RESEARCH STATEMENT

Venkatesh Srinivasan (venk@cs.wisc.edu, http://www.cs.wisc.edu/∼venk)

My research interests span the areas of program synthesis, program analysis, and programming languages. Modern research in program synthesis has led to the development of various tools and techniques for creating correct-by-construction programs from specifications. Applications of program synthesis span from transforming spreadsheet data to synthesizing forwarding tables in routers. My research advances the state-of-the-art in program synthesis by applying program-synthesis techniques to binary analysis and rewriting. In particular, I developed (i) algorithms that synthesize machine-code implementations from specifications, and (ii) tools that use machine-code synthesis to either modify or extract a reusable component from a binary. In the following, I will describe the aforementioned algorithms and tools while highlighting my research contributions, and present some directions that I plan to pursue in the future.

Algorithms for Machine-code Synthesis

Machine-code synthesis is the problem of synthesizing an instruction sequence that implements a semantic specification of the desired behavior, given as a formula in quantifier-free bit-vector logic (QFBV). A key challenge in synthesizing machine code is the enormous size of the synthesis search-space. Instruction sets like Intel's IA-32 have around 43,000 unique instruction schemas; this huge instruction pool, along with the exponential cost inherent in enumerative synthesis, results in an enormous search space for the synthesizer: even for relatively small specifications, a naïve synthesizer might take several days to find an implementation. In this section, I describe how I addressed this challenge while designing an algorithm for machine-code synthesis, and subsequent improvements to the algorithm. The techniques I developed to combat the size of the synthesis search-space are not restricted to machine code in particular, and can be applied to other program synthesizers as well.

Base Algorithm. I developed a machine-code synthesizer called McSynth [2], which is parameterized by the instruction set of the target instruction-sequence. McSynth is the first machine-code synthesizer to be applied to the integer subset of a full instruction set. McSynth uses (i) a divide-and-conquer strategy to split the input formula into several independent smaller sub-formulas and synthesize code for the sub-formulas, and (ii) footprint-based search-space pruning heuristics to prune away candidates during synthesis. Experiments with the IA-32 instruction set showed that McSynth is 3 to 5 orders of magnitude faster than a baseline enumerative synthesizer.

Speeding up machine-code synthesis. McSynth brought down synthesis time from days to minutes, but it was still not fast enough: McSynth times out for several larger QFBV formulas; even for smaller formulas, McSynth takes several minutes to find an implementation. Consequently, if a binary-rewriter client supplies a formula as input to McSynth, the client has to wait several minutes before McSynth finds an implementation. This delay might not be tolerable for a client that has to invoke the synthesizer multiple times to rewrite an entire binary (e.g., a machine-code partial evaluator). I made several improvements to the synthesis algorithm used in McSynth, and developed McSynth++ [5], which is an improved version of McSynth. In addition to a novel pruning heuristic, the improvements incorporate a number of ideas known from the literature, which I adapted in novel ways for the purpose of speeding up machine-code synthesis. Experiments for IA-32 showed that the improvements enable synthesis of code for 12 out of 14 formulas on which McSynth times out, speeding up the synthesis time by at least 1981X, and for the remaining formulas, speeds up synthesis by 3X.

Model-assisted machine-code synthesis. McSynth and McSynth++ perform a linear search over the space of instruction sequences, i.e., after exhausting all one-instruction sequences, they search through all two-instruction sequences, and so on (modulo pruning). One can see that this search strategy is not very efficient because not all k-instruction sequences are equally likely to implement the input specification. (For example, if the specification computes the sum of the contents of two registers, then an instruction that performs bitwise AND is clearly not relevant.) In ongoing work, I am collaborating with a machine-learning graduate student at UW-Madison, Ara Vartanian, to convert the linear search in McSynth into a best-first search over the space of instruction sequences. The cost heuristic for the search comes from language and correlation models built from a corpus of equivalent ⟨QFBV formula, instruction sequence⟩ pairs. (Note that it is straightforward to build such a corpus by harvesting several instruction sequences from binaries, and converting them into QFBV formulas via symbolic execution.) The language model favors instruction sequences that occur more frequently in the corpus, and the correlation model favors instruction sequences whose opcodes and operand sizes are more likely to implement the input formula. Once built, our synthesizer will be the first of its kind to use features of specifications and implementations to guide its search.

Binary rewriting via synthesis

One of my principal motivations to pursue machine-code synthesis is to develop a general framework for semantics-based binary rewriting. A machine-code synthesizer decouples transformations to program semantics from synthesis, and allows one to create multiple binary-rewriting tools that use the following recipe: (i) convert instructions in the binary to QFBV formulas, (ii) use analysis results to transform QFBV formulas, and (iii) use the synthesizer to produce an instruction-sequence that implements each transformed formula. Below, I describe two binary-rewriting tools that I created by instantiating the above framework.

Machine-code partial evaluation. I developed the first partial evaluator that works on machine code.[1] The partial evaluator WiPEr [3] performs off-line partial evaluation of machine code. WiPEr's algorithm follows the classical two-phase approach of binding-time analysis (BTA) followed by specialization. WiPEr's specializer specializes an explicit representation of the semantics of an instruction, and emits residual code via machine-code synthesis. Moreover, to create code that allows the stack and heap to be at different positions at run-time than at specialization-time, the specializer represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. WiPEr can be used to specialize binaries with respect to commonly used inputs to produce faster binaries, as well as to extract an executable component from a bloated binary (e.g., extract compress from the `gzip` binary).

Machine-code slicing. Most instructions in instruction sets such as Intel's IA-32 and ARM are multi-assignments: they have several inputs and several outputs (registers, flags, and memory locations). This aspect of the instruction set introduces a granularity issue during slicing: there are often instructions at which we would like the slice to include only a subset of the instruction's semantics, whereas the slice is forced to include the entire instruction. Consequently, the slice computed by state-of-the-art tools is very imprecise, often including essentially the entire program. I developed a tool McSlice [4] that slices machine code more accurately. To counter the granularity issue, McSlice performs slicing at the microcode level, instead of the instruction level, and obtains a more precise microcode slice. To reconstitute a machine-code program from a microcode slice, McSlice uses machine-code synthesis. Experiments on IA-32 binaries of FreeBSD utilities show that, in comparison to slices computed by a state-of-the-art tool (CodeSurfer/x86), McSlice reduces the size of backward slices by 33%, and forward slices by 70%. The backwards executable-slicing functionality of McSlice can also be used to extract executable components from binaries (e.g., extract line count from the `wc` binary).

Other Contributions

In the past, I have developed a reverse-engineering tool called Lego [1], which recovers class hierarchies and composition relationships from stripped binaries. Lego takes a stripped binary as input, and uses information obtained from dynamic analysis to (i) group the functions in the binary into classes, and (ii) identify inheritance and composition relationships between the inferred classes. The software artifacts recovered by Lego can be subsequently used to understand the object-oriented design of software systems that lack documentation and source code, e.g., to enable interoperability. Experiments with binaries of open-source applications (whose sizes ranged from 3.2 to 122 thousand lines of C++ code) show that the class hierarchies recovered by Lego have a high degree of agreement—measured in terms of precision and recall—with the hierarchy defined in the source code. Lego complements the existing suite of reverse-engineering tools that recover software artifacts (e.g., types, variable proxies, control-flow and program-dependence graphs, etc.) from stripped binaries.

Future Directions

In the future, I plan to continue my existing line of work on instantiating the synthesis-based binary-rewriting framework to develop more tools, and developing better algorithms for enumerative program synthesis.

Porting existing binaries to approximate hardware. Approximate computing is a promising approach that trades quality of result for energy efficiency. Research in approximate computing has led to the development of microarchitectures and high level languages that support approximation. However, there are no tools that can port existing binaries that run on deterministic hardware to approximate hardware. A key property that such a tool has to enforce is that values computed by approximate instructions should never affect deterministic instructions. I plan to borrow ideas from my work on machine-code slicing and partial evaluation, and use them in conjunction with machine-code synthesis to develop a tool that can translate deterministic binaries into approximate ones.

[1] Confirmed by personal communication with Neil Jones, Robert Glück, and Saumya Debray.

Verifier-assisted synthesis. Research in program-verification technology over the last two decades has resulted in a large collection of tools and techniques for program verification. Using existing program-verification tools, one could build a large corpus of properties and programs that satisfy/violate the properties. Subsequently, the corpus can be used to build models that relate features of properties to features of programs that satisfy/violate the properties. I plan to investigate building such models to assist the search performed by enumerative program synthesizers.

References

[1] V. Srinivasan and T. Reps, Recovery of class hierarchies and composition relationships from machine code, In *CC*, 2014.

[2] V. Srinivasan and T. Reps, Synthesis of machine code from semantics, In *PLDI*, 2015.

[3] V. Srinivasan and T. Reps, Partial evaluation of machine code, In *OOPSLA*, 2015.

[4] V. Srinivasan and T. Reps, An improved algorithm for slicing machine code, In *OOPSLA*, 2016.

[5] V. Srinivasan, T. Sharma, and T. Reps, Speeding up machine-code synthesis, In *OOPSLA*, 2016.