GDSL: A Generic Decoder Specification Language for Interpreting Machine Language

Alexander Sepp, Julian Kranz and Axel Simon[1,2]

Technische Universität München
85748 Garching, Germany

Abstract

The analysis of executable code requires the reconstruction of instructions from a sequence of bytes (or words) and a specification of their semantics. Most front-ends addressing this problem only support a single architecture, are bound to a specific programming language, or are hard to maintain. In this work, we present a domain specific language (DSL) called GDSL (Generic Decoder Specification Language) for specifying maintainable instruction decoders and the translation of instructions to a semantics. We motivate its design by illustrating its use for the Intel x86 platform. A compiler is presented that generates C code that rivals hand-crafted decoder implementations.

Keywords: executable analysis, binary analysis, instruction decoder, program semantics

1 Introduction

The analysis of executable code has become a recent focus in program analysis in order to address the analysis of malware, closed-source software and to tackle compiler-induced bugs. 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 internal representation expressed in the implementation language of the analysis. The goal of our work is to build an infrastructure to specify decoders and translations to

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

[2] Email: firstname.lastname@in.tum.de

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.

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

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 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 prototypes 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 steadily increasing 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:

```
while|do|switch|case  { printf("keyword %s",yytext); }
[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 reoccurring 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 to use. 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 registers r8 and which pointer r/m8 stand for. Within our decoder specification language, we define functions r/m8[3] and r8 to generate the arguments of an instruction. The content of the mod/rm-byte are read by a sub-decoder named /r that stores the read byte in the internal decoder state. This sub-decoder can be re-used in the decoder for ADD and SUB:

```
val main [0x00 /r] = binop ADD r/m8 r8
val main [0x28 /r] = binop SUB r/m8 r8
```

Here, the decoder main is declared as reading 0x00 (resp. 0x28) from the input before running the sub-decoder /r. The binop function is a simple wrapper that executes functions r/m8 and r8 (which access the values stored by /r) and applies the results to the passed-in constructor (here ADD and SUB). By using sub-decoders such as /r that communicate via the internal state, our main decoder comes very close to the specification in Fig. 1 of the Intel manual.

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 enables users to add instructions that have not been anticipated in the original design of /r. In summary, GDSL improves over existing approaches as follows:

- Its abstraction mechanisms enable the definition of instruction decoders that are very close to the syntax used in manufacturer's manuals, thereby ensuring maintainability even by the end users of the decoder framework.
- Our specification is type checked during compilation and overlapping patterns are detected. This ensures high fidelity of the resulting decoder, especially in the presence of mistakes in the manufacturer's manuals.
- The DSL is flexible enough to accommodate a variety of architectures. Due to its general nature, it is possible to add translations from native instructions to some abstract semantics, which will enable binary analysis tools to analyse code for any architecture that is described with our framework.
- We provide a prototype compiler that generates C code which is competitive with other decoders. The specifications can be translated to other languages or used for other purposes (e.g. test generation) by writing a new backend.

After the next section presents the design of GDSL, Sect. 3 illustrates its expressiveness by detailing the decoding of Intel prefixes. Section 4 presents an evaluation of our implementation before Sect. 5 presents related work.

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

$$
\begin{array}{rcll}
\mathit{Decl} & ::= & \underline{\texttt{granularity}}\ \underline{\texttt{=}}\ \underline{\texttt{num}}\ \left[\underline{\texttt{lsbfirst}}\right] & (1)\\
 & | & \underline{\texttt{export}}\ \underline{\texttt{id}}^{*} & (2)\\
 & | & \underline{\texttt{datatype}}\ \underline{\texttt{t-id}}\ \underline{\texttt{=}}\ \underline{\texttt{con}}\ \left[\underline{\texttt{of}}\ \mathit{Type}\right]\ \left(\underline{\texttt{|}}\ \underline{\texttt{con}}\ \left[\underline{\texttt{of}}\ \mathit{Type}\right]\right)^{*} & (3)\\
 & | & \underline{\texttt{type}}\ \underline{\texttt{t-id}}\ \underline{\texttt{=}}\ \mathit{Type} & (4)\\
 & | & \underline{\texttt{val}}\ \underline{\texttt{id}}\ \underline{\texttt{id}}^{*}\ \underline{\texttt{=}}\ \mathit{Expr} & (5)\\
 & | & \underline{\texttt{val}}\ \underline{\texttt{id}}\ \underline{\texttt{[}}\mathit{TokPat}^{*}\underline{\texttt{]}}\ \underline{\texttt{=}}\ \mathit{Expr} & (6)\\
 & | & \underline{\texttt{val}}\ \underline{\texttt{id}}\ \underline{\texttt{[}}\mathit{TokPat}^{*}\underline{\texttt{]}}\ \left(\underline{\texttt{|}}\ \mathit{Expr}\ \underline{\texttt{=}}\ \mathit{Expr}\right)^{+} & (7)\\
\mathit{Type} & ::= & \underline{\texttt{int}}\ |\underline{\texttt{[}}\ \underline{\texttt{num}}\ \underline{\texttt{]}}\ |\ \underline{\texttt{t-id}} & (8)\\
 & | & \underline{\texttt{\{}}\ \underline{\texttt{field}}\ \underline{\texttt{:}}\ \mathit{Type}\left(\underline{\texttt{,}}\ \underline{\texttt{field}}\ \underline{\texttt{:}}\ \mathit{Type}\right)^{*}\underline{\texttt{\}}} & (9)\\
\mathit{TokPat} & ::= & \underline{\texttt{hex-num}}\ |\ \underline{\texttt{id}}\ |\ \underline{\texttt{'}}\ \mathit{BitPat}^{*}\ \underline{\texttt{'}} & (10)\\
\mathit{BitPat} & ::= & \mathit{BitStr}\ \left(\underline{\texttt{|}}\ \mathit{BitStr}\right)^{*} & (11)\\
 & | & \underline{\texttt{id}}\ \underline{\texttt{@}}\ \mathit{BitStr}\ \left(\underline{\texttt{|}}\ \mathit{BitStr}\right)^{*} & (12)\\
 & | & \underline{\texttt{id}}\ \underline{\texttt{:}}\ \underline{\texttt{num}} & (13)\\
\mathit{BitStr} & ::= & \left(\underline{\texttt{0}}\ |\ \underline{\texttt{1}}\ |\ \underline{\texttt{.}}\right)^{+} & (14)\\
\mathit{Expr} & ::= & \underline{\texttt{case}}\ \mathit{Expr}\ \underline{\texttt{of}}\ \mathit{Pat}\ \underline{\texttt{:}}\ \mathit{Expr}\ \left(\underline{\texttt{|}}\ \mathit{Pat}\ \underline{\texttt{:}}\ \mathit{Expr}\right)^{*} & (15)\\
 & | & \underline{\texttt{if}}\ \mathit{Expr}\ \underline{\texttt{then}}\ \mathit{Expr}\ \underline{\texttt{else}}\ \mathit{Expr} & (16)\\
 & | & \underline{\texttt{let}}\ \left(\underline{\texttt{val}}\ \underline{\texttt{id}}\ \underline{\texttt{=}}\ \mathit{Expr}\right)^{+}\ \underline{\texttt{in}}\ \underline{\texttt{Expr}} & (17)\\
 & | & \mathit{Expr}\ \mathit{Expr}\ |\ \underline{\texttt{num}}\ |\ \underline{\texttt{'}}\ \mathit{BitStr}\ \underline{\texttt{'}}\ |\ \underline{\texttt{id}}\ |\ \underline{\texttt{con}} & (18)\\
 & | & \underline{\texttt{\{}}\ \underline{\texttt{field}}\ \underline{\texttt{=}}\ \mathit{Expr}\ \left(\underline{\texttt{,}}\ \underline{\texttt{field}}\ \underline{\texttt{=}}\ \mathit{Expr}\right)^{*}\ \underline{\texttt{\}}} & (19)\\
 & | & \underline{\texttt{\$field}}\ |\ \underline{\texttt{@\{}}\ \underline{\texttt{field}}\ \underline{\texttt{=}}\ \mathit{Expr}\ \left(\underline{\texttt{,}}\ \underline{\texttt{field}}\ \underline{\texttt{=}}\ \mathit{Expr}\right)^{*}\ \underline{\texttt{\}}} & (20)\\
 & | & \underline{\texttt{do}}\ \left(\mathit{Expr}\underline{\texttt{;}}\ |\ \underline{\texttt{id}}\ \underline{\texttt{<-}}\ \mathit{Expr}\underline{\texttt{;}}\right)^{*}\ \mathit{Expr}\ \underline{\texttt{end}} & (21)\\
 & | & \underline{\texttt{update}}\ \mathit{Expr}\ |\ \underline{\texttt{query}}\ \mathit{Expr}\ |\ \underline{\texttt{return}}\ \mathit{Expr} & (22)
\end{array}
$$

Fig. 2. Syntax of the GDSL language.

2 General Language Overview

This section discusses the design of GDSL by illustrating the use of the various syntactic constructs. The general idea is that the decoder specification is an executable functional program that consumes the input sequence and produces a heap containing the abstract syntax tree (AST) that represents the recognized instruction. After the AST in the heap has been processed, the heap can be reused for decoding the next instruction, thereby avoiding the need for a garbage collector or for allocating memory with each instruction.

The grammar of GDSL is shown in Fig. 2. A file consists of a sequence of definitions given by *Decl*. The `granularity` statement can be given once and defines the size of the tokens that the decoder consumes. A token is measured in bits and is the smallest granularity that a processor reads from memory.

```
 1 granularity 8
 2 export main
 3 datatype instr = ADD of {op1:op, op2:op}
 4
 5 val binop cons giveOp1 giveOp2 = do
 6    operand1 <- giveOp1;
 7    operand2 <- giveOp2;
 8    return (cons {op1=operand1, op2=operand2})
 9 end
10
11 val /r ['mod:2 reg:3 rm:3'] =
12   update @{mod=mod, reg/opcode=reg, rm=rm}
13 val /0 ['mod:2 000 rm:3'] =
14   update @{mod=mod, reg/opcode='000', rm=rm}
15 val r/m8 = do
16    r <- query $rm;
17    return (case r of '000': Reg AL | '001': Reg BL )
18 end
19
20 val main [0x80 /0] = binop ADD r/m8 imm8
21 val main [0x00 /r] = binop ADD r/m8 r8
22 val main [0x01 /r] | $opndsz = binop ADD r/m16 r16
23                    | $rexw = binop ADD r/m64 r64
24                    | otherwise = binop ADD r/m32 r32
```

Fig. 3. Specification for decoding the Intel ADD instruction.

For Intel x86, the token size is 8 (and each instruction can have between one and fifteen tokens). For standard ARM instructions, the token size is 32 (and no instruction is longer than one token). Other processors are in between these extremes, for instance, MicroChip's PIC architecture has a token size of 14. The optional `lsbfirst` keyword states that bit sequences in decoders start with the least significant bit, a notation used for e.g. PowerPC.

The `export` keyword states which of the decoders are publicly visible to the client code. Line 3 shows the production for algebraic data types that introduce (or extend) the type `t-id` with constructors `con`. As in ML, each constructor takes zero or one argument, allowing the definition of enumerations such as `datatype register = AX | BX | CX | DX` or AST nodes such as `datatype op = Reg of register | Mem of {size : int, reg : op} | Imm8 of [8]`. Here, the argument to the `Mem` constructor is a record while `Imm8` takes a bit vector of 8 bits, written `[8]`. Bit vectors and `int` are the only basic data types with singleton bit-vectors acting as Booleans. Abbreviations for complex types can be introduced using the `type` keyword in line 4.

Productions 5, 6, and 7 introduce functions, decoders and decoders with guards, respectively. Functions and decoders differ in that functions take arguments and have exactly one definition whereas decoders read from the implicit input stream and definitions with the same name augment each other. Consider the decoder snipped in Fig. 3. Here, `binop` and `r/m8` in lines 5 and 15 are functions taking three and no arguments, respectively. In contrast, lines 11, 13 and 20 define decoders whose right-hand-side is evaluated if the token sequence in the square brackets matches the current input. Tokens can be specified in three ways (Production 10): either as a hexadecimal number (c.f. the first token of `main`), as a call to another decoder (c.f. the second token of `main`) or as a bit pattern (as used in the `/r` and `/0` decoders). Bit patterns, in turn, are enclosed in ticks and are given by Productions 11, 12, and 13:

- strings of 0,1,. (c.f. `000` in `/0`); the dot acts as a wildcard; a set of bit strings can be specified by separating them using a vertical bar, e.g. `00|01|10`
- as above, with a leading variable separated by `@`; the variable is bound to the actual bits in the input; for instance, `/0` could have been written

  ```
  val /0 [’mod:2 reg@000 rm:3’] =
    update @{mod=mod, reg/opcode=reg, rm=rm}
  ```
- a variable with a width in bits; the notation `v:3` is syntactic sugar for `v@...`; examples are `mod`, `reg` and `rm` in the decoders `/r` and `/0`

The semantics of “calling” another decoder within a token sequence is that the pattern of the called decoder is substituted where it appears and that its body is prepended to the right-hand-side of the decoder. For instance, `main [0x80 /0]` is translated internally as follows:

```
val main [0x80 ’mod:2 000 rm:3’] = do
  update @{mod=mod, reg/opcode=’000’, rm=rm};
  binop ADD r/m8 imm8
end
```

After this translation, the patterns are translated token-by-token into C `case`-statements during which overlapping patterns are detected. For token sizes larger than 8 bits, nested `case`-statements are generated.

The bodies of functions and decoders are given by the *Expr* production. Here, Productions 15, . . . ,18 give the standard constructs found in a functional language with *Expr Expr* in line 18 denoting function application. Our language allows the creation of compound values using records which are collections of field names bound to a value. Productions 19 allows the construction of new records (used in line 8 of Fig. 3). The value of a field `foo` is extracted using `$foo` which itself is a function. Thus, `$foo {foo=7}` evaluates to 7. Analogously, `@{foo=x}` is a function taking a record and setting the field `foo` to `x`. For instance `@{bar=’110’} {foo=7}` evaluates to `{bar=’110’,foo=7}`.

In order to allow for an internal state, each decoder is a monad, a concept borrowed from the pure functional language Haskell [1]. A monad is an abstract type containing a function from an input state to an output state and a result. The motivation for monads is to chain together computations that operate on a state without requiring side-effects in the language. Production 21 details the `do`-statement which threads together monadic actions whose result can be bound to an identifier. The result of the `do`-statement is that of the last action. Production 22 presents the three monadic actions of our language: `update` f applies f to the internal state (and is usually a record update); `query` f returns the result of applying f to the internal state (and is usually a record field selector); and `return` x that returns x as a result.

Besides `query`, the internal state can also be accessed using guards: the first guard of `$opndsz`, `$rexw`, and `otherwise` in line 22 that evaluates to `'1'` determines which right-hand-side is evaluated. Guards are functions taking the internal state as argument. Thus, `opndsz` and `rexw` are record fields in the internal state and `otherwise` is a function always returning `'1'`.

3 Decoding x86 Prefixes

One challenge in decoding x86 instructions is the correct handling of prefixes: they either serve to modify the following instruction or they are part of the following opcode (a so-called mandatory prefix). In the latter case, other prefixes are allowed between the mandatory prefix and the actual opcode. For example, both instruction sequences: **67 f3** `45 0f 7e d1` and **f3 67** `45 0f 7e d1` encode movq xmm10,xmm9 where `67` is an ADDRSZ prefix and `f3` is a REPNE prefix, but used here as mandatory prefix to the opcode `0f 7e`. Moreover, `45` is another "standard" REX prefix and `d1` the mod/rm byte. Confusingly, the REX prefix must immediately precede the opcode, otherwise it is ignored.

Certain instructions such as mulss, mulsd, and mulpd share the same opcode, here `0f 59`, but have different mandatory prefixes, namely `f2`, `f3`, and `66`, respectively. As a consequence, the *order* in which prefixes occur becomes important. Moreover, while the last occurrence of `f2` and `f3` determines the mandatory prefix, an occurrence of `66` is only recognized as mandatory prefix if `f2` and `f3` cannot start an instruction. A correct decoder recognizes:

66 f3 **f2** 0f 59 ff	mulsd xmm7,xmm7	Mandatory prefix: **0xf2**
66 f2 **f3** 0f 59 ff	mulss xmm7,xmm7	Mandatory prefix: **0xf3**
66 0f 59 ff	mulpd xmm7,xmm7	Mandatory prefix: **0x66**
f2 66 0f 59 ff	mulsd xmm7,xmm7	Mandatory prefix: **0xf2**

Mandatory prefixes can easily be handled in GDSL by using different decoders, depending on the last relevant prefix. We decode prefixes as follows:

```
val prefixes [0x66] = p/66
val prefixes [0xf2] = p/f2
val prefixes [0xf3] = p/f3
val prefixes [] = main
val p/66 [0x66] = p/66
val p/66 [0xf2] = p/66/f2
val p/66 [0xf3] = p/66/f3
val p/66 [] = after /66 main
val p/f3 [0x66] = p/66/f3       # f3 dominates 66
val p/f3 [0xf2] = p/f3/f2
val p/f3 [0xf3] = p/f3
val p/f3 [] = after /f3 main
val p/f3/f2 [0x66] = p/66/f3/f2 # f3/f2 dominates 66
val p/f3/f2 [0xf2] = p/f3/f2
val p/f3/f2 [0xf3] = p/f2/f3
val p/f3/f2 [] = after /f2 (after /f3 main)
... # analogous for p/f2, p/66/f2, p/66/f3, p/f2/f3,
    #               p/66/f3/f2, p/66/f2/f3
val /66 [] = continue
val /f2 [] = continue
val /f3 [] = continue
val /66 [0x0f 0x59 /r] = binop MULPD xmm xmm/m128
val /f2 [0x0f 0x59 /r] = binop MULSD xmm xmm/m64
val /f3 [0x0f 0x59 /r] = binop MULSS xmm xmm/m32
val main [...] = ...
```

The entry point that is exported to the user is `prefixes`. When reading the sequence `f3 f2 0f 59 ff`, it dispatches to `p/f3` which itself reads `f2` and enters the `p/f3/f2`. Since the next byte `0f` has no match in `p/f3/f2`, the expression `after /f2 (after /f3 main)` is executed. The `after` function calls the decoder `/f2` and, if it fails, continues with `(after /f3 main)`. The latter expression runs `f3` and, if this decoder fails, runs `main`. On our example byte sequence, the `/f2` decoder succeeds in consuming the remaining bytes `0f 59 ff` and returns the `mulsd` instruction. By construction of the prefix decoders, at most four lookups can lead to failure: one prefix decoder, `/66`, `/f2`, `/f3`. Thus, at most one byte of the sequence is examined more than once.

Observe that `after` and `continue` can be defined directly within GDSL:

```
val after fst snd = do update @{cont=snd}; fst end
val continue = do decoder <- query $cont; decoder end
```

Here, `after` stores its argument `snd` in the decoder state and executes the decoder `fst`. The `continue` function retrieves the stored decoder and dispatches to it. This completes the design of our prefix decoders.

Framework	**Time**	**#Instrs**	**p/f2/f3**	**p/66/f2/f3**	REX
BeaEngine	311ms	672207	–	–	–
distorm	362ms	671991	–	–	–
GDSL	1364ms	671991	√	√	√
IDA Pro	/	/	√	–	√
libopcodes	574ms	671991	–	–	–
metasm	4m21s	/	–	–	√
udis86	856ms	673965	–	–	–
xed2	795ms	671991	√	√	√

Fig. 4. Evaluation of different disassembler frameworks.

4 Evaluation

We have specified a substantial fraction of the Intel x86 instruction set in GDSL as well as decoders for smaller architectures like MSP430. In this section we compare the performance and correctness of the Intel x86 decoder.

4.1 Performance

We compare the performance of our generated code with several existing disassembler projects. Table 4 shows the runtime for a linear sweep disassembly of a binary consisting of 671991 instructions in the *.text* segment. The size of the *.text* segment was 3032027 bytes. The binary is one of our earlier decoders and is a statically linked x86_64 executable for Linux. Due to linking `libc` statically, it included several *SSE* and *VEX* instructions. We used *BeaEngine* [2], *distorm* [3], *IDA Pro* [4], *libopcodes* as shipped in a Debian package [5], *metasm* [6], *udis86* [7], and the *xed2* disassembler library that comes with the *pintool* [8] package. We ran all tests on an Intel Core i7 on Linux in 64-bit mode. The discrepancy in the number of decoded instructions for *BeaEngine* and *udis86* is due to incorrectly decoded instructions which subsequently results in decoding further incorrect instructions due to different offsets.

We included the *metasm* package to complete the comparison with a disassembler not written in C. A possible reason for the results of the *metasm* package being slower is that it does not only do a linear sweep but also resolves symbols and does some control-flow analyses using the decoded instructions. Similarly, we were unable to run a linear-sweep disassembly using *IDA Pro*.

As can be seen from Table 4, the generated C code of GDSL is between two to four times slower than that of the hand-written libraries. However, decoding is unlikely to be the bottleneck in program analysis so that we deem the performance acceptable. Moreover, there are many standard compiler optimizations that can improve (shorten) the generated code considerably which would help performance. For example, the higher-order nature of GDSL (partially applied functions may be passed as parameters) requires a process called closure conversion [9]. After inlining, most functions are first order and could be called more efficiently [10]. Furthermore, substituting available expressions would eliminate many redundant calculations on bit-vectors.

4.2 Correctness

Due to the complications of decoding byte sequences that contain prefix bytes, we compared the various disassemblers for correctness. Table 4 features three columns, labelled **p/f2/f3**, **p/66/f2/f3**, and REX, which test various prefix combinations as described in Sect. 3: **p/f2/f3** states if the order of `f2` and `f3` is honoured, **p/66/f2/f3** states if additionally 66 loses its mandatory prefix status once `f2` or `f3` was read, and REX states if this prefix is correctly ignored if not immediately preceding the opcode. A tick indicates a correct decoder.

According to the Intel manual, adding arbitrary prefixes may result in unpredictable behavior for certain instructions. We created byte sequences whose behaviour is unpredictable according to the manual and verified that an Core i7 processor executes them as if the superfluous prefixes were absent. While it could be argued that decoding sequences that are marked with unpredictable behavior is undesirable for program analysis, such sequences are routinely emitted by the *gcc* compiler who inserts prefixes in front of nop and ret instructions for alignment purposes. As an example, consider the following 14-byte padding sequence that occurred in our test binary:

```
666666662e0f1f840000000000:
  nop WORD PTR cs:[rax+rax*1+0x0]
```

Here, four 66 prefixes precede a segment override prefix 2e before a nop opcode `f1 f8` follows which takes an elaborate argument. Furthermore, malware may add spurious prefixes as additional code obfuscation technique. Thus, a decoder has to recognize more than what the manual recommends.

On the contrary, certain applications, such as the search for gadgets (byte sequences that form a specific instruction), require that a decoder only recognizes instructions common to all processors. Our GDSL language can use guards from barring certain instructions from being recognized. Certain aspects, such as the difference between 32-bit and 64-bit mode can be implemented using different prefix decoders (the REX prefix is a normal instruction in 32-bit mode). We believe that an open-source implementation of a decoder is likely to converge to a decoder that is correct under all such configurations.

5 Related Work

Most decoder libraries for the Intel x86 instructions generate or use tables for mapping opcodes to instructions, however, the decoding of prefixes and arguments is usually hand-coded [5,2,3,6]. One notable exception is *SLED* [11], a specification language for encoding and decoding, which is a comprehensive specification language similar to GDSL. *SLED* specifies mnemonics using opcode-centric tables, thereby assigning fixed values to mnemonics. Besides mnemonics, it is possible to define pattern variables that associate names with sequences of bits. The mnemonics and pattern variables are then used to define an instruction. The fields of a pattern variables in such a definition can be specialized using constraints. Since these constraints are rather generic, it is not clear to which extend they can check if the resulting instruction definitions overlap (i.e. that the intersection of constraints is empty) and, thus, how often it can be avoided that constraints must be tested in sequence in order to find a matching pattern. Their approach is similar to regular expression matching, but without allowing repetition. Since the x86 allows for multiple and identical prefixes in many, but not arbitrary sequences, certain prefixed instructions are difficult to specify. In particular, the padding example using a nop in Sect. 4.2 is difficult to specify using *SLED* due to the inability to specify repetition. In fact, to our understanding, the specification given in [11] for x86 would not accept any instruction with superfluous prefixes. Even then, the ability of *SLED* to decode and encode instructions requires the specification to be bi-directional and therefore becomes relatively hard to understand and to maintain.

Another approach was taken by Fox et al. [12]. In their work they describe a formal model of the complete ARMv7 instruction set encoded in the HOL4 proof system [13]. The model directly operates on word sequences, as even the decoding logic is specified in the proof system. Besides mere decoding logic, a full semantics of the ARMv7 instruction set is also provided whose fidelity against an ARMv7 implementation was proved. Since the direct use of the decoder that is written in the HOL4 proof system is difficult, a provably correct translation to GDSL would be desirable.

Further afield is the specification of semantics for which many intermediate representations have been suggested [14,15,16]. The expressed goal of GDSL is to also specify how a processor instruction can be translated to an intermediate representation that describes its semantics. Using a common framework can help to make the various intermediate representations comparable and usable in various analysis frameworks. Recently, Reps et al. have proposed to compile an abstract transformer for each processor instruction in order to obtain more a more efficient analysis [17]. Future work will address how a different backend to our compiler can follow this setup.

Our implementation of GDSL is available at https://bitbucket.org/mb0/gdsl. It is written in SML/NJ v110.74 and released under a BSD license.

References

[1] S. Peyton Jones, *Haskell 98 Language and Libraries: the Revised Report.* Cambridge University Press, 2003.

[2] "BeaEngine," http://www.beaengine.org, 2012, version 4.1 rev 172. [Online]. Available: http://www.beaengine.org

[3] "distorm," http://www.ragestorm.net/distorm/, 2012, version 3.1. [Online]. Available: http://www.ragestorm.net/distorm/

[4] Hex-Rays, "IDA Pro Disassembler," http://www.hex-rays.com/idapro, 2012, version 6.0.101001. [Online]. Available: http://www.hex-rays.com/idapro

[5] "libopcodes," http://packages.debian.org/testing, 2012, package binutils-dev-2.22-6. [Online]. Available: http://packages.debian.org/testing

[6] "metasm," http://metasm.cr0.org/, 2012, retrieved on 2012/05/25. [Online]. Available: http://metasm.cr0.org/

[7] "udis86," http://udis86.sourceforge.net, 2012, version 1.7. [Online]. Available: http://udis86.sourceforge.net

[8] "xed2," http://www.pintool.org, 2012, version 2.11-49306. [Online]. Available: http://www.pintool.org

[9] A. W. Appel, *Compiling with Continuations.* New York, New York, USA: Cambridge University Press, 1992.

[10] A. Kennedy, "Compiling with Continuations, Continued," in *International Conference on Functional Programming.* New York, New York, USA: ACM, 2007, pp. 177–190.

[11] N. Ramsey and M. F. Fernández, "Specifying Representations of Machine Instructions," *Transactions of Programming Languages and Systems*, vol. 19, no. 3, pp. 492–524, May 1997.

[12] A. Fox and M. O. Myreen, "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture," in *International Conference on Interactive Theorem Proving*, ser. LNCS. Springer, 2010, pp. 243–258.

[13] K. Slind and M. Norrish, "A Brief Overview of HOL4," in *International Conference on Theorem Proving in Higher Order Logics*, ser. LNCS. Springer, 2008, pp. 28–32.

[14] C. Cifuentes and S. Sendall, "Specifying the Semantics of Machine Instructions," in *International Workshop on Program Comprehension*, ser. IWPC '98. Washington, Washington DC, USA: IEEE Computer Society, 1998.

[15] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The BINCOA Framework for Binary Code Analysis," in *Computer Aided Verification*, ser. LNCS. Springer, 2011, pp. 165–170.

[16] A. Sepp, B. Mihaila, and A. Simon, "Precise Static Analysis of Binaries by Extracting Relational Information," in *Working Conference on Reverse Engineering*, M.Pinzger and D. Poshyvanyk, Eds. Limerick, Ireland: IEEE Computer Society, Oct. 2011.

[17] J. Lim and T. Reps, "A System for Generating Static Analyzers for Machine Instructions," in *International Conference on Compiler Construction*, ser. LNCS. Springer, 2008, pp. 36–52.