The GDSL toolkit: Generating Frontends for the Analysis of Machine Code

Axel Simon
Technical University Munich
axel.simon@in.tum.de

Julian Kranz
Technical University Munich
kranzj@in.tum.de

Abstract

Any inspection, analysis or reverse engineering of binaries requires a translation of the program text into an intermediate representation (IR) that conveys the semantics of the program. To this end, we propose a domain specific language called GDSL (Generic Decoder Specification Language) that facilitates the translation from byte streams to instructions and from there to other intermediate representations. We present the GDSL toolkit, containing a compiler from GDSL to C, instruction decoders (currently for Intel x86 and Atmel AVR), translations to semantics, and optimizations of the semantics. Other processors, semantics and optimizations can be added, thereby providing a common platform for building frontends for the analysis of binaries. The emitted C code is human-readable and outperforms hand-written code such as the XED decoder shipped with the Intel Pin toolkit.

Categories and Subject Descriptors [*Security and privacy*]: Software reverse engineering; [*Security and privacy*]: Intrusion/anomaly detection and malware mitigation; [*Theory of computation*]: Program analysis

General Terms binary analysis, instruction decoding, intermediate representation

Keywords Intel x86, RReil

1. Overview

The analysis of executable code gains increasing importance due to the need to understand and debug malware, closed-source software and compiler-induced canaries, etc. The reconstruction of assembler instructions from an input (byte) sequence that comprise the program is the first step towards these analyses. The second step is to map each statement to a meaning which may be a value-, timing- or energy semantics, etc., depending on the goal of the analysis. Both aspects are commonly addressed by writing an architecture-specific decoder and a translator to some intermediate representation using the implementation language of the analysis. The presented GDSL toolkit provides the infrastructure to specify decoders and translations to semantics using a domain specific language (DSL) that can be compiled into the programming language of existing analysis tools. To this end, we present GDSL and motivate its design by the task of specifying decoders for Intel x86.

The incentive for creating a DSL to specify the decoder and semantics of assembler instructions was a discussion at a Dagstuhl seminar on the analysis of executable code. Here, it was realized that many research groups implemented prototype analyses using an architecture specific decoder and a hand-written semantic interpretation. Besides duplication of work, these approaches are usually incomplete, are bound to one architecture and are hard to maintain since their representation of instructions is geared towards a specific project. In the presence of recurring extensions to instruction sets and the need to adapt an analysis to new targets – such as virtual machines contained in malware – maintainability and simplicity of decoder specifications is of increasing importance.

To this end, it is desirable to group instructions logically or, when converting a manufacturer's manual, in alphabetical order; we call this mnemonic-centric specification. For the sake of efficiency, however, a decoder must make a decision based on the next value from the input sequence (opcode-centric dispatch) which precludes testing opcode patterns one after the other. While a classic scanner generator like `lex` can convert a mnemonic-centric specification to an opcode-centric decoder, it allows and encourages overlapping patterns. Consider the following `lex` scanner specification:

```
1 while|do|switch|case  { printf("keyword %s",yytext); }
2 [a-zA-Z][a-zA-Z0-9]*  { printf("ident %s",yytext); }
```

Here the patterns for the keywords and the identifier are overlapping: the input `while` matches both rules. In this case, `lex` uses the rule that appears first in the specification file. Thus, a keyword is returned. Overlapping patterns are desirable in a scanner specification since they improve readability and conciseness. In an instruction decoder, however, overlapping patterns are undesirable since the sequence in which the rules are written starts to matter which, in turn, precludes a mnemonic-centric specification. Hence, a DSL for maintainable decoder specifications must provide a concise way of writing non-overlapping patterns to exactly match an instruction.

Another challenge is the processing of non-constant bits of an instruction that are used to specify parameters. Since parameter bits often follow re-occurring patterns, an abstraction mechanism is required to keep the specification concise. For example, the mod/rm-byte in Intel x86 instructions follows many opcodes and determines which register and memory address to use as argument. Figure 1 shows an excerpt of the Intel manual where the first column shows the two bytes that together form an instruction. The second byte `/r` is the mod/rm-byte that determines which 8-bit register `r8` and which pointer `r/m8` stand for. Within our decoder specification language, we define functions `r/m8`[1] and `r8`

PPREW '14, January 25th, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2556464.2559596

[1] We allow / as part of an identifier to accommodate the Intel nomenclature.

Opcode	Instruction	Description
00 /r	ADD r/m8,r8	Add r8 to r/m8.
28 /r	SUB r/m8,r8	Subtract r8 from r/m8.

Figure 1. Two typical instructions in the Intel x86 manual.

to generate the arguments of an instruction. The contents of the mod/rm-byte are read by a sub-decoder named `/r` that stores the read byte in an internal decoder state. This sub-decoder can be reused in the decoder for ADD and SUB:

```
1 val decode [0x00 /r] = binop ADD r/m8 r8
2 val decode [0x28 /r] = binop SUB r/m8 r8
```

Here, the decoder `decode` is declared as reading `0x00` (resp. `0x28`) from the input before running the sub-decoder `/r`. The `binop` function merely executes `r/m8` and `r8` (which access the values stored in the internal state by `/r`) and applies the results to the passed-in constructor (here `ADD` and `SUB`), thereby constructing an instruction with two arguments. By using sub-decoders such as `/r` that communicate via the internal state, our `decode` function comes very close to the specification in the Intel manual shown in Fig. 1.

Since our DSL is an ML-like functional language, it is powerful enough to describe all parts of a decoder, even `r/m8` and `r8` that are often hand-coded primitives in other decoder frameworks. This comprehensive approach allows for adding instructions that use a different encoding for arguments by defining new sub-decoders. In general, new abstractions can be created for other instruction sets.

Translation to a Semantics. Given a decoder taking byte streams to instructions, it is possible to create a static analysis by associating a semantic action with each instruction. However, since the number of processor instructions is very large, it is likely that any pragmatic approach will be restricted to handling only the most common instructions while ignoring or grossly over-approximating the less common ones. An alternative approach is to translate the processor instructions into a small intermediate representation (IR) and implement an analysis for the IR. This approach has the advantage that new instructions and even other processors can be added without extending the analysis itself. As a functional language, GDSL provides algebraic data types and pattern matching which facilitates the symbolic computation involved in compilation processor instructions to an intermediate representation. We demonstrate how this translation can be implemented by translating Intel x86 instructions into RReil, an IR that was designed to concisely express the value semantics of assembler instructions [14].

Optimizations. An inherent problem of translating all the effects of an instruction is that many IR computations are necessary to state how CPU flags are affected. Such a precise semantics is over-the-top for, say, a simple taint analysis that merely tracks how values flow between CPU registers. However, more expressive analyses are also slowed down unnecessarily since most flags are usually never read. One idea to make the analysis of the generated IR more efficient is to optimize the translated code before analyzing it. The idea is illustrated in Fig. 2. Here, the translation of the two Intel x86 instructions `cmp eax, ebx; jl tgt` is shown. The result is a sequence of RReil assignments `x =:n e` where the n is the number of bits written to x. The binary operators $<_s$`:n`, $\leq_s$`:n`, $=_s$`:n`, and $<_u$`:n`, $\leq_u$`:n`, $=_u$`:n` implement signed and unsigned comparisons, respectively, between arguments of n bits. Their result is always a one-bit value. The last statement is a conditional branch that redirects execution to `nxt` and `tgt`, depending on the one-bit value `LTS`. The figure shows how standard compiler optimizations such as a dead code elimination and a forward expression substitution can significantly reduce the size of the IR. Given that these optimizations mainly traverse and transform the IR, the GDSL language is also a good choice for implementing them. Moreover, since these optimizations are written for RReil, any processor front end that translates to RReil can benefit from them.

Usability. Using a domain-specific language to implement the aforementioned tasks is only beneficial if learning and using the language saves time over implementing the same functionality using an off-the-shelf programming language. With respect to the decoder, the advantages of separating the various rules from the actual implementation has already been highlighted. For writing the auxiliary decoding functions, the semantic translations and the optimizations, a functional language is desirable as it results in more compact and maintainable code than using an imperative language. Moreover, GDSL uses a sophisticated type inference, thereby avoiding most of the complexity of type annotations [15]. Pure functional languages such as Haskell are deemed to be difficult to debug and rather slow. In particular, purity requires that an update of a field in a record must copy the whole record since the record may be accessible via other variables. The GDSL language uses a single state monad where the state is a record. The compiler can easily determine that no other reference to the monadic state exists and, hence, replace each update to the monadic record state with a destructive update. Other techniques such as unboxing [10] render the emitted C code close to human-written C code. Moreover, the structure of the emitted C code is close to the input functional program, thereby enabling the user to debug and to profile a GDSL program at the C level. Since the emitted code resembles hand-written C, the C compiler is able to perform thorough optimizations on the code. Indeed, our x86 decoder outperforms the XED decoder shipped with the Intel Pin toolkit.

Existing Frontends. The goal of the GDSL toolkit is to provide a platform for generating frontends for various processors, possibly with other IRs besides the RReil value semantics (e.g. energy or timing semantics) and with output formats besides C. Currently, the toolkit ships with a full 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 they require new constructs in the RReil IR that have not yet been finalized. Many of the remaining undefined instructions would have to be treated as primitives since 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 second frontend for Atmel AVR microcontrollers has a semantic translation for all instructions. The instructions of this architecture are 16 bit wide and serve as a testbed for our compiler which generates efficient cascades of case-statements rather than a single case-statement over 2^{16} input words. Future work will address the addition of other popular platforms. In particular, we envisage to semi-automatically translate a proven decoder and semantic translation for the ARM architecture [9]. Besides our own efforts, we hope to see contributions from the community that extend the GDSL toolkit with other architectures, semantics and optimizations.

The remainder of the paper illustrates in more detail how an instruction decoder and the translation to semantics is specified. In addition, we present the generated C code, thereby illustrating the structural resemblance between the concise GDSL input and the C output. Specifically, the next section presents a fragment of the x86 instruction decoder. Section 3 describes the corresponding C code. The translation to the RReil IR and optimizations of the IR are sketched in Sect. 4. Section 5 presents an experimental evaluation before Sect. 6 discusses related work.

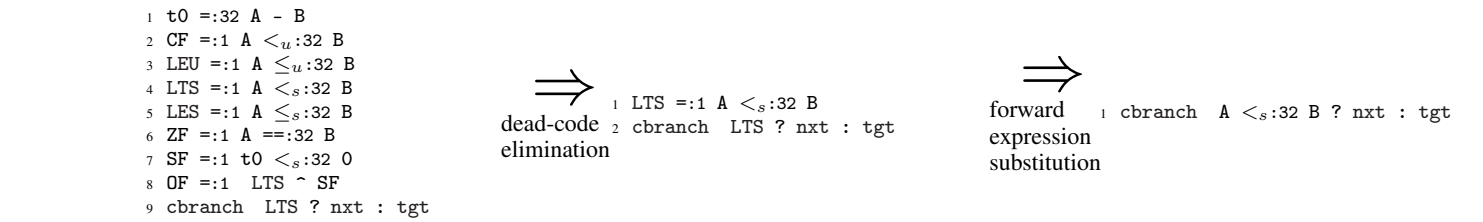

Figure 2. Translation of the native Intel instructions `cmp eax, ebx; jl tgt` into RReil and applying optimizations. Here, `LEU = CF | ZF`, `LTS = SF ^ OF` and `LES = SF ^ OF | ZF` are *virtual flags*, that is, variables that are not present in the processor [14]. Note that this example is idealized since the removed flags may not actually be dead if they are used after the branch instruction.

```
1  # declare a datatype for instructions
2  type insn = ADD of {opnd1:opnd,opnd2:opnd}
3            | INC of {opnd1:opnd}
4
5  # declare a datatype for operands of an instruction;
6  # immediates, scaled/offset operands omitted
7  type opnd =
8     REG of register # declaration of register omitted
9   | MEM of memory   # declaration of memory omitted
10
11 val decode [0x00 /r] = binop ADD r/m8 r8
12 val decode [0x01 /r] = do
13    # query the $opndsz flag of the monadic state
14    opndsz <- query $opndsz;
15    if opndsz then binop ADD r/m16 r16
16              else binop ADD r/m32 r32
17 end
18
19 # a sub-decoder reading one byte and storing
20 # 2, 3 and 3 bits of it in the monadic state
21 val /r ['mod@.. reg/opcode@... rm@...'] =
22   update @{mod=mod, reg/opcode=reg/opcode, rm=rm}
23
24 # a function that generates two-argument instructions;
25 # here giveOpX are monadic functions
26 val binop cons giveOp1 giveOp2 = do
27   op1 <- giveOp1;
28   op2 <- giveOp2;
29   return (cons { opnd1=op1, opnd2=op2 })
30 end
31
32 # a typical function passed as argument to binop:
33 # it queries the value of the reg/opcode field in the
34 # monadic state and returns the appropriate AST in
35 # form of an algebraic data type, here a register
36 val r16 = do
37    r <- query $reg/opcode;
38    case r of
39       '000': return (REG AX)
40     | '001': return (REG CX)
41     | '010': return (REG DX)
42     | '011': return (REG BX)
43    end # other cases omitted
44 end
```

Figure 3. A minimal decoder for Intel x86 instructions.

2. Specifying Decoders in GDSL

This section illustrates the GDSL language by presenting a fragment of an Intel x86 instruction decoder. The domain-specific part of the language are *decoder declarations* featuring a pattern-match syntax that implicitly reads from the input byte stream and that allows calls to other decoders [13]. These declarations are in addition to a pure functional language with flexible records and a built-in state monad. We illustrate the various aspects with an example.

```
1  val decode = do
2    tok <- consume8;
3    case tok of
4      '00000000' : do
5        /r;
6        binop ADD r/m8 r8
7      end
8    | '00000001' : do
9        /r;
10       opndsz <- query $opndsz;
11       if opndsz then binop ADD r/m16 r16
12                 else binop ADD r/m32 r32
13     end
14   end
15 end
16 val /r = do
17   tok <- consume8;
18   rm <- slice tok 0 3;
19   reg/opcode <- slice tok 3 3;
20   mod <- slice tok 6 2;
21   update @{mod = mod, reg/opcode = reg/opcode, rm = rm}
22 end
```

Figure 4. Desugaring the decoders.

The GDSL program in Fig. 3 declares two algebraic data types, one to represent instructions and one for the arguments of the instructions. It then defines two decoders, namely `decode` and `/r`, which use the special pattern syntax `[...]` to indicate that they read from the internal byte stream. This pattern syntax is internally desugared to the code in Figure 4. Here, the primitive `consume8` is used to extract a byte from the input stream over which a `case`-statement dispatches. The pattern `0x00` is translated into the bit-pattern `'00000000'` which is a built-in data type containing the bit-string and its size. The use of the sub-decoder `/r` in both patterns is translated into a call to `/r` in lines 5 and 9, before the right-hand-side of the decode rules are evaluated. In `/r` another 8-bit token is consumed and dissected into 3, 3 and 2 bits using a primitive function `slice`. These patterns correspond to the wildcard patterns `..` and `...` in line 21 of Fig. 3 that match two and three bits of any value. The syntactic sugar for decoding patterns is very powerful in practice as the compiler can re-order cases and decompose pattern matches with wildcard bits into cascades of case-statements. In the example, all bits are wildcard bits and the case-statement in `/r` is removed. The remaining code in Fig. 4 resembles that in Fig. 3.

In order to detail the remaining code, we illustrate the use of records in GDSL. A record is a set of field/value pairs. Records are flexible in the sense that the set of fields may change. Suppose that `r` denotes the record `{f=42, g="g"}`. An update `@{g='01', h="h"}` is a function that replaces or adds fields to a record, that is, evaluating `@{g='01', h="h"} r` yields the record `{f=42, g='01', h="h"}`. Note that an update can change the type of a field. A field `f` is extracted from a record `r` using `$f r` where `$f` is called a field

GDSL	::=	*val id id** = **Expr**	function decl.
	|	*val id* [**TokPat***] = **Expr**	decoder decl.
	|	*val id* [**TokPat***] (| **Expr** = **Expr**)$^+$	
TokPat	::=	*hex-num* | *id* | ' **BitPat*** '	single token
BitPat	::=	**BitStr** (| **BitStr**)*	bit patterns
	|	*id* @ **BitStr** (| **BitStr**)*	pattern binding
BitStr	::=	(*0* | *1* | .)$^+$	bit string
Action	::=	*id* ← **Expr** ; **Action**	monadic stmts
	|	**Expr** ; **Action**	
	|	**Expr**	
Cases	::=	**CasePat** : **Expr** ; **Cases** | ε	body of case
CasePat	::=	' **BitStr** ' | *num*	
	|	*con id* | *con*	
Expr	::=	*let* **Core** *in* **Expr** *end*	binding
	|	*if* **Expr** *then* **Expr** *else* **Expr**	bifurcation
	|	*case* **Expr** *of* **Cases** *end*	conditional
	|	**Expr Expr**$^+$	function appl.
	|	{ (*field* = **Expr**)* }	record constant
	|	@ { (*field* = **Expr**)* }	record update
	|	$*field*	record selector
	|	*query* **Expr** | *update* **Expr**	monadic actions
	|	*do* **Action** *end*	monadic seq.
	|	" *string* " | ' (*0* | *1*)* ' | *num*	constants
	|	*con* | *id*	constructor/var

Figure 5. The GDSL language without data type declarations.

selector. For instance, `$f r` yields `42`. Note that both, `@{h="h"}` and `$h` are functions that take a record as argument.

Records are a powerful feature when combined with the built-in state monad of GDSL. It borrows the `do ...` syntax from Haskell to specify sequences of monadic actions. The actions of the state monad can be thought of as functions taking the state and returning a value together with a new, possibly modified state. GDSL provides three primitive monadic actions from which larger monadic actions can be constructed using the `do` notation: `query f` applies `f` to the internal state and yields the result, `update f` updates the internal state `s` to `(f s)`, and `return v` merely yields `v`. Note that `return` does not alter the control flow but only serves to return the value `v` as the result of a monad. The initial state of the GDSL monad is the empty record `{}`. An action `update @{f=7}` will set the field `f` in the monadic state to `7` whereas `query $f` is an action that returns the current value of `f` in the monadic state. These two operations are the key to separating the dissection of the input stream from the generation of operands. For instance, line 22 in Fig. 3 sets the `reg/opcode` field in the monadic state which is later extracted using `query $reg/opcode` in line 37. Another example is the extraction of the `opndsz` field from the monadic state in line 14. This field is set before `decode` is called, its value depends on certain instruction prefixes that the x86 architecture defines.

The grammar of GDSL is shown in Fig. 5. A toplevel declaration is either a function, a decoder, or a decoder with guarded right-hand sides [13] which are similar to the guards in Haskell. A pattern consists of tokens, each token **TokPat** may either be a hexadecimal number, a call to another decoder or a bit pattern **BitPat**. The latter is formed of sequences of 0,1 and the wildcard `.` with the option of binding the matched result to an identifier as in line 21 of Fig. 3. The body of a function or a decoder is an expression **Expr** whose definition is quite standard with the exception of the previously defined constructs for records and monadic actions. The base types of GDSL are integers, strings and bit vectors of fixed size. For simplicity, any Boolean expression returns a bit vector containing one bit that denotes the truth value.

```
1  typedef struct { obj_t opnd1; obj_t opnd2; } struct1_t;
2  typedef struct { int_t tag; struct1_t payload; }
3                                              con_struct1_t;
4  static inline obj_t constructor_ADD(struct1_t arg) {
5    return alloc_con_struct1((con_struct1_t){1/* ADD */,
6                                                arg});
7  }
8  obj_t decode() { /* decode */
9    int_t tok = consume8();
10   switch (slice(tok, 0, 8)) {
11     case 0: /* '00000000' */ {
12       _slash_r();
13       return binop(constructor_ADD,r_slash_m8,r8);
14     }; break;
15     case 1: /* '00000001' */ {
16       _slash_r();
17       opndsz = s->state.opndsz;
18       if (opndsz)
19         return binop(constructor_ADD,r_slash_m16,r16);
20       else
21         return binop(constructor_ADD,r_slash_m32,r32);
22     }; break;
23   };
24 }
25 static void _slash_r() { /* /r */
26       int_t tok, rm, reg_slash_opcode, mod;
27       tok = consume8();
28       rm = slice(tok, 0, 3);
29       reg_slash_opcode = slice(tok, 3, 3);
30       mod = slice(tok, 6, 2);
31       s->state.mod = mod;
32       s->state.reg_slash_opcode = reg_slash_opcode;
33       s->state.rm = rm;
34 }
35 static inline obj_t constructor_REG(int_t arg) {
36   return alloc_con_int((con_int_t){3/* REG */, arg});
37 }
38 static obj_t r16() {                          /* r16 */
39   int_t r = s->state.reg_slash_opcode;
40   switch (slice(r, 0, 3)) {
41     case 0: { return constructor_REG(CON_AX); }; break;
42     case 1: { return constructor_REG(CON_CX); }; break;
43     case 2: { return constructor_REG(CON_DX); }; break;
44     case 3: { return constructor_REG(CON_BX); }; break;
45   };
46 }
47 static obj_t binop(obj_t (*cons)(struct1_t),/* binop */
48              obj_t (*giveOp1)(),obj_t (*giveOp2)()) {
49   obj_t op1 = giveOp1();
50   obj_t op2 = giveOp2();
51   return cons((struct1_t){.opnd1=op1,.opnd2=op2});
52 }
```

Figure 6. The generated C code of the decoders.

3. Translation to C

After the decoder patterns are desugared as in Fig. 4, the GDSL compiler generates an intermediate imperative language that is optimized before it is translated to C. The optimizations attempt not to alter the structure of the input program in order to enable easy debugging and profiling of the resulting code. For instance, no inlining or specialization of functions is performed so that each expression in the GDSL code has one corresponding expression in the C code. The result is a close resemblance between the input in Fig. 3 and the C code in Fig. 6. Note that this close correspondence is only possible if the program does not exploit the whole expressivity of the source language (which is usually the case). For instance, any argument to `binop` could have been a partially applied function, in which case a closure has to be

a)
```
1  val sem-add x = do
2    sz <- sizeof2 ($opnd1 x) ($opnd2 x);
3    a <- lval sz ($opnd1 x);
4    b <- rval sz ($opnd1 x);
5    c <- rvals Signed sz ($opnd2 x);
6    t <- mktemp;
7    add sz t b c;
8    emit-add-adc-flags sz (var t) b c (imm 0) '1';
9    write sz a (var t)
10 end
```

b)
```
1  T0 =:16 (A + B)
2  A =:16 T0
3  T0 =:16 (D + C)
4  T1 =:1 T0.15 ^ D.15
5  T2 =:1 T0.15 ^ C.15
6  OF =:1 T1 & T2
7  SF =:1 T0 <s:16 0
8  ZF =:1 T0 ==:16 0
9  CF =:1 T0 <u:16 D
10 PF =:1 T0.7 ==:1 T0.6
11 PF =:1 PF ==:1 T0.5
12 PF =:1 PF ==:1 T0.4
13 PF =:1 PF ==:1 T0.3
14 PF =:1 PF ==:1 T0.2
15 PF =:1 PF ==:1 T0.1
16 PF =:1 PF ==:1 T0
17 T3 =:1 T0.4 ^ D.4
18 AF =:1 T3 ^ C.4
19 LEU =:1 CF | ZF
20 LTS =:1 SF !=:1 OF
21 LES =:1 LTS | ZF
22 D =:16 T0
```

Figure 7. The translator function a) and a translation result b)

passed rather than the addresses of a C function as done in line 13,19, and 21. Also, the arguments of the `REG` constructor in lines 41-44 are preprocessor-defined constants rather than pointers to heap allocated constructors which would be required if one of the registers had an argument. The basis of these simplifications is an unboxing optimization based on a monomorphic type inference. Unlike standard unboxing [10], our optimization is able to infer, for instance, that a function parameter requires no closure and can therefore be passed as a simple C function pointer. The resulting C code is not only close to human-written code, it is also amenable to optimizations in off-the-shelf C compilers.

4. RReil Intermediate Representation

Many intermediate representations for giving semantics to assembler instructions exist, each having its own design goals such as minimality [5, 8], mechanical verifiability [9], reversibility [12], or expressivity [5, 14]. Our own RReil IR [14] 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 [8, 9, 11]. Note that GDSL can be used to emit IRs other than RReil and we welcome any such contributions.

Figure 7a) shows the GDSL code that translates an x86 `ADD` instruction into RReil. The argument `x` contains the payload of the `ADD` constructor, namely a record containing an `opnd1` and `opnd2` field. Consider applying the function to the two instructions `ADD AX,BX; ADD CX,DX`, resulting in the RReil code in Fig. 7b). Here, we applied a dead code elimination that removes all flag computations of the first instruction, leaving lines 1 and 2 for the translation of `ADD AX,BX`. We illustrate how the code for `ADD CX,DX` is generated where the dead code elimination had no effect. Line 2 of Fig. 7a) computes the size of the arguments (here: 16) by examining both arguments in case one of them is a constant. The first operand is translated into an l-value `a` and an r-value `b`. Line 5 computes an r-value of the second operand where a constant is sign-extended to 16 bits (irrelevant for `DX`). Line 6 creates a temporary variable `t` to which the sum of `a` and `b` is written to by the `add` function in line 7. This function stores the corresponding RReil assignment inside the monadic state, thereby alleviating the programmer to explicitly pass around a list of generated statements.

decoder	dec. errors	time	x86 insns	exe size
XED 2.12	488	1.2s	2,667,829	1344kb
XED 2.11	0	1.7s	2,667,248	1024kb
GDSL 0.9.0	0	1.1s	2,667,248	295kb

Figure 8. Decoding performance compared to Intel's XED.

This RReil assignment is shown in line 3 of Fig. 7b). Storing the result in a temporary register is necessary in order to compute the various flags in line 8 that depend on the input values `b`, `c` and the output value `t`. For instance, the overflow flag `OF` is the disjunction of the exclusive-or of the most significant bits of the result `T0` and the two inputs `C` and `D` as computed in lines 4 to 6 in b). The parity of the lower eight bits of the result is computed in lines 10 to 16, followed by the computation of `AF` and the virtual flags (see Fig. 2). Line 9 in Fig. 7a) generates code to write the temporary register to the target register `a`, resulting in line 22 of Fig. 7b).

Although the presented semantics for `ADD` relies on many other auxiliary functions, it is interesting to note that these building blocks can be reused in the translation of other instructions. In particular, vector instructions can be translated by passing the building block of the base operation to a function that applies this operation, say, 8 times at different bit offsets. Thus, a functional language where higher order functions can seamlessly be passed as arguments is very useful to write semantic translations. Observe that a single x86 instruction such as `ADD` often requires several decoding rules to cater for different argument types. In our x86 specification, the overall size of the semantic translation is about the size of the decoder declarations. Given that we have implemented about half the x86 instructions, the semantic translation per instruction is only about twice the size of its decoder.

Observe that eliminating dead assignments in the RReil semantics of `ADD AX,BX; ADD CX,DX` is a big win since all flag computations of the first addition can be removed since they are overwritten by the second addition. We have implemented this liveness analysis in GDSL whose performance we evaluate in the next section. In the future, we envisage to also perform other optimizations on RReil which will benefit other architectures translating to RReil.

5. Experimental Evaluation

We evaluated our toolkit with respect to two main criteria: the performance of our decoder compared to the XED decoder from the Intel Pin toolkit [7] and the effectiveness of different dead code eliminations on the IR. We chose a decoder written by Intel under the assumption that it is faithful to the decoding performed in x86 hardware. Moreover, it is one of the fastest openly available x86 decoders [13]. Figure 8 compares the performance of XED with our GDSL decoder using the `clang` binary. Interestingly, the more recent version of the decoder is unable to decode all of the instructions which may be a reason for its significant performance gain compared to the older version. Yet, the decoder generated from the GDSL specification decodes all instructions while being no slower than XED.

Figure 9 contains measurements for three different approaches to eliminate dead code. The first approach removes dead code on a per-instruction basis while the second approach considers a basic block at a time. The third approach additionally considers the successor block(s) if known. All data has been collected using the clang binary. The first column of the table lists the size of the resulting RReil code while the second column shows the time needed to generate it. The last two columns compare the size of the optimized semantics to the original RReil and binary machine code, respectively. For this, the third column gives the growth factor,

	IR stmts	time	growth	IR red.
decoding and translation	19M	5.5s	7.0	-
optimizing instructions in isolation	17M	93s	6.4	9%
optimizing basic blocks in isolation	12M	68s	4.6	35%
optimizing basic blocks considering successors	9M	174s	3.4	52%

Figure 9. Performance and evaluation of the decoding, translation, and optimization of the 2,667,248 instruction clang binary.

GDSL program	heap residency avg.	max.
x86 decoder + printing	1.0kb	2.5kb
x86 decoder + translator + printing	7.7kb	66kb
x86 decoder + translator + liveness	206kb	67Mb
x86 decoder + translator + lookahead liven.	497kb	67Mb

Figure 10. GDSL program performance.

that is, the number of RReil statements divided by the number of Intel instructions. The fourth column states the percentage of the removed RReil code compared to the non-optimized translation.

As visible in the table, the optimizer is currently disproportionally slow compared to the decoder and semantic translator. Furthermore, the single instruction liveness analysis consumes more time than the basic block-wise analysis, probably due to a costly initialization of the liveness analysis. Future work should address these performance issues.

Finally, Fig. 10 evaluates the heap residency of different GDSL programs. The maximum heap residency is relevant since the heap of a GDSL program can currently only be recycled once a basic block has been decoded, translated and optimized. We again use the `clang` binary as input. Depending on the optimization level, the heap has to hold the data structures necessary to optimize several basic blocks, the one under inspection and its successors. The two columns contain the average and the maximum value. While the more complex optimizations have a larger maximum footprint, it is reasonable for processing off-the-shelf binaries using current hardware.

6. Related Work and Conclusion

Over the years, many projects have addressed the analysis of executable code. In all cases, some sort of decoding and semantic interpretation is necessary. These endeavors often follow a pragmatic approach in handling the most common instructions which may suffice for test programs but falls short for general binaries.

Most decoder libraries for the Intel x86 instructions generate or use tables for mapping opcodes to instructions. The decoding of prefixes and arguments, however, is usually hand-coded [1–4]. One notable exception is *SLED* [12], a specification language for encoding and decoding, which is a comprehensive specification language similar to GDSL. *SLED* specifies mnemonics using opcode-centric tables. Thus, the specification in their approach cannot follow the Intel manual, thereby making it harder to maintain the decoder. Moreover, to our understanding, amending the x86 specification [12] to accept instructions with superfluous prefixes is not possible.

Another approach was taken by Fox et al. [9]. In their work they describe a formal model of the complete ARMv7 instruction set encoded in the HOL4 proof system. The model directly operates on word sequences, as even the decoding logic is specified in the proof system. Besides the mere decoding logic, a full semantics of the ARMv7 instruction set is also provided whose fidelity against an ARMv7 implementation was proved. Future work will address a mechanic translation of this specification to GDSL.

Several intermediate representations have been proposed to specify instruction semantics [5, 6, 14]. The expressed goal of GDSL is to allow a processor instruction to be translated into any IR that suits the intended analysis. Using a common framework can help to make the various intermediate representations comparable and usable in various analysis frameworks. Lim et al. have proposed to compile an abstract transformer for each processor instruction in order to obtain a more efficient analysis [11]. Their argument is that synthesizing a transformer for, say, a taint analysis can eliminate most of the irrelevant computations that happen in an instruction. Fig. 9 shows that when all semantic operations are needed, only 9% of the computations is dead. Future should therefore address how a different backend to our compiler can follow their setup and to then compare the effectiveness of their approach for various analyses with the current GDSL approach of an inter-basic block optimization using a full semantics.

References

[1] BeaEngine. `http://www.beaengine.org`, 2012. Version 4.1 rev 172.

[2] distorm. `http://www.ragestorm.net/distorm/`, 2012. Version 3.1.

[3] libopcodes. `http://packages.debian.org/testing`, 2012. Package binutils-dev-2.22-6.

[4] metasm. `http://metasm.cr0.org/`, 2012. Retrieved on 2012/05/25.

[5] 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.

[6] C. Cifuentes and S. Sendall. Specifying the Semantics of Machine Instructions. In *International Workshop on Program Comprehension*, IWPC '98, Washington, 1998. IEEE Computer Society.

[7] Intel Corp. xed2. `http://www.pintool.org`, 2012.

[8] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. CanSecWest Vancouver, Canada, 2009.

[9] 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.

[10] X. Leroy. Unboxed objects and polymorphic typing. In *Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[11] 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.

[12] N. Ramsey and M. F. Fernández. Specifying Representations of Machine Instructions. *Trans. of Programming Languages and Systems*, 19(3):492–524, May 1997.

[13] 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.

[14] 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.

[15] 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.