1

Towards Verified Decompilation using Lean 4

Joe Hendrix
Galois, Inc

Acknowledgements


Andrew Kent


Simon Winwood


Ledah Casburn


Jason Dagit


Andrei Stefanescu

We are building this with help from

- Developers of Lean 4: Leonardo de Moura and Sebastian Ullrich
- CVC4 Team: Haniel Barbosa, Mathias Preiner, Arjun Viswanathan, Cesare Tinelli, and Clark Barrett
- UIUC X86_64 ISA formalization: Sandeep Dasgupta

This presentation describes work sponsored by the Office of Naval Research under Contract No. N68335-17-C-0558.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

About Galois

- Galois focus is to ensure **trusted** systems are **trustworthy**.
- Predominant focus on consulting, tools, techniques and demonstrations.
- Contribute heavily to open source.
- Use spinouts for commercialization.
- We have expertise and hire in a variety of area: cryptography, data science, distributed systems, formal methods, hardware, programming languages, etc.

4

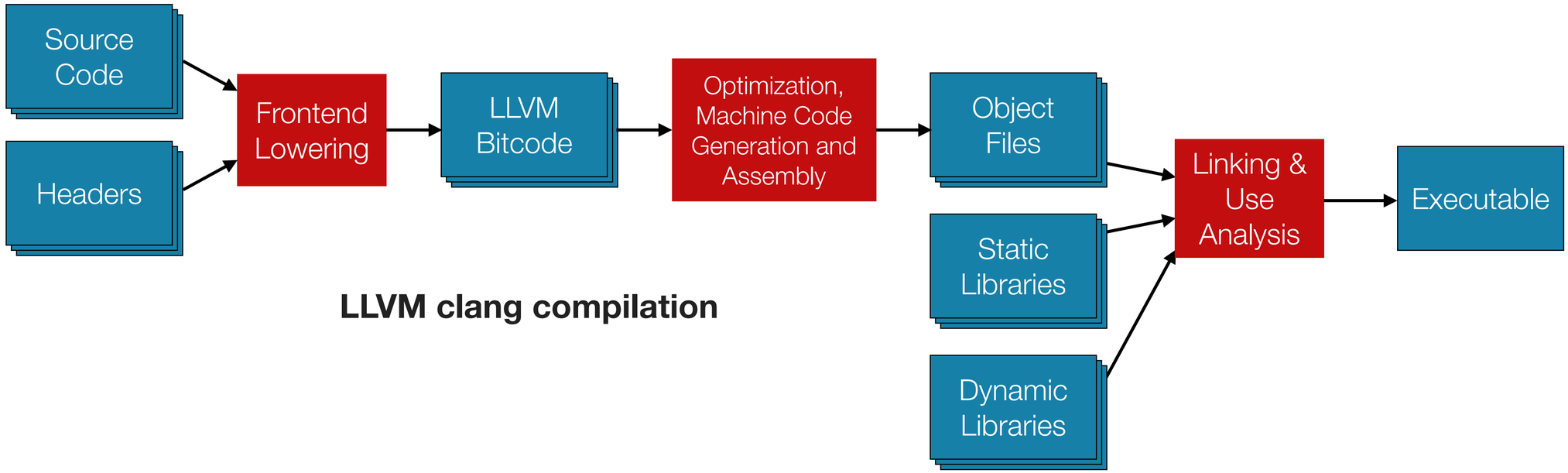
Lean at Galois

- Our SAW symbolic execution tools internally use a dependent calculus, SAWCore, for representing program values.
  - SAWCore has been heavily influenced by Lean.
- We have also had two projects using Lean for verification:
  - A protocol verification effort using Lean 3.
  - A LLVM decompiler verifier using Lean 4
- There are many people at Galois interested in other ITPs as well: Coq, Isabelle, ACL2

Decompilation

Decompilation

- **Decompilation** recovers information obscured during compilation to reverse the process.



LLVM clang compilation

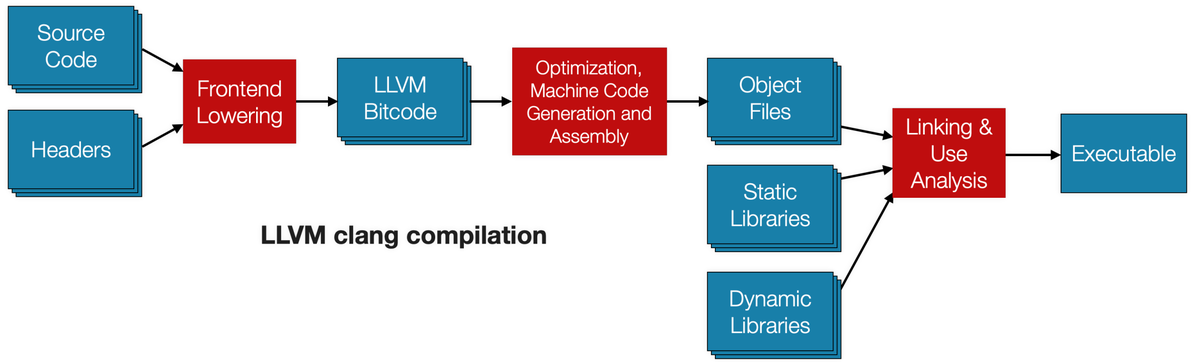
Decompiler Uses

- Decompilation is currently used for **program understanding**.
  - Decompile to a **language understandable by people**.
  - Use a combination of static analysis, heuristics, and hints from the user.
  - Without hints or existing source to target, it is likely impossible to recover the original source.
- Research looking into using decompiler to **recompile** an application.
  - Use new security measures or optimizations on legacy applications to take advantage of new features or security mechanisms.
  - Patch an existing binary.
  - Porting a compiled binary from one platform to another.

Recompilation Observations

Recompilation use case differences.

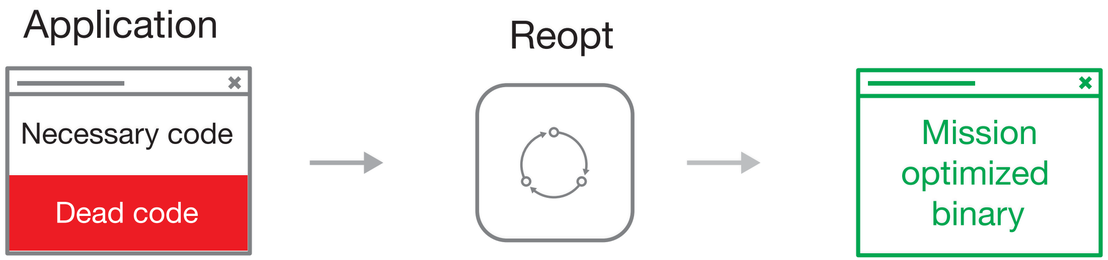
- Sufficient to lift to **compiler IR** or object file representation rather than source.
- **Assured** decompilation is much more important.



LLVM clang compilation

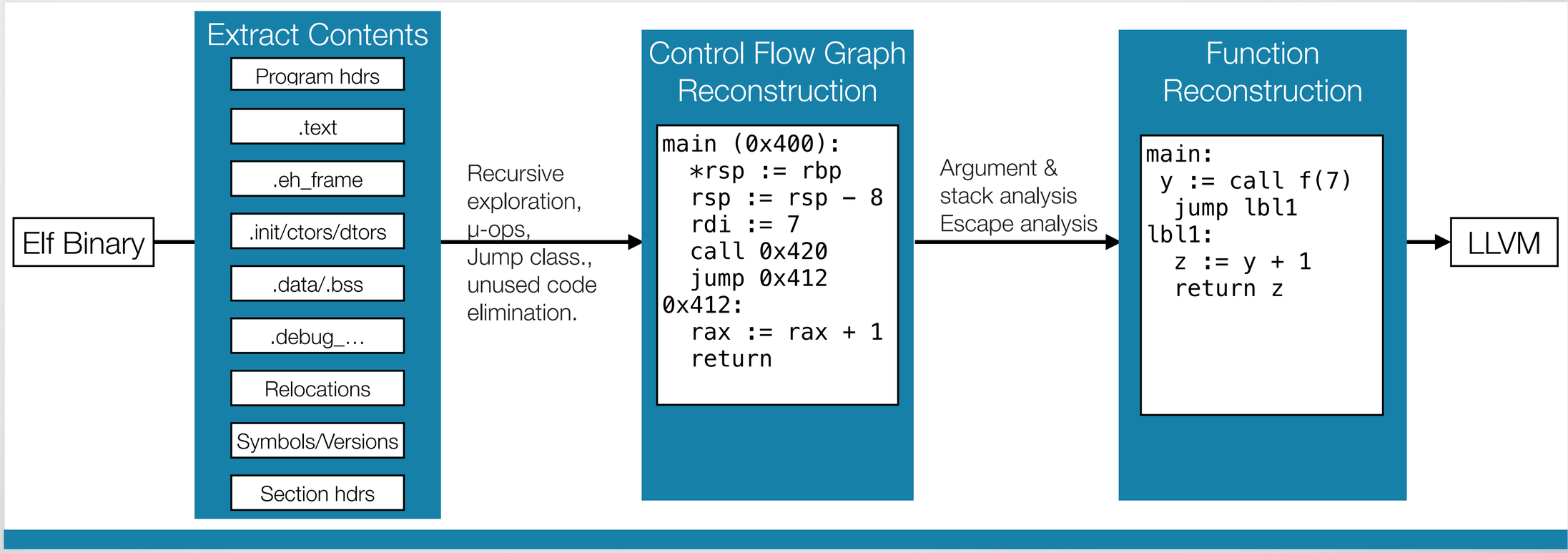
Program Recompilation

- My talk today is focused on binary raising for **reopt**, a tool for optimization of compiled executables.



- This can be used for optimization, dead code elimination, and hardening legacy binaries.

Reopt's Decompilation Pipeline



Verification

Verification Properties

Recompilation Soundness

- Every observable execution in the LLVM should be possible in the machine code program.

$$t \in traces(P_{LLVM}) \Rightarrow \exists\ t' \in traces(P_{MC}),\ t \equiv t'$$

Verification Soundness

- If a property is true of the raised program, then it should be true of the machine code program.

Observational Equivalence

- Our current notion of equivalence is based on event traces.
- Required events include:
  - Writes to non-stack addresses.
  - Other operations that may raise signals (e.g., divide-by-zero).
  - System calls
- Internally, we make additional equivalence checks for compositional purposes.

Verification Approaches

1. Build a **verified decompiler** using interactive theorem proving.

- Decompilation is an open-ended problem.
- Very complex to implement, and needs continued improvement.

2. Use an automated checker to check the programs are equivalent.

- Program equivalence is ordinarily decidable…
- However, the decompiler output is structurally similar to input binary.
- We have developed a **compositional approach** that checks equivalence of **basic blocks** using SMT solving.

Verification Approaches

1. Build a **verified decompiler** using interactive theorem proving.

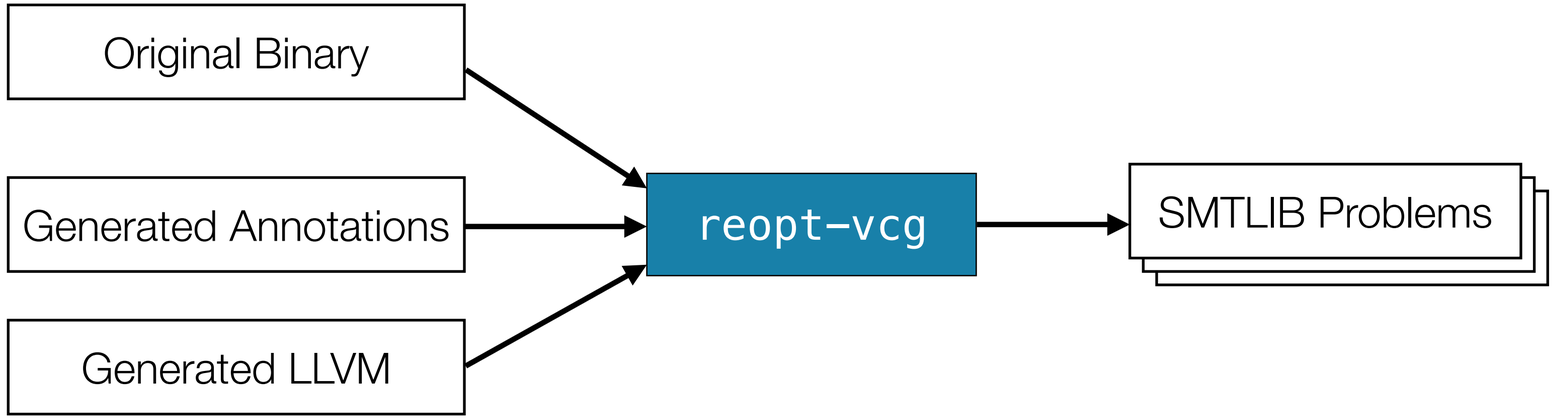
- Decompilation is an open-ended problem.
- Very complex to implement, and needs continued improvement.

2. Use an automated checker to check the programs are equivalent.

- Program equivalence is ordinarily decidable…
- However, the decompiler output is structurally similar to input binary.
- We have developed a **compositional approach** that checks equivalence of **basic blocks** using SMT solving.

Verification Approach

- We have implemented a verifier based on translation validation.



Correctness claim: If all SMTLIB SAT problems are unsat, then the generated LLVM refines the original binary

Compositional Proofs

- The key to making this work is to develop a compositional proof strategy, and this uses additional proof obligations:
  - Functions must respect the ABI (how arguments are passed, callee-saved registers, etc)
  - The bottom of the stack must have unallocated guard pages, and no function stack frame is larger than the guard page region.
  - Our block annotations describe preconditions on each block, which can be assumed at the start of each block execution, and must be proven when jumping to a new block.
- One limitation, as we do not control LLVM stack usage completely, the recompiled LLVM program may use more stack space than the original machine code program.

Current Verifier Status

- Reopt generates the annotation files.
- Working on small test programs, and now debugging issues on a larger web server example.
- Code statistics

Tool/Library	Rough LOC
reopt	134kloc
reopt-vcg (- semantics)	23kloc
Lean x86 semantics	62kloc

- reopt and reopt-vcg are publicly available, but still under active development.

  `https://github.com/GaloisInc/reopt`

  `https://github.com/GaloisInc/reopt-vcg`

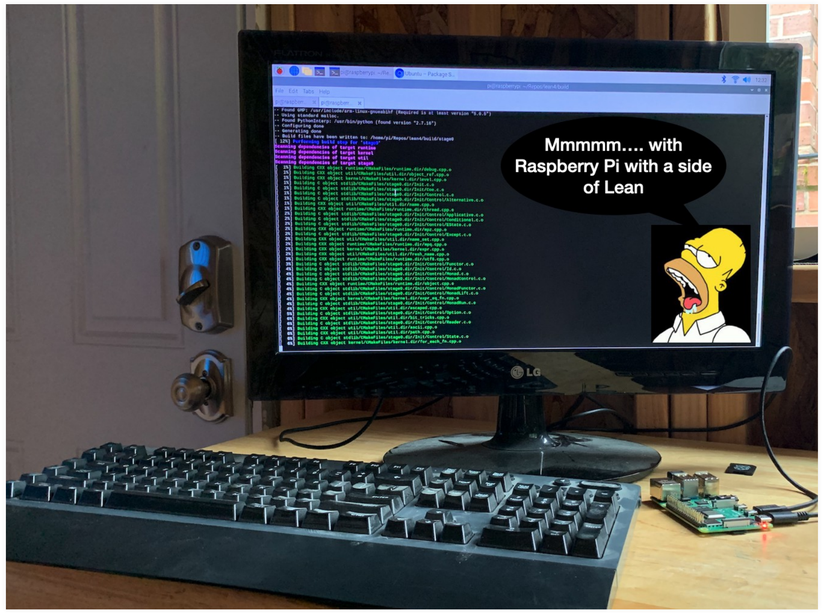
- Fledgling libraries in Lean 4 for:
  - LLVM bitcode parsing
    - Links against libllvm (using `@extern` attribute)
  - Elf object file parsing
  - x86_64 machine code decoding and interpreting
    - As of yesterday we can import semantics specific using UIUC K x86_64 formalization.
  - SMT Generation and running external solvers.

Planned Work

- Continue expanding the scope of programs we can decompile and verify.
- Type analysis of heap in machine code programs.
  - Needed for sound verification of programs with indirect function calls.
- Formalize correctness of reopt-vcg

One more thing

- We are also working on getting Lean 4 controlling a robot.
- Andrew Kent is working on this.
- We are really excited about a programming language with powerful verification capabilities.



Thank You