GDSL: A Universal Toolkit for Giving Semantics to Machine Language

Julian Kranz, Alexander Sepp, and Axel Simon[1]

Technische Universität München, Institut für Informatik II, Garching, Germany
`firstname.lastname@in.tum.de`

Abstract. The static analysis of executable programs has gained importance due to the need to audit larger and larger programs for security vulnerabilities or safety violations. The basis for analyzing executables is the decoding of byte sequences into assembler instructions and giving a semantics to them. We use our domain specific language GDSL that facilitates this task to specify Intel x86 semantics. In particular, we show how simple optimizations of the generated code can drastically reduce its size. Moreover, since these optimizations are also written in GDSL they can be re-used with other processor front-ends. Hence, analyses based on our toolkit can be adapted to several architectures with little change.

1 Introduction

The static analysis of executable programs has gained increasing importance in the last decade. Reasons are the need to audit larger and larger programs for security vulnerabilities, the on-line detection of malware in virus scanners, and the need to verify software in the presence of binary third-party libraries, inline assembler, and compiler-induced semantics. The basis for analyzing executables is the decoding of byte sequences into assembler instructions and giving a semantics to them. The challenge here is one of scalability: a single line in a high-level language is translated into several assembler (or "native") instructions. Each native instruction, in turn, is translated into several semantic primitives. These semantic primitives are usually given as an *intermediate representation* (IR) and are later evaluated over an abstract domain [3] tracking intervals, value sets, taints, etc. In order to make the evaluation of the semantic primitives more efficient, a *transformer-specification language* (TSL) was recently proposed that compiles the specification of each native instruction directly into operations (transformers) over the abstract domain [6], thus skipping the generation of an IR. These tailored transformers are then optimized by a standard compiler. Our toolkit follows the more traditional approach of generating an IR that an analysis later interprets over the abstract domains. In contrast to the TSL approach, we perform optimizations on the IR program that represents a complete basic block rather than on a single native instruction. We show that the semantics of instructions can be simplified considerably when taking the surrounding instructions into account which highlights the optimization potential of using an IR.

[1] This work was supported by DFG Emmy Noether programme SI 1579/1.

$$\begin{array}{lcl}
\textbf{stmts} & ::= & \textbf{stmt} \mid \textbf{stmt} \,\underline{;}\, \textbf{stmts} \\
\textbf{stmt} & ::= & \textbf{var} \,\underline{=}\, \underline{:}\, \underline{\mathit{int}}\ \textbf{expr} \\
 & \mid & \textbf{var} \,\underline{=}\, \underline{:}\, \underline{\mathit{int}}\ \underline{[}\ \textbf{addr}\ \underline{]} \\
 & \mid & \underline{[}\ \textbf{addr}\ \underline{]}\ \underline{=}\, \underline{:}\, \underline{\mathit{int}}\ \textbf{expr} \\
 & \mid & \underline{\mathit{if}}\ \textbf{sexpr}\ \underline{\mathit{then}}\ \textbf{stmts}\ \underline{\mathit{else}}\ \textbf{stmts} \\
 & \mid & \underline{\mathit{while}}\ \textbf{sexpr}\ \underline{\mathit{do}}\ \textbf{stmts} \\
 & \mid & \underline{\mathit{cbranch}}\ \textbf{sexpr}\ \underline{?}\ \textbf{addr}\ \underline{:}\ \textbf{addr} \\
 & \mid & \underline{\mathit{branch}}\,(\,\underline{\mathit{jump}} \mid \underline{\mathit{call}} \mid \underline{\mathit{ret}}\,)\ \textbf{addr} \\
 & \mid & (\,\textbf{var}\ \underline{:}\ \underline{\mathit{int}})^{*}\ \underline{=}\ \underline{''}\,\underline{\mathit{id}}\,\underline{''}\,(\,\textbf{linear}\ \underline{:}\ \underline{\mathit{int}})^{*} \\
\textbf{cmp} & ::= & \underline{\leq_s} \mid \underline{\leq_u} \mid \underline{<_s} \mid \underline{<_u} \mid \underline{=} \mid \underline{\neq}
\end{array}
\qquad
\begin{array}{lcl}
\textbf{var} & ::= & \underline{\mathit{id}} \mid \underline{\mathit{id}}\ \underline{.}\ \underline{\mathit{int}} \\
\textbf{addr} & ::= & \textbf{linear}\ \underline{:}\ \underline{\mathit{int}} \\
\textbf{linear} & ::= & \underline{\mathit{int}}\ \underline{\cdot}\ \textbf{var}\ \underline{+}\ \textbf{linear} \mid \underline{\mathit{int}} \\
\textbf{sexpr} & ::= & \textbf{linear} \mid \underline{\mathit{arbitrary}} \\
 & \mid & \textbf{linear}\ \textbf{cmp}\ \underline{:}\ \underline{\mathit{int}}\ \textbf{linear} \\
\textbf{expr} & ::= & \textbf{sexpr} \\
 & \mid & \textbf{linear}\ \textbf{bin}\ \textbf{linear} \\
 & \mid & \underline{\mathit{sign\text{-}extend}}\ \textbf{linear}\ \underline{:}\ \underline{\mathit{int}} \\
 & \mid & \underline{\mathit{zero\text{-}extend}}\ \textbf{linear}\ \underline{:}\ \underline{\mathit{int}} \\
\textbf{bin} & ::= & \underline{\mathit{and}} \mid \underline{\mathit{or}} \mid \underline{\mathit{xor}} \mid \underline{\mathit{shr}} \mid \ldots
\end{array}$$

Fig. 1. The syntax of our RReil (Relational Reverse Engineering Language) IR. The construct “$\underline{:}\ \underline{\mathit{int}}$” denotes the size in bits whereas “$\underline{.}\ \underline{\mathit{int}}$” in the **var** rule denotes a bit offset. The statements are: assignment, read from address, write to address, conditional, loop (both only used to express the semantics within a native instruction), conditional branch, unconditional branch with a hint of where it originated, and a primitive *"id"*.

2 RReil Intermediate Representation

Many intermediate representations for giving semantics to assembler instructions exist, each having its own design goals such as minimality [1, 4], mechanical verifiability [5], reversibility [7], or expressivity [1, 9]. Our own RReil IR [9], presented in Fig. 1, was designed to allow for a precise numeric interpretation. For instance, comparisons are implemented with special tests rather than expressed at the level of bits which is common in other IRs [4–6].

3 The Generic Decoder Specification Language (GDSL)

We developed a domain specific language called GDSL [8] that is best described as a functional language with ML-like syntax. It features bespoke pattern matching syntax for specifying instruction decoders. Dependability of GDSL programs is increased by a sophisticated type inference [10] that eliminates the need of specifying any types. The algebraic data types and a special infix syntax facilitates the specification of instruction semantics and program optimizations.

The GDSL toolkit contains a compiler for GDSL as well as decoders, semantic translations and optimizations written in GDSL. The benefit of specifying optimizations in GDSL is that they can be re-used for any input architecture since they operate only on RReil. Besides a few instruction decoders for 8-bit processors, the toolkit provides an Intel x86 decoder for 32- and 64-bit mode that handles all 897 Intel instructions. In terms of translations into RReil, we provide semantics for 457 instructions. Of the 440 undefined instructions, 228 are floating point instructions that we currently do not handle since our own analyzer cannot handle floating point computations. Many of the remaining undefined instructions would have to be treated as primitives as they modify or query the internal CPU state or because they perform computations whose RReil semantics is too cumbersome to be useful (e.g. encryption instructions).

a)
```
val sem-cmovcc insn cond = do
  size <- sizeof insn.opnd1;
  dst <- lval size insn.opnd1;
  dst-old <- rval size insn.opnd1;
  src <- rval size insn.opnd2;

  temp <- mktemp;
  mov size temp dst-old;

  _if cond _then
   mov size temp src;

  write size dst (var temp)
end
```

b)
```
T0 = [32](B)
if (FLAGS/6) {
  T0 = [32](A)
} else {
}
B = [32](T0)
B/32 = [32](0)
```

Fig. 2. The translator function a) and a translation result b)

4 Writing Semantics using GDSL

As a pure, functional language with algebraic data types and a state monad, GDSL lends itself for writing translators in a concise way as illustrated next.

4.1 An Example Intel Instruction

The following GDSL example shows the translation of the Intel instruction `cmov`. The instruction copies the contents of its source operand to its destination operand if a given condition is met. The instruction contains a condition (which is part of the opcode) and two operands, one of which can be a memory location. The translation of the instruction instance `cmovz ebx, eax` (using the Intel x86 architecture with the 64 bit extension) into RReil is shown in Fig. 2b). In order to illustrate the translation, we first detail the output of the GDSL decoder which is a value of the algebraic data type `insn` that is defined as follows:

```
type insn =                                  # an x86 instruction
   CMOVZ of {opnd1: opnd, opnd2: opnd}
 | ...      # other instruction definitions omitted
```

Thus, the `CMOVZ` constructor carries a record with two fields as payload. Both fields are of type `opnd` which, for instance, carry a register or a memory location:

```
type opnd =                                  # an x86 operand
   REG of register
 | MEM of memory
 | ...      # immediates, scaled operands and operands with offsets omitted
```

Note that all variants (here `REG` and `MEM`) implicitly contain information about the access size. In the example above, the instruction `cmovz ebx, eax` is represented by `CMOVZ {opnd1 = REG EBX, opnd2 = REG EAX}` where `EAX` is 32-bits. The following section details helper functions that operate on `opnd` values.

4.2 Generating RReil statements using GDSL monadic functions

Each semantic translator generates a sequence of RReil statements. The sequence is stored inside the state of a monad. Each RReil statement is represented by a respective GDSL monadic function which builds the RReil statement from its arguments and appends it to the current list of statements. The following RReil generator functions are used in the example:

- `mov size destination source`
  The `mov` function generates the RReil `mov` statement that copies the source RReil register to the destination RReil register.
- `_if condition _then statements`
  This function generates the RReil *if·then·else* statement (with an empty *else* branch). The special mix-fix notation `_if c _then e` is a call to a mix-fix function whose name is a sequence of identifiers that each commence with an underscore. For instance, the `_if_then` function above is defined as follows:

```
val _if c _then a = do
  ...        # add if c then a else {} to statement list
end
```

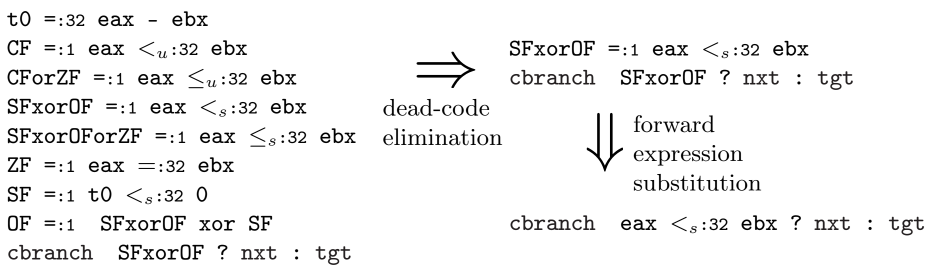
Furthermore, the following functions operate on x86 operands. In the context of the generation of RReil code, this expression is a short form for the RReil register, memory location, or immediate value associated with the x86 operand:

- `sizeof x86-operand`
  Returns the size of an x86 operand in bits; here, `sizeof (REG EBX) = 32`.
- `lval size x86-operand`
  The `lval` function turns an x86 operand into an RReil left hand side expression, that is, either *var* or *[addr]*. Here, `lval 32 (REG EBX)` yields the RReil register B that contains the 32 bits of the Intel `EBX` register.
- `rval size x86-operand`
  The `rval` function turns an x86 operand into an RReil **expr**. In the example, `rval 32 (REG EAX)` yields the RReil register A.
- `write size destination source`
  The `write` function emits all statements necessary to write to an x86 operand. The operand is specified using the *destination* parameter; it is the return value of an associated call to `lval`. In Fig. 2 (b) lines 6 through 7 originate from the call to `write`.

Finally, the `mktemp` function is used to allocate a temporary variable.

4.3 The Translator

The translation function for `cmovz ebx, eax` is shown in Fig. 2a). The **do ... end** notation surrounding the function body is used to execute each of the



```
t0 =:32 eax - ebx
CF =:1 eax <_u:32 ebx                        SFxorOF =:1 eax <_s:32 ebx
CForZF =:1 eax ≤_u:32 ebx          ⟹        cbranch  SFxorOF ? nxt : tgt
SFxorOF =:1 eax <_s:32 ebx       dead-code
SFxorOFForZF =:1 eax ≤_s:32 ebx  elimination      ⇓  forward
ZF =:1 eax =:32 ebx                                  expression
SF =:1 t0 <_s:32 0                                   substitution
OF =:1  SFxorOF xor SF                       cbranch  eax <_s:32 ebx ? nxt : tgt
cbranch  SFxorOF ? nxt : tgt
```

Fig. 3. Translation of the native Intel instructions `cmp eax, ebx; jl tgt` into RReil and applying optimizations. Here, `CForZF`, `SFxorOF`, `SFxorOFForZF` are *virtual flags*, that is, translation-specific variables whose value reflect what their names suggest [9]. Note that this example is idealized since the removed flags may not actually be dead.

enclosed monadic functions in turn. The decoded Intel instruction is passed-in using the *insn* parameter; the condition is determined by the caller depending on the actual mnemonic. The condition is an one-bit RReil expression. In the `cmovz ebx, eax` example, it is `FLAGS/6` which corresponds to the zero-flag.

The translation itself starts with a code block that is very common in instruction semantics: The operation's size is determined by looking at the size of one operand (line 2) and the respective operands are prepared for reading (using the `rval` monadic function) and writing (using the `lval` monadic function). Next, a new temporary RReil register is allocated and initialized to the current value of the destination operand (lines 7 and 8). This completes all preparations; the actual semantics of the instruction is implemented by the code lines 10 through 11. The condition is tested and, if it evaluates to *true*, the source operand is copied to the destination operand. It is important to note that the condition is not evaluated at translation time, but at runtime by RReil. Finally, the (possibly) updated value of the temporary RReil register is written to the corresponding Intel register by code line 13.

One might think that the instruction pointlessly reads the source operand and writes the destination operand in case the condition evaluates to *false*. It is, however, necessary since the writeback can also cause further side effects that still need to occur, even if no data is copied. This is visible in Fig. 2 (b). Since the instruction uses a 32 bit register in 64 bit mode, the upper 32 bits of the register are zeroed even if the lower 32 bits are unchanged (see line 7). This is done by the `write` function.

5 Optimizing the RReil Code

The design of RReil also allows for an effective optimization of the IR which is illustrated in Fig. 3. The example shows the typical code bloat when translating two native instructions where the first sets many flags of which the second only evaluates one. In order to reduce the number of statements, we translate a complete basic block and remove all assignments to dead variables. The next sections consider two optimizations, one of which is currently implemented.

5.1 Liveness Analysis and Dead Code Elimination

The optimization strategy we implement is a dead-code elimination using a backwards analysis on the IR code. To this end, we first need to obtain a set of live variables to start with. A simple approach assumes that all variables are live at end of the block. This has the drawback that assignments to variable that are always overwritten in the succeeding blocks cannot be removed. We address this problem by refining the live-set for basic blocks that do not jump to computed addresses: Specifically, we infer the live variable set of the immediately succeeding blocks and use this live set as start set, thereby removing many more assignments to dead variables. We perform a true liveness analysis, that is, we refrain from marking a variable as live if it is used in the right-hand side of an assignment to a dead variable. For the body of while loops, however, this approach would require the calculation of a fixpoint. Since `while` loops are used rarely by our translator and since their bodies show little potential for optimization, a more conservative notion of liveness is used that does not require a fixpoint computation. This approach marks a variable as live even if it used in an assignment to a dead variable. With this strategy, the dead code elimination takes linear time in the size of the basic block.

5.2 Forward Expression Substitution

In the future, we plan to also perform forward substitution and simplification. These optimizations become important for architectures like ARM where most instructions may be executed conditionally, depending on a processor flag. Compilers use this feature to translate small bodies of conditionals without jumps. Consider a conditional whose body translates to two native instructions $i_1;\ i_2$ that are executed if f holds. These are translated into the RReil statements $\underline{\mathit{if}}\, f\, \underline{\mathit{then}}\, [\![i_1]\!]\, \underline{\mathit{else}}\, ;\, ;\, \underline{\mathit{if}}\, f\, \underline{\mathit{then}}\, [\![i_2]\!]\, \underline{\mathit{else}}\, ;$ which ideally should be simplified to $\underline{\mathit{if}}\, f\, \underline{\mathit{then}}\, [\![i_1]\!];\, [\![i_2]\!]\, \underline{\mathit{else}}\, ;$. Without this optimization, a static analysis will compute a join of the unrelated states of the $\underline{\mathit{then}}$- and $\underline{\mathit{else}}$-branches of the first $\underline{\mathit{if}}$-statement. The thereby incurred loss of precision is particularly problematic for the TSL approach since each instruction is executed on a single domain that, in general, will not be able to join two states without loss of precision.

6 Empirical Evaluation

We measured the impact of our dead-code elimination on a linear sweep disassembly of standard Unix programs. Each basic block, that is, a sequence of Intel instructions up to the next jump, is translated into semantics. Column 'native loc' in Fig. 4 lists the number of native instructions found in its *.text* section of each benchmark. Column 'non-opt' shows that the increase in number of lines of code (loc) is about 6-fold. The next triple column 'single' shows how performing liveness analysis and dead code elimination on the semantics of a single instruction reduces the size by about 7%. Performing this optimization on the

program	native	non-opt		intra			inter		
	loc	loc	time	loc	time	red	loc	time	red
bash	144k	907k	9.69s	640k	3.06m	29.53%	454k	7.85m	49.98%
cat	6685	39k	0.39s	28k	8.11s	29.87%	21k	18.44s	45.55%
echo	2742	15k	0.15s	11k	3.20s	28.74%	8323	7.09s	46.25%
less	21k	152k	1.40s	105k	30.76s	30.93%	61k	75.18s	59.90%
ls	15k	106k	0.97s	66k	21.55s	37.96%	49k	52.49s	53.76%
mkdir	6870	45k	0.41s	29k	9.27s	35.31%	21k	21.77s	52.71%
netstat	15k	86k	0.82s	63k	17.01s	26.04%	53k	36.37s	38.51%
ps	13k	68k	0.69s	45k	14.71s	33.52%	40k	29.49s	41.51%
pwd	3330	19k	0.18s	14k	3.87s	27.02%	11k	8.50s	43.11%
rm	7903	47k	0.48s	33k	9.59s	30.09%	25k	23.13s	47.01%
sed	8608	54k	0.54s	37k	11.10s	31.36%	28k	26.80s	48.55%
tar	50k	317k	3.21s	215k	64.11s	32.33%	161k	2.68m	49.23%
touch	7506	47k	0.49s	31k	9.40s	34.43%	23k	24.87s	50.71%
uname	2684	15k	0.15s	11k	3.11s	27.78%	8288	6.82s	45.19%
Xorg	346k	2081k	23.36s	1408k	7.00m	32.33%	1067k	18.26m	48.70%

Fig. 4. Evaluating the reduction of the IR due to dead-code optimization (considering succeeding blocks), showing x86 and IR instructions. All times were obtained on an Intel Core i5 running at 3.20Ghz.

whole basic block yields the result in column ‘intra’ where already a third of the instructions is identified as redundant. The ‘inter’ columns show the result of computing the liveness information as described in Sect. 5.1: in case the decoded basic block ends in a direct jump, the (one or two) basic blocks that are branched-to are decoded, translated and their set of live variables is computed. Using this refined liveness set, the dead code optimization is able to remove between 40% and 60% of lines relative to the non-optimized semantics. Thus, by applying dead code elimination using the information in the neighboring basic blocks only, our semantics of Intel instructions is roughly 3 times larger than the size of the native disassembly.

In order to compare our translation into RReil with the TSL approach [6] where a bespoke abstract transformer is generated for each native instruction, consider again column ‘single’ of Fig. 4. Since this column shows the reduction when only considering the semantics of a single instruction, it provides an estimate of how many abstract transformers in a TSL translation a standard compiler can remove due to dead code elimination. While the TSL translations are optimized otherwise, it is questionable if this can rival the effect of removing not 7%, but around 50% of instructions, as our inter-basic block analysis does.

The running times of our current implementation are dominated by the liveness analysis. We are confident that we can reduce the overhead of our optimizations considerably by using better data structures. However, even with our current implementation, it likely that any non-trivial analysis will benefit from the reduction of the IR more than it will suffer from the additional overhead of our liveness analysis. Indeed, for code that is not self-modifying, all basic block could be translated up-front and their translation be re-used.

Given these benefits, we hope that our open-source[1] GDSL toolkit becomes an attractive front-end for any analysis targeting executable programs.

7 Future Work

Future work will extend our toolkit with decoders and translations for other architectures. In particular, it would be interesting to mechanically translate the verified bit-level ARM semantics [5] into RReil. Moreover, given that an analysis that features a GDSL front-end can handle any architecture specified in GDSL, we hope for contributions from the community to further extend the range of architectures that GDSL offers. GDSL would also lend itself for defining semantics besides the RReil value semantics, namely energy or timing semantics.

In the long run, we hope that the GDSL toolkit will become the preferred choice for analyzing machine code, thereby replacing proprietary decoders (such as the popular xed2 decoder from Intel's PIN toolkit [2]) that are often equipped with a minimal, application-specific semantics covering only a few instructions.

References

1. S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In *Computer Aided Verification*, LNCS, pages 165–170. Springer, 2011.
2. Intel Corp. xed2. `http://www.pintool.org`, 2012.
3. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *International Symposium on Programming*, pages 106–130, Paris, France, April 1976.
4. T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest, Canada, 2009.
5. A. Fox and M. O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258, Edinburgh, UK, 2010. Springer.
6. J. Lim and T. Reps. A System for Generating Static Analyzers for Machine Instructions. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *LNCS*, pages 36–52. Springer, 2008.
7. N. Ramsey and M. F. Fernández. Specifying Representations of Machine Instructions. *Trans. of Programming Languages and Systems*, 19(3):492–524, May 1997.
8. A. Sepp, J. Kranz, and A. Simon. GDSL: A Generic Decoder Specification Language for Interpreting Machine Language. In *Tools for Automatic Program Analysis*, ENTCS, Deauville, France, September 2012. Springer.
9. A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M.Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, October 2011. IEEE Computer Society.
10. A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion. In *Partial Evaluation and Program Manipulation*, SIGPLAN, Rome, Italy, January 2013. ACM.

[1] The toolkit is available at `https://bitbucket.org/jucs/gdsl/src`