Scalable semantic virtual machine framework for language-agnostic static analysis

Maxim Menshchikov

10th of September 2018

St.Petersburg State University

Author

Maxim Menshchikov —

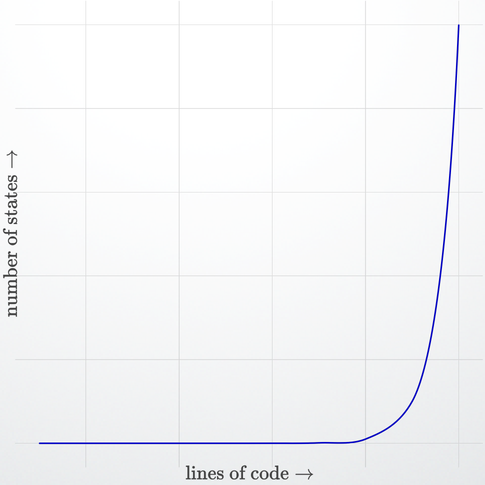
- Postgraduate student at St. Petersburg State University, faculty of Mathematics and Mechanics, department of Software Engineering.
- Software engineer at OKTET Labs.
- Ex-security analyst.

Research interests:

- Methods of static analysis, development of static analyzers.
- Virtualization.
- Parallelization and distributed technologies.
- Networking.

Scope of the problem

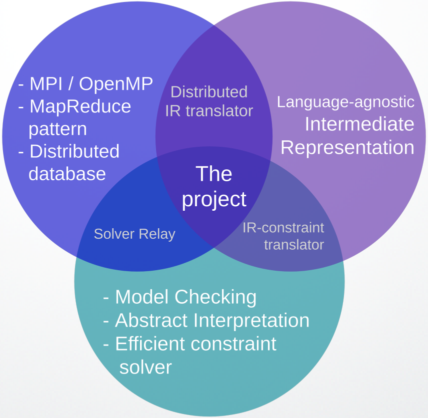
State space is often subject to combinatorial explosion when the size of the program goes beyond some boundary.



Towards the best approach

Our goal is to analyze programs in a distributed manner while keeping algorithms detached from distributed technologies.

Other variants:
Threads?
Single storage?
Agents?



Other variants:
LLVM IR?
Assembler?
Plain AST?

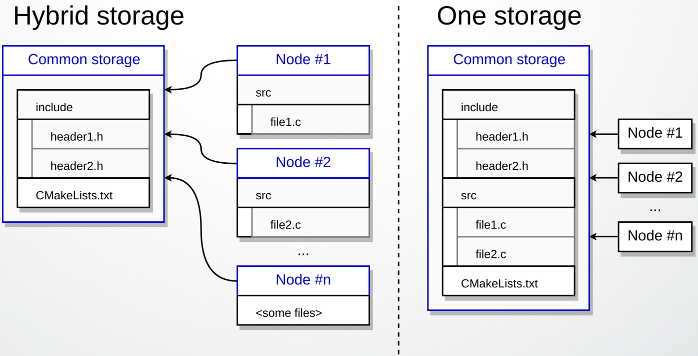
Plan of a talk

1. **File distribution** mechanism → 2. **Intermediate Representation** language → 3. Semantic **Hypervisor** / **Virtual Machine** (+object lookup) →

→ 4. **Principles** and **obstacles** → 5. **Testing**, conclusions

File system

We use a hybrid approach in which only selected files are transferred to nodes, while the rest (e.g. headers) reside on a distributed storage.



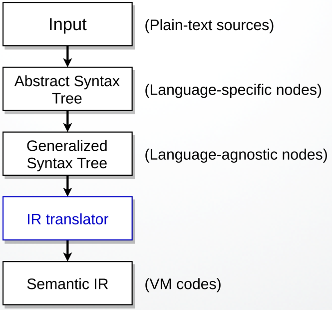
File distribution

We use a simple suboptimal algorithm:

1. Calculate total input file size: $S = \sum s_i$. Let's consider m to be a number of files.
2. Divide it by the number of nodes (N): $S_n = \frac{S}{N}$
3. For each node $1 \leq i \leq N$ initialize $r_i = S_n$ (the remainder)
4. Loop until all files are scheduled:
   4.1 Loop by nodes (i):
      4.1.1 Terminate loop if all files are scheduled.
      4.1.2 Take the biggest file s_j ($j \in [0, m]$) of size satisfying the following requirement:
      $$s_j \leq r_i$$
      4.1.3 If such file doesn't exist, take the smallest file satisfying this requirement:
      $$s_j > r_i$$
      4.1.4 $r_i := r_i - s_j$

Language-agnostic processing

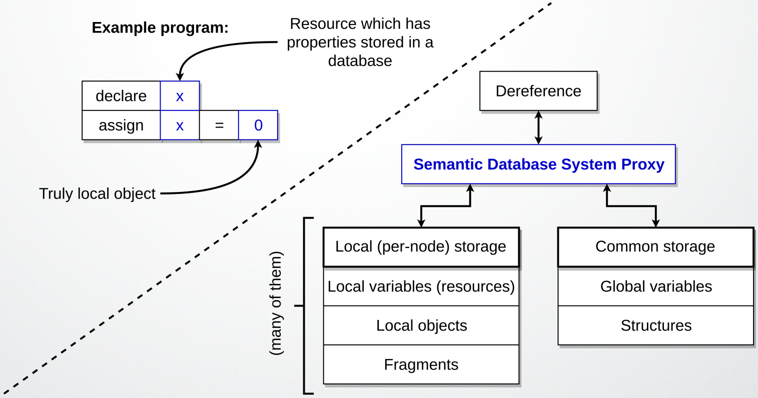
To perform language-agnostic processing we use our own intermediate representation language. Programs in this IR are generated by C $\to$ generalized syntax $\to$ IR translator.



Alternative: LLVM IR.

Semantic storage behind the IR

The IR is not a standalone language. All semantic objects are serializable and have ID, properties and tags. Immutable objects are used to mimic low-level behavior.



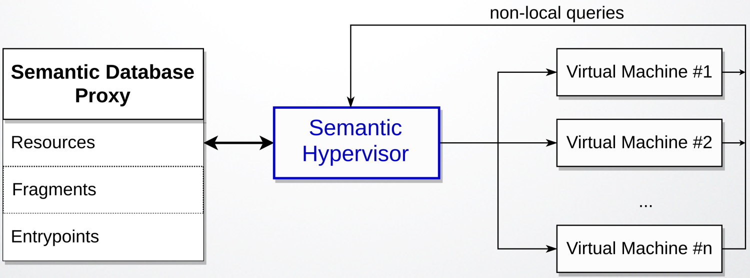
VM IR operational codes

Following operational codes are commonly used in IR throughout the analyzer:

declare **resource**	*resource declaration*
invoke **expr1** = **expr2** with arguments **arg1, ..., argN**	*procedure invocation with storing result to a separate variable*
assign **resource** / **object** = **expression**	*assignment*
branch **condition** [or continue]	*start of a block with condition*
end branch	*end of a block with condition*
check **expression**	*condition that is never true*
constraint **expression**	*condition that is always true (constraint)*
return **expression**	*return from a function*
start / end	*start/end of a procedure*
system **cmd**	*internal command*

Hypervisor and virtual machines

Operational codes are executed on Semantic Virtual Machines, supervised by Semantic Hypervisor. Virtual Machines alone do intraprocedural analysis, while Hypervisor can schedule interprocedural tasks on a specific node.

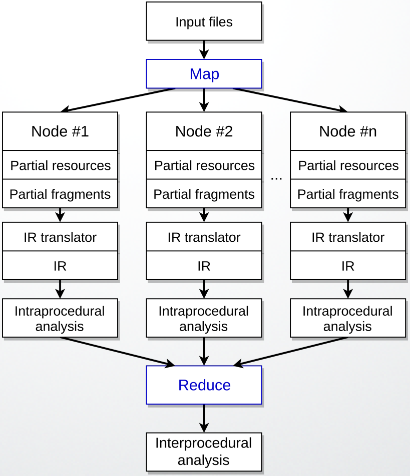


Object lookup algorithm

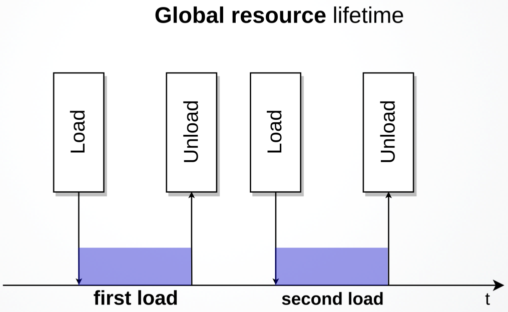
1. Local lookup:
   1.1 Search for a target resource among local resources.
   1.2 Search for a target const reference and const value among local objects.
2. Global lookup:
   Search for a target resource among global resources and load it if found using hypervisor's relay.
3. Horizontal lookup:
   Search for a target resource among resources of other nodes.
4. Preliminary creation:
   Create a global resource with non-existent mark of a deduced type.
   (missing information would be filled later).

Main principles

1. MapReduce pattern.
   MapReduce-like pattern is used for the analysis whenever applicable.



Main principles (2)



2. Lazy semantics loading.

   Semantics is always stored in a database and can be loaded on demand.

3. Automatic data unloading.

   Unused data is deallocated as soon as possible.

Main principles (3)

4. Code simplification.
   Remove: unused variables, branches and loops with conditions that are always false, complex syntax constructs.
5. Keeping data as local as possible, eliminating transfer overhead.

Obstacles

- Distributed source location keeping isn't trivial.

```
if (x) {

line, col
```

- Looping immutable objects requires a variable refresh.

n times

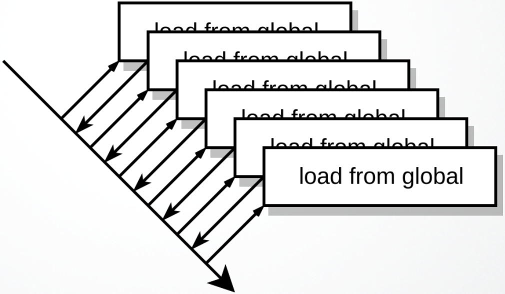
```
init ConstRef(X) = i

init ConstVal(Y) = i

assign ConstRef(X) = ConstVal(Y) + 1
```

Obstacles

- Unstructured sources can heavily impact the performance.



Test results

We tested the approach on various inputs of our Verification Example Framework[1]. Intraprocedural and interprocedural model checking had been started[2]. Similar results have been achieved with Linux kernel.

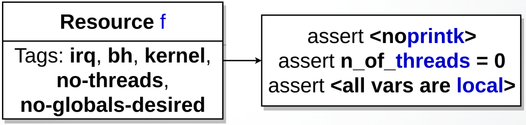
Approach	Performance (%)	Precision (%)
Semantic VM	100	100
Distributed Semantic VM (4 nodes)	369	100
Distributed Semantic VM (8 nodes)	720	100
VM with assembler-like input	114	63
Generic semantics-driven analysis	176	80

[1] Potentially will be open in the future.

[2] Precision is derived from a number of correct checking tries

Novelty

- Unlike LLVM IR and so — our language is built specifically for static analysis and its programs are easily translatable to constraint systems.



- Almost linear scaling is achieved for some kinds of analysis.
- Flexibility of the approach is greater since virtual machine implementation can know a little about distribution.

Conclusion

- A special virtual machine IR for static analysis had been developed.
- An algorithm for file distribution had been battle-tested and had shown to be "good enough".
- Algorithms for distributed object lookup and other operations had been prepared.
- Hypervisor/Semantic virtual machine framework had been designed and implemented as a part of our analyzer, showing almost linear scaling in some kinds of analysis.

Still need to:

- Further evaluate the architecture on various projects.
- Prepare a production-ready solution after the analyzer is done (Now it is only a Proof of Concept).

Questions?

Other publications about the project:

- Menshchikov M.A. Hybrid system of static analysis with proof-based verification of invariants. Master's thesis.
- Menshchikov M., Lepikhin T. 5W+1H Static Analysis Report Quality Measure. TMPA-2017.
- Menshchikov M.A., Lepikhin T.A. Applying MapReduce to static analysis. CPS-2017.
- Menshchikov M.A. Race condition detection in C code using static analysis. Bachelor's thesis.
- Menshchikov M.A., Lepikhin T.A. Function context detection in the program source code. CPS-2016.

Tech

Main technologies and projects used around the analyzer:

Parser backend:	LLVM/Clang
Database:	MongoDB
Distribution:	OpenMPI
Parallelization:	OpenMP
Constraint solver:	CVC4
Testing:	Google Test