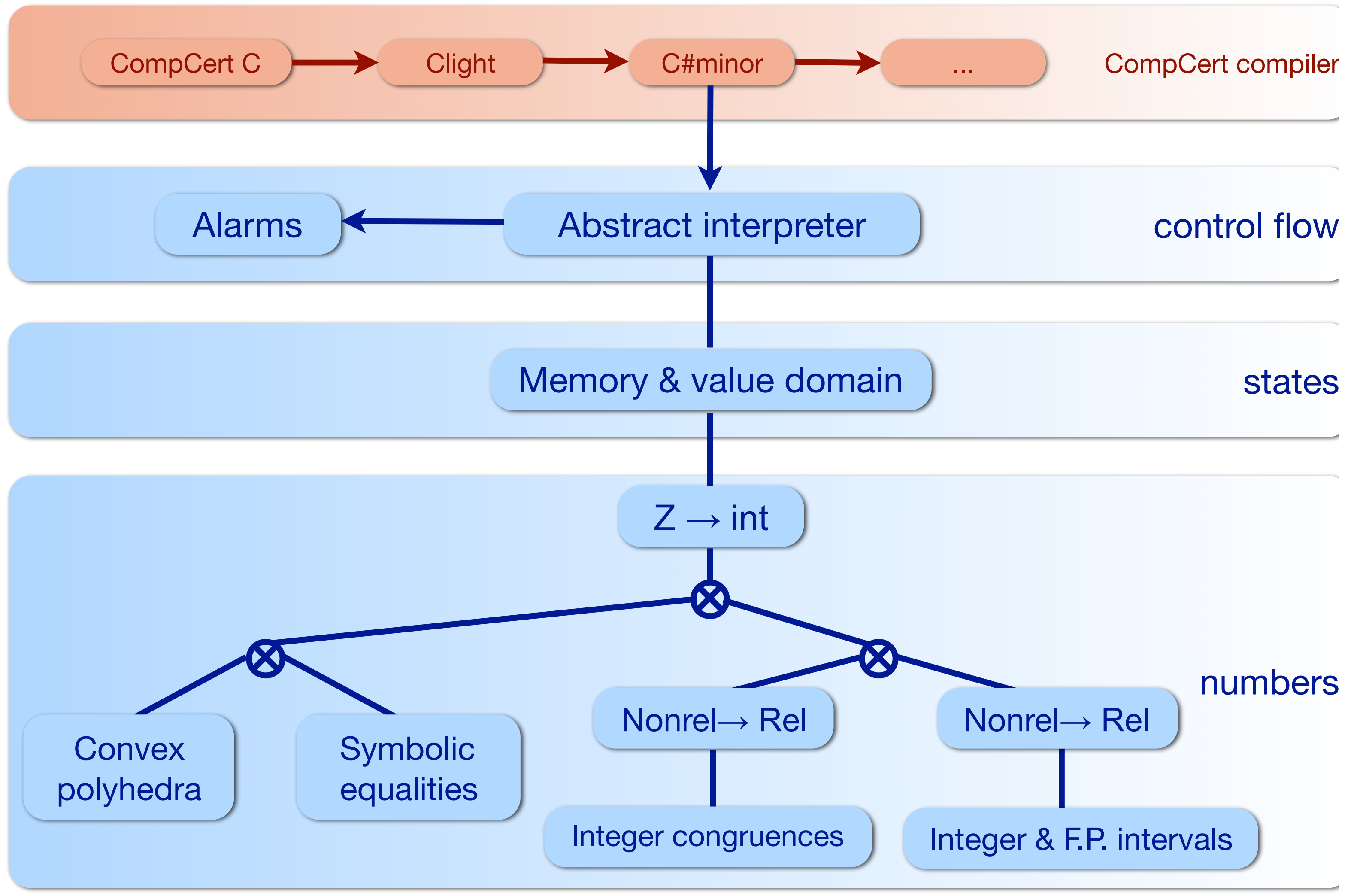
well founded measure

Verified Static Analysis meets CompCert

The verified C static analyzer Verasco [POPL'15]

Goal: develop and verify in Coq a realistic static analyzer by abstract interpretation

- language analyzed: the CompCert subset of C
- nontrivial abstract domains, including relational domains
- modular architecture inspired from Astrée's
- to prove the absence of undefined behaviors in C source programs



Verasco architecture

Slogan:

- if « CompCert ≈ 1/10th of GCC but formally verified »,
- likewise « Verasco ≈1/10th of Astrée but formally verified »

Defining Cryptographic Constant-Time Preservation

Cryptographic constant-time property: defining leakages

- We enrich the CompCert traces of events with **leakages** of two types
  - either the truth value of a condition,
  - or a pointer representing the address of
    - either a memory access (i.e., a load or a store)
    - or a called function
- Using **event erasure**, from $s \xrightarrow{t} s'$ we can extract
  - the compile-only judgment $s \xrightarrow{t}_{\text{comp}} s'$
  - the leak-only judgment $s \xrightarrow{t}_{\text{leak}} s'$
- **Program leakage** is defined as the behavior of the $\rightarrow_{\text{leak}}$ semantics

Cryptographic constant-time property: preservation

- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

$$\begin{tikzcd} s \arrow[d, dash, "\varphi"'] \arrow[r, "t"] & {}^{*}\text{leak} \\ s' \arrow[r, "t'"] & {}^{*}\text{leak} \end{tikzcd}$$

implies $t = t'$

Cryptographic constant-time property: preservation

- We note $\varphi(s, s')$ the fact that two initial states s and s' share the same values for public inputs, but may differ on the values of secret inputs

- A program is **constant-time secure w.r.t. φ** if for two initial states s and s' such that $\varphi(s, s')$ holds, then both leak-only executions starting from s and s' observe the same leakage

Main Theorem (Constant-Time security preservation): Let P be a safe Clight source program that is compiled into an x86 assembly program P'. If P is constant-time w.r.t. φ, then so is P'.

Proving Cryptographic Constant-Time Preservation

Proving cryptographic constant-time preservation

A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams

Proving cryptographic constant-time preservation

A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\to_{\mathrm{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\to_{\mathrm{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams

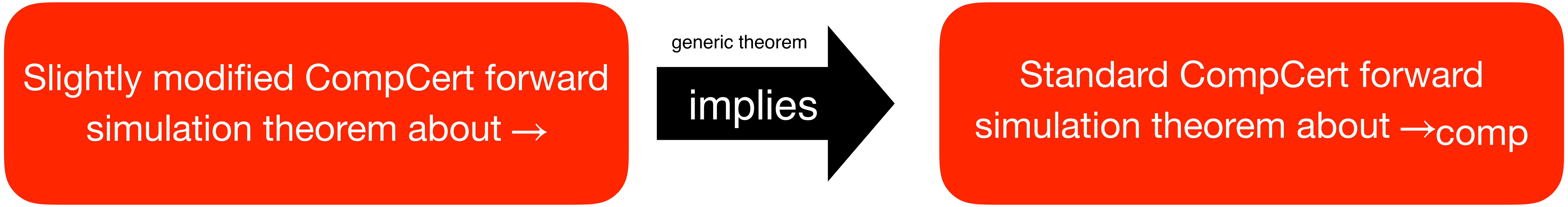
Standard CompCert forward simulation theorem about $\to_{\mathrm{comp}}$

Standard CompCert forward simulation proof script

Proving cryptographic constant-time preservation

A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams



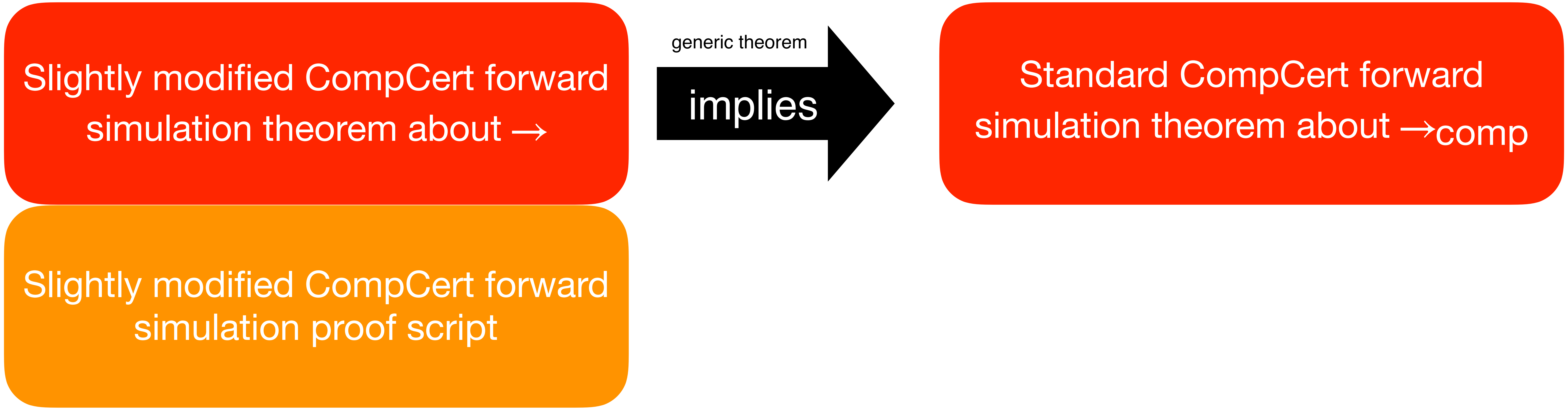
Slightly modified CompCert forward simulation theorem about $\rightarrow$

Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$

Proving cryptographic constant-time preservation

A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams



Slightly modified CompCert forward simulation theorem about $\rightarrow$

Slightly modified CompCert forward simulation proof script

generic theorem

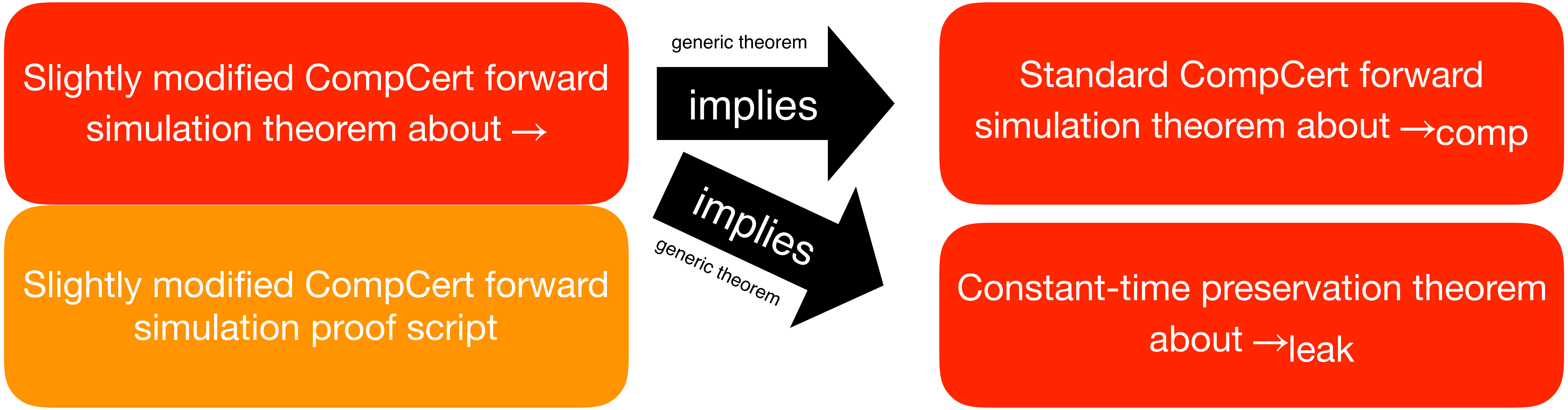
implies

Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$

Proving cryptographic constant-time preservation

A proof engineering perspective

- Cryptographic constant-time preservation is a property about the leak-only semantics $\rightarrow_{\text{leak}}$
- But existing CompCert simulation diagrams deal with the compile-only semantics $\rightarrow_{\text{comp}}$
- Our proof engineering strategy is to benefit as much as possible from the **proof scripts** of these diagrams



Slightly modified CompCert forward simulation theorem about $\rightarrow$

Slightly modified CompCert forward simulation proof script

Standard CompCert forward simulation theorem about $\rightarrow_{\text{comp}}$

Constant-time preservation theorem about $\rightarrow_{\text{leak}}$

Four proof techniques

- Each technique provides a specific tradeoff between generality and proof tractability
- The first three are slight relaxations of the classical forward diagram and reuse existing scripts

Trace preservation

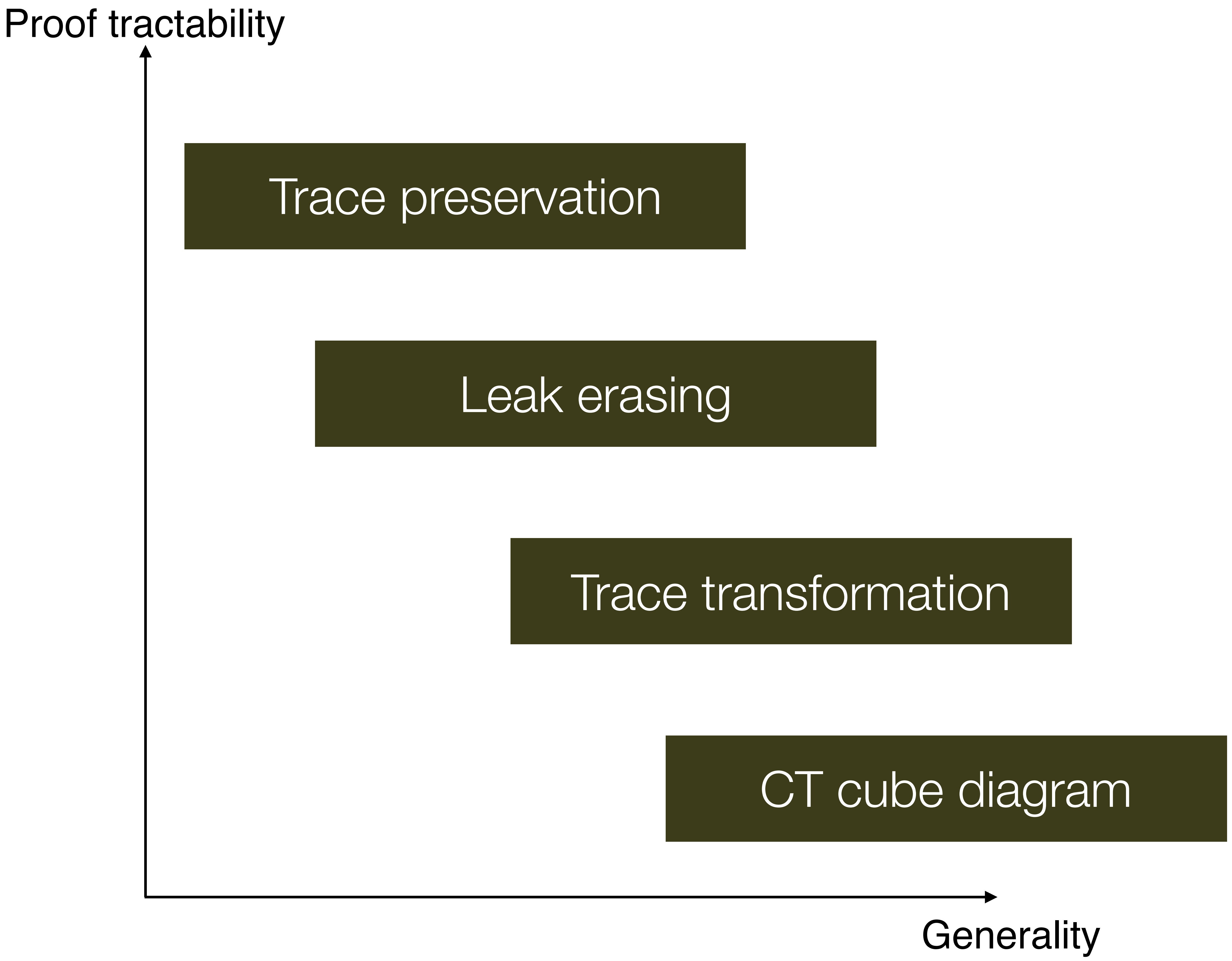
Leak erasing

Trace transformation

CT cube diagram

Four proof techniques

- Each technique provides a specific tradeoff between generality and proof tractability
- The first three are slight relaxations of the classical forward diagram and reuse existing scripts



A palette of proof methods

Trace preservation

Leak erasing

Trace transformation

CT cube diagram

Compiler pass	Diagram used	Explanation on the pass
Cshmgen		Type elaboration, simplification of control
Cminorgen		Stack allocation
Selection		Recognition of operators and addr. modes
RTLgen		Generation of CFG and 3-address code
Tailcall		Tailcall recognition
Inlining		Function inlining
Renumber		Renumbering CFG nodes
ConstProp		Constant propagation
CSE		Common subexpression elimination
Deadcode		Redundancy elimination
Allocation		Register allocation
Tunneling		Branch tunneling
Linearize		Linearization of CFG
CleanupLabels		Removal of unreferenced labels
Debugvar		Synthesis of debugging information
Stacking		Laying out stack frames
Asmgen		Emission of assembly code

A palette of proof methods

Trace preservation

Leak erasing

Trace transformation

CT cube diagram

6/17

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen		Stack allocation
Selection		Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall recognition
Inlining		Function inlining
Renumber	Trace preservation	Renumbering CFG nodes
ConstProp		Constant propagation
CSE		Common subexpression elimination
Deadcode		Redundancy elimination
Allocation		Register allocation
Tunneling		Branch tunneling
Linearize		Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking		Laying out stack frames
Asmgen		Emission of assembly code

A palette of proof methods

Trace preservation	6/17
Leak erasing	5/17
Trace transformation	
CT cube diagram	

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen		Stack allocation
Selection	Leak erasing	Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall recognition
Inlining		Function inlining
Renumber	Trace preservation	Renumbering CFG nodes
ConstProp		Constant propagation
CSE	Leak erasing	Common subexpression elimination
Deadcode	Leak erasing	Redundancy elimination
Allocation	Leak erasing	Register allocation
Tunneling	Leak erasing	Branch tunneling
Linearize		Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking		Laying out stack frames
Asmgen		Emission of assembly code

A palette of proof methods

Trace preservation 6/17

Leak erasing 5/17

Trace transformation 5/17

CT cube diagram

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen	Trace transformation	Stack allocation
Selection	Leak erasing	Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall recognition
Inlining	Trace transformation	Function inlining
Renumber	Trace preservation	Renumbering CFG nodes
ConstProp	Trace transformation	Constant propagation
CSE	Leak erasing	Common subexpression elimination
Deadcode	Leak erasing	Redundancy elimination
Allocation	Leak erasing	Register allocation
Tunneling	Leak erasing	Branch tunneling
Linearize		Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking	Trace transformation	Laying out stack frames
Asmgen	Trace transformation	Emission of assembly code

A palette of proof methods

Trace preservation	6/17
Leak erasing	5/17
Trace transformation	5/17
CT cube diagram	1/17

Compiler pass	Diagram used	Explanation on the pass
Cshmgen	Trace preservation	Type elaboration, simplification of control
Cminorgen	Trace transformation	Stack allocation
Selection	Leak erasing	Recognition of operators and addr. modes
RTLgen	Trace preservation	Generation of CFG and 3-address code
Tailcall	Trace preservation	Tailcall recognition
Inlining	Trace transformation	Function inlining
Renumber	Trace preservation	Renumbering CFG nodes
ConstProp	Trace transformation	Constant propagation
CSE	Leak erasing	Common subexpression elimination
Deadcode	Leak erasing	Redundancy elimination
Allocation	Leak erasing	Register allocation
Tunneling	Leak erasing	Branch tunneling
Linearize	CT cube diagram	Linearization of CFG
CleanupLabels	Trace preservation	Removal of unreferenced labels
Debugvar	Trace preservation	Synthesis of debugging information
Stacking	Trace transformation	Laying out stack frames
Asmgen	Trace transformation	Emission of assembly code

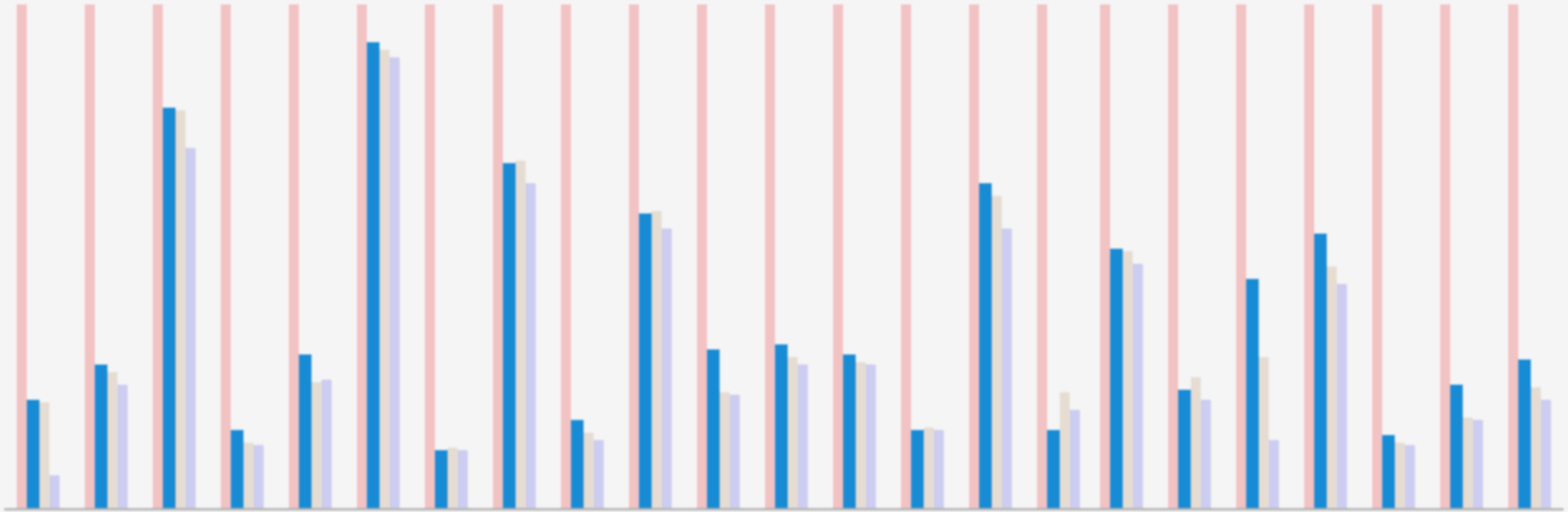
Conclusion and perspectives

Conclusion

- A machine checked-proof that a mildly modified version of the CompCert compiler preserves cryptographic constant-time
- A carefully crafted methodology that maximises proof reuse

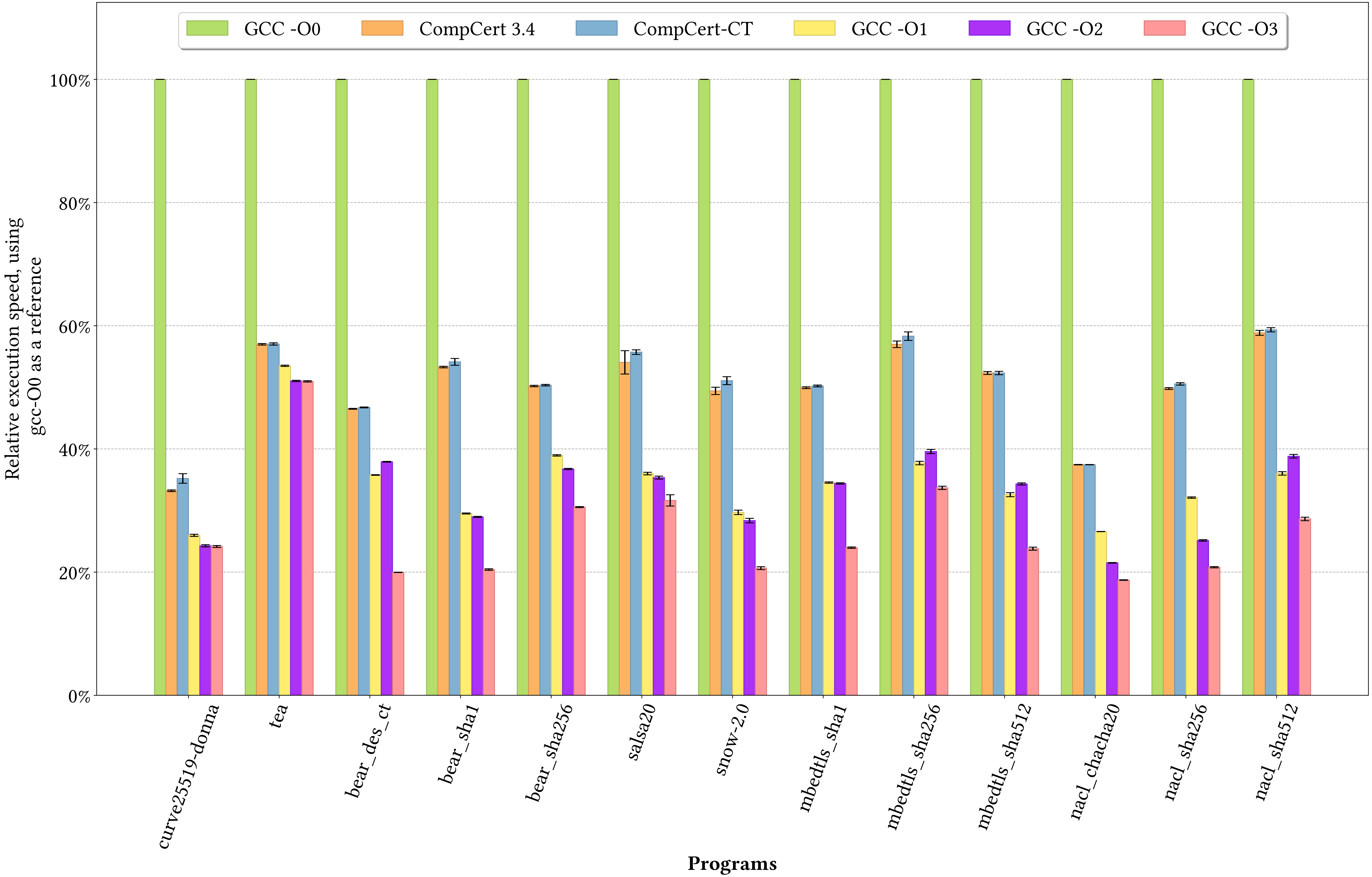
Perspectives

- Make CompCert generate more efficient code for crypto programs (e.g. using SIMD instructions)
- Explore other observational information-flow policies and adapt CompCert



Execution times of 23 benchmark programs compiled with ■ `gcc -O0`, ■ CompCert, ■ `gcc -O1`, and ■ `gcc -O2`

https://www.absint.com/compcert/


GCC -O0
CompCert 3.4
CompCert-CT
GCC -O1
GCC -O2
GCC -O3
Relative execution speed, using gcc-O0 as a reference
100%
80%
60%
40%
20%
0%
curve25519-donna
tea
bear_des_ct
bear_sha1
bear_sha256
salsa20
snow-2.0
mbedtls_sha1
mbedtls_sha256
mbedtls_sha512
nacl_chacha20
nacl_sha256
nacl_sha512
Programs