Binary Decompilation to LLVM IR

Sandeep Dasgupta* Vikram Adve†

August 24, 2016

Analyzing and optimizing programs from their executable has a long history of research pertaining to various applications including security vulnerability analysis, untrusted code analysis, malware analysis, program testing, and binary optimizations.

This work is a step towards the same broader objective by decompiling the input binary into an intermediate form (IR) of LLVM, which is a widely-used compiler infrastructure. The main challenge of the work involves extracting the variable (both scalar and aggregate) and type information from the input binary into a fully functional IR. For this we have used a publicly available tool called McSema [1] which can convert x86 machine code to functional LLVM IR. McSema support translation of x86 machine code, including integer, floating point, and SSE instructions. One of the downside of McSema recovered IR is that the variable (scalar/aggregate) and type information is missing. The current work is about recovering those.

As an initial step, we have developed tools [2] for better debugging of McSema generated IR. Notables are "source mapper", which maps source (input binary) information to generated LLVM IR, and a LLVM backward slicer. Also we have identified optimization oportunities, like scalar replacement of aggregates, in the design of McSema generated IR to improve its quality.

Mcsema uses a big flat array to model the runtime process stack i.e. all the reads/writes made by a binary on its runtime stack are modeled into this array. The first step towards our goal is to identify variables in this array and promote them as separate symbols which requires deconstructing this global array, that Mcsema shares between all the procedure, into per procedure array which is used to model the stack frame of that procedure. Such stack deconstruction is important because doing symbol promotion right on the global array could be very conservative because an indirect write made by a different procedure may prevent symbol promotion in the current procedure. We have implemented a transformation pass which can do the stack deconstruction and also implemented variable recovery and symbol promotion as described in [4, 3]

The next step is to infer the type of the recovered variables. We are developing a prototype model for type inferencing based on [4, 5].

References

[1] *Mcsema*. `https://github.com/trailofbits/mcsema`.

[2] *Source mapper and llvm backward slicer*. `https://github.com/sdasgup3/llvm-slicer`.

[3] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, *A compiler-level intermediate representation based binary analysis and rewriting system*, in Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13, New York, NY, USA, 2013, ACM, pp. 295–308.

*Electronic address: `sdasgup3@illinois.edu`
†Electronic address: `vadve@illinois.edu`

[4] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, *Scalable variable and data type detection in a binary rewriter*, in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, New York, NY, USA, 2013, ACM, pp. 51–60.

[5] M. Noonan, A. Loginov, and D. Cok, *Polymorphic type inference for machine code*, in Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, New York, NY, USA, 2016, ACM, pp. 27–41.