Computer Science and Information Systems 11(4):1337–1359 DOI: 10.2298/CSIS131203076K

A New Approach to Instruction-Idioms Detection in a Retargetable Decompiler

Jakub Křoustek, Fridolín Pokorný, and Dušan Kolář

Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic
ikroustek@fit.vutbr.cz, xpokor32@stud.fit.vutbr.cz, kolar@fit.vutbr.cz

Abstract. Retargetable executable-code decompilation is a one of the most complicated reverse-engineering tasks. Among others, it involves de-optimization of compiler-optimized code. One type of such an optimization is usage of so-called instruction idioms. These idioms are used to produce faster or even smaller executable files. On the other hand, decompilation of instruction idioms without any advanced analysis produces almost unreadable high-level language code that may confuse the user of the decompiler.
In this paper, we revisit and extend the previous approach of instruction-idioms detection used in a retargetable decompiler developed within the Lissom project. The previous approach was based on detection of instruction idioms in a very-early phase of decompilation (a front-end part) and it was inaccurate for architectures with a complex instruction set (e.g. Intel x86). The novel approach is based on delaying detection of idioms and reconstruction of code to the later phase (a middle-end part). For this purpose, we use the LLVM optimizer and we implement this analysis as a new pass in this tool. According to experimental results, this new approach significantly outperforms the previous approach as well as the other commercial solutions.

Keywords: compiler optimizations, reverse engineering, decompiler, Lissom, instruction idioms, LLVM, LLVM IR

1. Introduction

Machine-code decompilation is a reverse-engineering discipline focused on reverse compilation. It performs an application recovery from binary executable files back into a high-level language (HLL) representation (e.g. C source code). Within the computer and information security, decompilation is often used for analysis of binary executable files. This is useful for vulnerability detection, malware analysis, compiler verification, code migration, etc.

In contrast to compilation, the process of decompilation is much more difficult because the decompiler must deal with incomplete information on its input (e.g. information used by the compiler but not stored within the executable file). Furthermore, the input machine code is often heavily optimized by one of the modern compilers (e.g. GCC, LLVM, MSVC). This makes decompilation even more challenging.

Furthermore, the process of decompilation is an ambiguous problem equivalent to the *halting problem* for a Turing machine [5]. This applies for example to a problem of separating code and date from the input binary program. There exist several heuristics and

algorithms to deal with this problem, but it makes it only partially computable — not in all cases.

Code de-optimization is one of the necessary transformations used within decompilers. Its task is to properly detect the used optimization and to recover the original HLL code representation from the hard-to-read machine code. One example of this optimization type is the usage of *instruction idioms* [38]. An instruction idiom is a sequence of machine-code instructions representing a small HLL construction (e.g. an arithmetic expression or assignment statement) that is highly-optimized for its execution speed and/or small size.

The instructions in such sequences are assembled together by using Boolean algebra, arbitrary-precision arithmetic, floating-point algebra, bitwise operations, etc. Therefore, the meaning of such sequences is usually hard to understand at the first sight. A notoriously known example is the usage of an exclusive or to clear the register content (i.e. `xor reg, reg`) instead of an instruction assigning zero to this register (i.e. `mov reg, 0`).

In our previous paper [21], we presented an approach of dealing with instruction-idioms detection and code reconstruction during decompilation. The implementation was tested on several modern compilers and target architectures. According to the experimental results, the proposed solution was highly accurate on the RISC (Reduced Instruction Set Computer) processor families—up to 98%; however, this approach was inaccurate (only 21%) for more complex architectures, such as CISC (Complex Instruction Set Computer).

In this paper, we present an enhanced approach of instruction-idioms detection and code reconstruction, which can be effectively used even on CISC architectures. It has been adapted within an existing retargetable decompiler developed within the Lissom project [22, 36]. Moreover, this decompiler is developed to be retargetable (i.e. independent on a particular target platform, operating system, file format, or a used compiler). Therefore, the proposed analysis has to be retargetable too.

This paper is organized as follows. In Section 2, we give an introduction to instruction idioms and their usage within compiler optimizations. The most common instruction idioms employed in the modern compilers are also presented and illustrated in there. Then, we briefly describe the retargetable decompiler developed within the Lissom project in Section 3. Afterwards, in Section 4, we present both of our approaches, the original one and the novel one. Section 5 discusses the related work of instruction-idioms detection. Experimental results are given in Section 6. In that section, we also compare both approaches together with one commercial solution. Section 7 closes the paper by discussing future research.

2. Instruction Idioms used in Compilers

In present, the modern compilers use dozens of optimization methods for generating fast and small executable files. Different optimizations are used based on the optimization level selected by the user. For example, the GNU GCC compiler supports these optimization levels[1]:

[1] See `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` for details.

- `O0` – without optimizations;
- `O1` – basic level of speed optimizations;
- `O2` – the common level of optimizations (the ones contained in `O1` together with basic function inlining, peephole optimizations, etc.);
- `O3` – the most aggressive level of optimizations;
- `Os` – optimize for size rather than speed.

In the nowadays compilers, the emission of instruction idioms cannot be explicitly turned on by some switch or command line option. Instead, these compilers use selected sets of idioms within different optimization levels. Each set may have different purpose, but multiple sets may share the same (universal) idioms.

There are several reasons why to use instruction idioms. The main reasons are given next.

- The most straightforward reason is to exchange slower instructions with the faster ones. These optimizations are commonly used even in the lower optimization levels.
- The floating-point unit (FPU) might be missing, but a programmer still wants to use floating-point numbers and arithmetic. Compilers solve this task via floating-point emulation routines (also known as *software floating point* or *soft-float* in short). Such routines are generated instead of hardware floating-point instructions and they perform the same operation by using available (integer) instructions.
- Compilers often support an optimization-for-size option. This optimization is useful when the target machine is an embedded system with a limited memory size. An executable produced by a compiler should be as small as possible. In this case, the compiler substitutes a sequence of instructions encoded in more bits with a sequence of instructions encoded in less bits in general. This can save some space in instruction cache too.

Another type of optimization classification is to distinguish them based on the target architecture. Some of them depend on a particular target architecture. If a compiler uses platform-specific information about the generated instructions, these instructions can be classified as platform-specific. Otherwise, they are classified as platform-independent.

As an example of a platform-independent idiom, we can mention the `div` instruction representing fixed-point division. Fixed-point division (signed or unsigned) is one of the most expensive instruction in general. Optimizing division leads to a platform-independent optimization.

On the other hand, clearing the content of a register by using the `xor` instruction (mentioned in the introduction) is a highly platform-specific optimization. Different platforms can use different approaches to clear the register content. As an example, consider the zero register on MIPS (`$zero` or `$0`), which always contains the value of 0. Using this register as a source of zero bits may be a faster solution than using the `xor` instruction.

Furthermore, different compilers use different instruction idioms to fit their optimization strategies. For example, GNU GCC uses an interesting optimization when making signed comparison of a variable. When a number is represented on 32-bits and bit number 31 is representing the sign, logically shifting the variable right by 31 bits causes to set the zeroth bit equal to the original sign bit. The C programming language classifies 1 as

true and 0 as a *false*, which is the expected result of the given less-than-zero comparison. This idiom is shown in Figure 1. Figure 1a represents a part of a source code with this construction. The result of its compilation with optimizations enabled is depicted in Figure 1b. We illustrate the generated code on the C level rather than machine-code level for better readability.

```
int main(void)
{
  int a, b;

  /* ... */

  b = a < 0;

  /* ... */
}
```

(a) Input.

```
int main(void)
{
  int a, b;

  /* ... */

  b = lshr(a, 31);

  /*... */
}
```

(b) Output (for better readability in C).

Fig. 1: Example of an instruction idiom (C code).

The compiler used the before-mentioned instructions idiom—replacing the comparison by the shift operation. The non-standardized `lshr()` function is used in the output listed in Figure 1b. The C standard does not specify whether operator ">>" means logical or arithmetical right shift. Compilers deal with it in an implementation-defined manner. Usually, if the left-hand-side number used in the shift operation is signed, arithmetical right shift is used. Analogically, logical right shift is used for unsigned numbers.

In Table 1, we can see a shortened list of instruction idioms used in common compilers. This list was retrieved by studying the source codes responsible for code generation (this applies to open-source compilers—GNU GCC 4.7.1, Open Watcom 1.9, and LLVM/`clang` 3.3) and via reverse engineering of executable files generated by these compilers (this method was used for other compilers—Microsoft Visual Studio C++ Compiler 16 and 17, Borland C++ 5.5.1, and Intel C/C++ Compiler XE13). Some of these instruction idioms are widespread among modern compilers. We have also found out that actively developed compilers, such as GNU GCC, Visual Studio C++, and Intel C/C++ Compiler, are using these optimizations heavily. For example, they generate the `idiv` instruction (fixed signed division) only in rare cases on the Intel x86 architecture. Instead, they generate optimized division by using magic number multiplication.

3. Lissom Project Retargetable Decompiler

In this section, we briefly describe the concept of an automatically generated retargetable decompiler developed within the Lissom project [22]. This decompiler aims to be independent on any particular target architecture, operating system, object file format, or

Table 1: Shortened list of instruction idioms found in compilers.

Instruction idiom	GNU GCC	Visual Studio C++	Intel C/C++ Compiler	Open Watcom	Borland C Compiler	LLVM
Less than zero test	✓	×	✓	×	×	✓
Greater or equal to zero test	✓	×	×	×	×	✓
Bit clear by using `xor`	✓	✓	✓	✓	✓	✓
Bit shift multiplication	✓	✓	✓	✓	✓	✓
Bit shift division	✓	✓	✓	✓	✓	✓
Division by `-2`	✓	×	×	×	×	✓
Expression `-x - 1`	✓	✓	×	×	×	×
Modulo power of two	✓	✓	✓	×	×	✓
Negation of a float	✓	×	×	×	×	×
Assign `-1` by using `and`	×	✓	✓	×	×	×
Multiplication by an invariant	✓	✓	✓	✓	×	✓
Signed modulo by an invariant	✓	×	✓	×	×	✓
Unsigned modulo by an invariant	✓	✓	✓	×	×	✓
Signed division by an invariant	✓	✓	✓	×	×	✓
Unsigned division by an invariant	✓	✓	✓	×	×	✓
Substitution by `copysignf()`	✓	×	×	×	×	×
Substitution by `fabsf()`	✓	×	×	×	×	×

originally used compiler. The concept of the decompiler is depicted in Figure 2. Its detailed description can be found in [36]. Currently, the decompiler supports decompilation of MIPS, ARM, and Intel x86 executable files stored in different file formats.

The input binary executable file is preprocessed at first. The preprocessing part tries to detect the used file format, compiler, and (optional) packer, see [18] for details. Afterwards, it unpacks and converts the examined platform-dependent application into an internal uniform Common-Object-File-Format (COFF)-based representation. Currently, we support conversions from UNIX ELF, Windows Portable Executable (WinPE), Apple Mach-O, Symbian E32, and Android DEX file formats. The conversion is done via our plugin-based converter, described in [20, 17]. Afterwards, such a COFF file is processed in the decompilation core that consists of three basic parts—a *front-end*, a *middle-end*, and a *back-end*. The last two of them are built on top of the LLVM Compiler Infrastructure [34]. LLVM Intermediate Representation (LLVM IR) [23] is used as an internal code representation of the decompiled applications in all particular decompilation phases.

After that, the unified COFF files are processed by the front-end part.Within this part, we use the ISAC architecture description language [24] for an automatic generation of the *instruction decoder*. The decoder translates the machine-code instructions into sequences of LLVM IR instructions. The resulting LLVM IR sequence characterizes behaviour of the original instruction independently on the target platform. This intermediate program

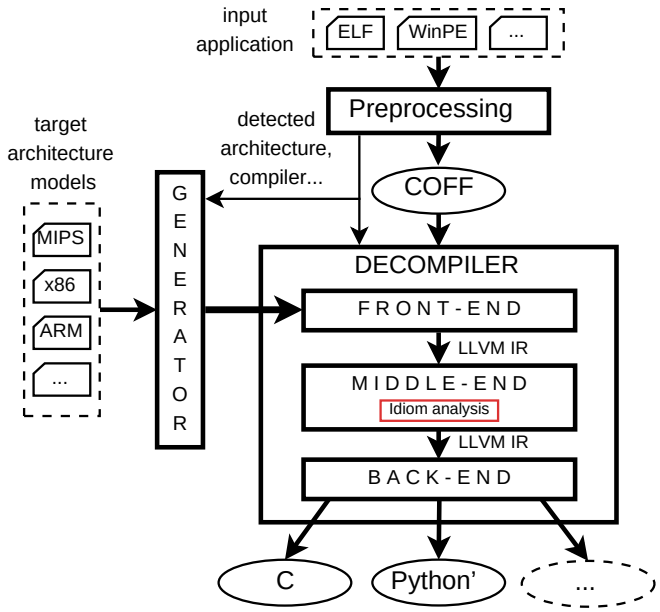

Fig. 2: The concept of the Lissom project retargetable decompiler.

representation is further analysed and transformed in the static-analysis phase of the front-end. This part is responsible for eliminating statically linked code, detecting the used ABI, recovery of functions, etc. [36]. When debugging information (e.g. DWARF, Microsoft PDB) or symbols are present in the input application, we may utilize them to get more accurate results, see [19].

The output of the front-end part (i.e. LLVM IR code representing the input application) is sizable. The main reason is because it reflects a complete behaviour of each machine-code instruction, which may not be necessary. For example, each side-effect of an instruction (e.g. setting a register flag based on instruction operands) is represented via the LLVM IR code, but results of these side-effects may not be used anywhere. Therefore, the front-end output is further processed within the middle-end phase, which is built on top of the LLVM `opt` tool. This phase is responsible for reduction and optimization of this code by using many built-in optimizations available in LLVM (e.g. optimizations of loops, constant propagation, control-flow graph simplifications) as well as our own passes. Besides our decompilation project, `opt` is normally used as an optimization part of the LLVM compiler toolchain.

Finally, the back-end part converts the optimized intermediate representation into the target high-level language (HLL). Currently, we support C and a Python-like language. The latter is very similar to Python, except a few differences—whenever there is no support in Python for a specific construction, we use C-like constructs. During the back-end conversion, high-level control-flow constructs, such as loops and conditional statements, are identified, reconstructed, and further optimized. Finally, it is emitted in the form of the target HLL.

The decompiler is also able to produce the call graph of the decompiled application, control-flow graphs for all functions, and an assembly representation of the application.

4. Idiom Analysis in the Retargetable Decompiler

The aim of the decompiler presented in the previous section is to allow retargetable decompilation independently on the particular target platform or the used compiler. Therefore, the methods of instruction-idioms detection and code reconstruction have to be retargetable too. For this purpose, we present two approaches that use the unified code representation in the LLVM IR format.

LLVM IR is a set of low-level instructions similar to assembly instructions. Moreover, LLVM IR is platform-independent and strongly typed, which meets our requirements. Therefore, machine instructions from different architectures can be easily mapped to sequences of LLVM IR instructions. This brings an ability to implement platform-independent instruction-idioms analysis.

The first approach was presented in our previous work [21]. We implemented this approach within the front-end part of the decompiler. However, according to our experimental results, this approach was not optimal for complex programs and architectures. Therefore, later in this section, we propose an advanced approach implemented in the middle-end phase. In the following text, we describe both approaches, compare them, and describe their disadvantages.

4.1. Original Approach

The simplified algorithm of instruction-idioms detection in the front-end is depicted in Algorithm 1. It sequentially inspects instructions and tries each one of them as a possible start of any instruction idiom (marked as $1, 2, \ldots, n$). Every instruction that follows has to use the expected operands and results. If an instruction idiom is found (via a function `FIND_IDIOM_X`), the inspection continues after the last instruction belonging to this detected idiom. Note that any instruction inserted by a compiler into a sequence representing an instruction idiom causes a failure in the instruction-idioms detection. In other words, this approach of instruction-idioms analysis is highly dependent on the order of instructions. If this order is violated or another instruction is scheduled by a compiler, the detection of instruction idioms fails, see Section 4.2 for details.

Algorithm 1 Detection of instruction idioms in the front-end phase.

```
instruction i = BasicBlock.start

while i ≠ BasicBlock.end do
    i += FIND_IDIOM_1(i)
    i += FIND_IDIOM_2(i)
    ...
    i += FIND_IDIOM_n(i)
end while
```

This algorithm is similar to a peephole technique used in optimizing compilers [6]. It operates on a basic-block level, where each basic block contains a continuous sequence of instructions described via LLVM IR operations. A particular idiom is detected only if a basic block contains predefined LLVM IR instructions stored in a proper order and they must contain expected operand values (e.g. constant, register number). Whenever an instruction idiom is detected, it is substituted by its more readable de-optimized version, once again in the LLVM IR form. While examining instructions in an idiom sequence, we may found unrelated instructions (e.g. inserted by a code-motion compiler optimization). In that case, the algorithm fails to detect the idiom. It should be noted that this algorithm does not search for a particular idiom over multiple basic blocks. An example of a traversal by this algorithm is depicted in Figure 3.

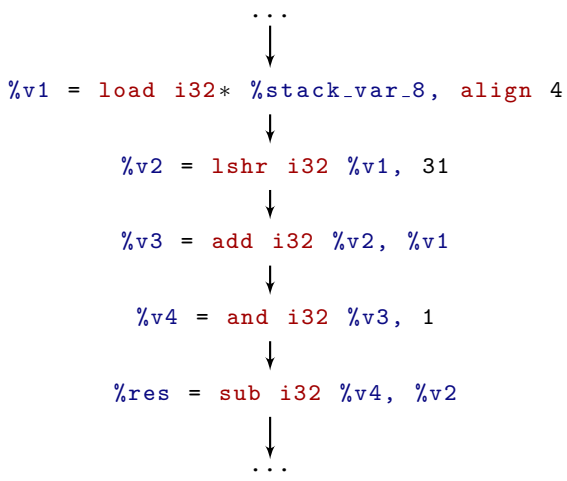

Fig. 3: Example of instruction-idioms inspection in the front-end part. All instructions are inspected sequentially.

An example demonstrating this substitution on the LLVM IR level is shown in Figure 4. Figure 4a represents the already mentioned `xor` bit-clear instruction idiom. To use register content, a register value has to be loaded into a typed variable `%1`. By using the `xor` instruction, all bits are zeroed and the result (in variable `%2`) can be stored back into the same register. To transform this idiom into its de-optimized form, a proper zero assignment has to be done. This de-optimized LLVM IR code is shown in Figure 4b. In this case, the typed variable `%2` holds zero, which can be directly stored in the register.

The detection of an instruction idiom in the front-end part is a challenging task because of the complexity of the input LLVM IR code. For example, the expected operand values (e.g. values used for magic number multiplication) may not be stored as clearly as in the original HLL source code. For example, the original HLL constant may not be stored directly as a number (i.e. an immediate value), but it may be computed through several machine-code instructions. These instructions fold the original value at run-time based on different resources (e.g. register value, memory content). For example, the MIPS

```
%1 = load i32* @regs0
%2 = xor i32 %1, %1
store i32 %2, i32* @regs0
```

(a) Optimized form of an instruction idiom in LLVM IR.

```
store i32 0, i32* @regs0
```

(b) De-optimized form of an instruction idiom in LLVM IR.

Fig. 4: Example of the bit-clear `xor` instruction-idiom transformation.

instruction set does not allow direct load of a 32-bit immediate value and it has to be done using more instructions (e.g. `lui` and `ori`). Therefore, the operand value is not stored directly within one instruction but it is assembled by an instruction sequence. This is quite complicated because the idiom-detection phase (as well as the rest of the decompiler) is done statically and run-time information is unavailable. To deal with this problem, we utilize a static-code interpreter, originally used for function reconstruction—see [36] for a detailed description of the interpreter.

Using an interpreter to statically compute a value stored in a register is quite common task in instruction-idioms analysis. An example is shown in Figure 5. An interpreter has to be run to statically compute a number stored in register `@regs3` by using the backtracking of previously used operations and their operands. The result obtained in this case is `680390859`. This number is used in optimized division by number `101` performed by the magic-number multiplication on ARM and the GNU GCC compiler. Another similar issue is accessing the data segment to load constants; the interpreter can solve this issue as well.

4.2. Novel Approach

As has been noted, the original approach (described in Section 4.1) was implemented in the front-end phase. However, this analysis does not fit in this phase. The LLVM IR representation in the front-end is on a very low level. With only a few exceptions, one machine code instruction is usually translated to multiple LLVM IR instructions. If we realise that a program usually contains thousands of instructions, we have a very large set of LLVM IR instructions to be inspected. This causes a negative impact on decompilation time and accuracy. However, both of these metrics can be enhanced if we inspect instruction idioms on a more optimized form.

Moreover, the front-end represents instructions in a native way—as a list of LLVM IR instructions. Implementing instruction-idioms analysis in this way is not easy due to the position of instructions, especially on CISC architectures (e.g. Intel x86). For example, CISC superscalar processors have instructions with varying execution time that can occupy different CPU[2] units (e.g. adder, multiplier, branch unit) at a different time. Furthermore, different techniques are used to reduce the run-time, such as maximal utilization

[2] Central processing unit.

```
%a = add i32 679477248, 0
store i32 %a, i32* @regs3

%b = load i32* @regs3
%b_1 = add i32 913408, 0
%b_2 = add i32 %b_1, %b
store i32 %b_2, i32* @regs3

%c = load i32* @regs3
%c_1 = add i32 203, 0
%c_2 = add i32 %c_1, %c
store i32 %c_2, i32* @regs3

; @regs3 contains value 680390859
; 680390859 = 203 + 913408 + 679477248
```

Fig. 5: An example of a constant computation in LLVM IR.

of CPU units (e.g. Thornton's or Thomasulo's algorithm [28, 1]). By using these techniques, superscalar processors can fetch and decode instructions more effectively. Therefore, modern compilers try to optimize instruction positions to improve performance and they also try to avoid instruction hazards (data, structural, or control) via spreading of instructions to different places.

As well as any other instruction, instruction idioms can be also spread across basic blocks. Looking for such a shuffled instruction idiom in a linear search, used in the original approach, can lead to a failure if advanced optimizations were turned on at the compile time. Therefore, a more sophisticated algorithm has to be used.

On the other hand, the middle-end phase represents instructions also as a sequence. However, these instructions can be easily inspected in a tree way—by using a derivation tree (see LLVM pattern matching [34]). Inspecting a derivation tree can remove problems with positioning of instructions. Furthermore, the main goal of the middle-end part is to optimize instructions and remove duplicate or redundant instructions, which results in a low-level transformation from machine code into LLVM IR. Therefore, we decided to move the front-end implementation of instruction-idioms analysis to the middle-end part.

For example, if we compare a simple sequence of machine code calculating multiplication, depicted in Figure 6, between the front-end and middle-end, there is a significant difference in code complexity in favor of the middle-end. This program, illustrated in assembly language, represents a part of a program that is being decompiled. Its instructions have to be decoded and stored as LLVM IR in the front-end part, see Figure 7.

```
; ...
mov @reg2, address
mul @reg2, 21
; ...
```

Fig. 6: Example of a program for decompilation (assembly code).

```
; ...
%0 = load i32* %address
store i32 %0, i32* @reg2

%1 = load i32* @reg2
%1_64 = sext i32 %1 to i64
%tmp1 = add i32 0, 0
%2 = add i32 21, 0
%2_64 = sext i32 %2 to i64
%3 = mul i64 %1_64, %2_64
%imm_32 = add i64 32, 0
%4 = lshr i64 %3, %imm_32
%5 = trunc i64 %4 to i32
%6 = trunc i64 %3 to i32
%tmp2 = add i32 0, 0
store i32 %6, i32* @reg0
%tmp3 = add i32 2, 0
store i32 %5, i32* @reg2
; ...
```

Fig. 7: LLVM IR representation of code from Figure 6 (in the front-end part).

Translation from machine instructions into LLVM IR is done for every single instruction and every dependence is omitted because there is no context information yet. This approach causes generation of a very large number of LLVM IR instructions. Such a representation is used in the front-end phase because it is a very early phase of the decompilation process.

LLVM IR representation in the front-end phase is not suitable for a high-level analysis, such as instruction-idioms analysis because of its complexity. Moreover, representation of instructions in LLVM IR is highly dependent on the target architecture. As can be seen in Figure 7, the result of the multiplication instruction (`mul`) on the used architecture is a 64-bit number stored in two registers. On some architectures the result of such a multiplication instruction is only a 32-bit number. As has been stated in introduction, instruction-idioms analysis should be architecture independent. This is only one of the problems related to inspecting instruction idioms in the front-end.

Contrariwise, the code depicted in Figure 7 is being heavily optimized during the middle-end phase and the result is shown in Figure 8. As can be seen, all architecture-dependent computations are removed; moreover, if the higher 32-bits of a 64-bit result are not used, they are removed in dead-code-elimination optimization too. Looking for an instruction idiom in such a straightforward representation is much easier, platform independent, and it leads to better instruction-detection results.

```
; ...
%1 = load i32* @reg2
%2 = mul i32 %1, 21
store i32 %2, i32* @reg2
; ...
```

Fig. 8: LLVM IR representation of code from Figure 6 (in the middle-end part).

The algorithm used in the middle-end differs from the algorithm originally used in the front-end (i.e. Algorithm 1). It is described in Algorithm 2. Every basic block is inspected sequentially starting from the beginning. Every instruction is treated as a possible root of a derivation tree containing one particular instruction idiom (again marked as $1, 2, \ldots, n$). If so, the derivation tree is inspected and if an instruction idiom is found, it can be easily transformed to its de-optimized form. Since a derivation tree does not depend on the position of instructions in LLVM IR but rather on the use of instructions, position-dependent problems are solved in this way.

Algorithm 2 Detection of instruction idioms in the middle-end phase.

```
function IDIOM_INSPECTOR_i(BasicBlock)
    for all instruction in BasicBlock do
        FIND_IDIOM_i(instruction)
    end for
end function

IDIOM_INSPECTOR_1(BasicBlock)
IDIOM_INSPECTOR_2(BasicBlock)
...
IDIOM_INSPECTOR_N(BasicBlock)
```

An example of this algorithm is depicted in Figure 9. Its derivation tree is illustrated in Figure 10.

As stated above, the main goal of the middle-end part is to optimize LLVM IR that was analysed in the front-end. The middle-end uses different `opt` passes to get optimal code for the back-end part. Instruction-idioms analysis has been developed as one of the basic-block passes of `opt`. Some transformations are useful for instruction-idioms pass, thus it is important to fit the instruction-idioms pass into a proper position within other passes.

It is also important to mention that decompilation of an executable file is a time consuming process. The decompilation time highly depends on the size of the input executable file. A good approach how to optimize instruction-idioms analysis is to use any available information to save decompilation time. This is especially important when we support many instruction idioms. Some of them are specific for a particular compiler and therefore, they can be omitted from the detection phase whenever another compiler is detected. On the other hand, detection of the used compiler (as described in [18]) may be inaccurate in some cases and the algorithm will not detect any used compiler. In that case, the idiom analysis tries to detect all the supported idioms. Another optimization approach is to detect only the platform-specific idioms based on the target architecture and omit idioms for other architectures.

However, the information about the architecture and compiler has to be propagated into the middle-end because it is usually not available in this phase. The only input in `opt` is a file with LLVM IR so this file has to carry this information by using the LLVM metadata mechanism. This gives us an ability to inspect only instruction idioms that are used by compilers on a given architecture.

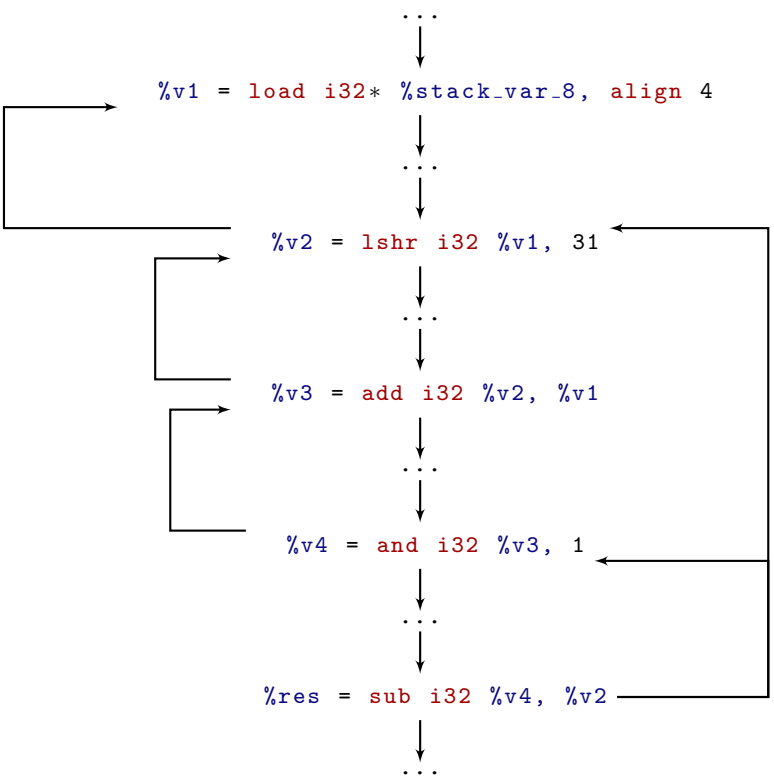

Fig. 9: Example of instruction-idioms inspection in the middle-end part. Instructions are inspected in a tree-way.

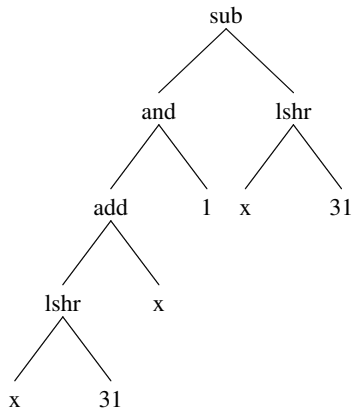

Fig. 10: Derivation tree created based on Figure 9.

The number of instructions in LLVM IR that are going to be analysed for instruction idioms should be as small as possible. As obvious, analysing less instructions takes

less time. Consider a signed division idiom, which was found in the GCC, Visual Studio and Intel C/C++ compilers. This instruction idiom can compute a magic number used in the multiplication instruction on ARM as shown in Figure 5. Even the magic number computed here is known at compile-time, it cannot be used in the multiplication instruction because of number of bits available in the multiplication instruction to represent a constant immediate. This computation would require advanced analysis in the instruction-idioms pass. LLVM `opt` can easily fold constants in an instruction combining pass (`instcombine`).

This instruction combining pass is run multiple times during the processing in the middle-end phase. However, besides our decompilation project, `opt` is normally used as an optimization part of the LLVM compiler toolchain and it also uses instruction idioms for code optimizations within the instruction combining pass. Therefore, this pass has tendencies to bring back idioms instead of the de-optimized code. This reason leads to turn some of the instruction combining pass optimizations off, mainly optimizations based on instruction idioms. This disabling has to be done via direct modification of `opt` source codes because there is no such command-line option, etc.

In present, we support detection of all instruction-idioms specified in Table 1, among others. In Figure 11, we demonstrate a code reconstruction for one of these idioms. In this figure, we can compare decompilation results with and without the instruction-idioms analysis. Figure 11a illustrates a simple C program containing the division idiom. The decompilation result obtained without instruction-idioms analysis is depicted in Figure 11c. It contains three shift operations and one multiplication by a magic value. Without the knowledge of the fundamentals behind this idiom, it is almost impossible to understand the resulting code. On the other hand, the decompilation result with instruction-idioms analysis enabled is well readable and a user can focus on the meaning of the program, not on deciphering optimizations done by a compiler, see Figure 11b.

In the conclusion of this section, we can state that the novel approach is more powerful and robust than the original one. The LLVM `opt`, used as a core of the middle-end phase, also supports passes over multiple basic blocks, which is promising for our future work.

5. Related Work

The fundamentals of instruction idioms and their usage within compiler optimizations are well documented, see [38, 3, 32, 12, 14, 29]. From these publications, we can gain insights into the principles behind instruction idioms as well as how and when to use them to obtain more effective machine code.

Contrariwise, the detection of instruction idioms and code reconstruction from machine code is mostly an untouched area of machine-code decompilation. This topic is only briefly mentioned in [5, 9, 37]. Nevertheless, some of the existing (non-retargetable) decompilers support this feature. In order to observe the state of the art, we look closely on their approaches.

We used a test containing five idioms from a larger list listed in Table 1. These idioms are the most common ones (e.g. multiplication via left shift) and the support of idiom detection within the tested decompiler should be easily discovered via these idioms. A source code of this test is listed in Figure 12. Each expression of the `printf` function represents one instruction idiom, whose meaning is described in Section 4. This source

```
int main(void)
{
  int a;

  /* ... */

  a = a / 10;

  /* ... */
}
```

(a) Input.

```
int main(void)
{
  int a;

  /* ... */

  a = a / 10;

  /* ... */
}
```

(b) Output with idiom analysis enabled.

```
int main(void)
{
  int a;

  /* ... */

  a = (lshr(a * 1717986919, 32) >> 2) - (a >> 31)
      ;

  /* ... */
}
```

(c) Output with idiom analysis disabled.

Fig. 11: C code example of decompilation with and without the idiom analysis.

code was compiled for different target platforms (i.e. processor architecture, operating system, and file format) based on their support in each decompiler. Finally, each decompiler was tested by using this executable file and we analysed the decompiled results afterwards.

Boomerang is the only existing open-source machine-code decompiler [4]. However, it is no longer developed. According to our tests, it was able to reconstruct only the first instruction idiom.

REC Studio (also known as REC Decompiler) is freeware, but not an open-source decompiler. It has been actively developed for more than 25 years [30]. None of the instruction idioms was successfully reconstructed. We only noticed that REC Studio can reconstruct the register cleaning idiom (via the `xor` instruction), described in Section 1.

SmartDec decompiler is another closed-source decompiler specialising on decompilation of C++ code, see [31] for details. However, SmartDec was unable to reconstruct any instruction idiom from the machine-code.

Hex-Rays decompiler [13] achieved the best results—three successfully reconstructed idioms from five (it succeeded in the first, second, and fourth test). Therefore, we have

```c
#include <stdio.h>
int main(void)
{
  int a;

  /* ... */

  printf("1. Multiply: %d\n", a * 4);
  printf("2. Divide: %d\n", a / 8);
  printf("3. >= 0 idiom: %d\n", a >= 0);
  printf("4. Magic sign-div: %d\n", a / 10);
  printf("5. XOR by -1: %d\n", -a - 1);
  return a;
}
```

Fig. 12: C source code used to test the decompilers.

chosen this decompiler for a more detailed comparison with our own solution, as described in Section 6.

There are two other interesting projects. The *dcc* decompiler was the first one of its kind, but it is unusable for modern real-world decompilation because it is no longer developed [33, 5]. On the other hand, the *Decompile-it.com* project looks promising, but the public beta version [7] is probably still in an early stage of development and it cannot handle any of these instruction idioms.

In conclusion, we cannot compare our idiom-detection algorithm with approaches used in other tools because of two reasons. (1) They are not distributed as open-source. (2) The open-source solutions do not support idiom recovery at all or they support only a very limited number of idioms. On the other hand, we can compare our results with the Hex-Rays Decompiler.

6. Experimental Results

This section contains an evaluation of both proposed approaches (i.e. the original one from [21] and the novel one) of instruction-idioms detection and code reconstruction. The decompiled results are also compared with the nowadays decompilation "standard"—the Hex-Rays Decompiler [13] that is a plugin to the IDA disassembler [15]. We used the latest versions of these tools, i.e. Hex-Rays Decompiler v1.8.0.130306 and IDA disassembler v6.4.130306. The Hex-Rays Decompiler is not an automatically generated retargetable decompiler, such as our solution, and it supports only the Intel x86 and ARM target architectures. Our solution also supports the MIPS architecture at the moment.

In our project, all the three mentioned architectures are described as instruction-accurate models in the ISAC language in order to automatically generate our retargetable decompiler. MIPS is a 32-bit processor architecture, which belongs to the RISC processor family. The processor description is based on the MIPS32 Release 2 specification [27]. ARM is also a 32-bit RISC architecture. The ISAC model is based on the ARMv7-A specification with the ARM instruction set [2]. The last architecture used for the comparison

is Intel x86 (also known as IA-32) that belongs in the CISC processor family. The model is based on the 32-bit processor core specified in [16] without extensions (e.g. x86-64).

We created 21 test applications in the C language. Each test is focused on the detection of a different instruction idiom. The Minimalist PSPSDK compiler (version 4.3.5) [26] was used for compiling MIPS binaries into the ELF file format, the GNU ARM toolchain (version 4.1.1) [11] for ARM-ELF binaries, and the GNU compiler GCC version 4.7.2 [10] for x86-ELF executables (the 32-bit mode was forced by the `-m32` option).

As can be observed, we used the ELF file format in each test case. However, the same results can be achieved by using the WinPE file format [35, 25]. All three compilers are based on GNU GCC. The reason for its selection is the fact that it allows retargetable compilation to all the three target architectures and it also supports most of the idioms specified in Sections 2 and 4.

Different optimization levels were used in each particular test case. Because of different optimization strategies used in compilers, not every combination of source code, compiler, and its optimization level leads to the production of an instruction idiom within the generated executable file. Therefore, we count only the tests that contain instruction idioms. Furthermore, it is tricky to create a minimal test containing an instruction idiom without its removal by compiler during compilation.

An example of this problem is depicted by using a C code with the multiplication idiom in Figure 13a. The result of this code can be computed during compilation. Therefore, the compiler emits directly the result without the code representing its computation (see the example in Figure 13b). Therefore, we use functions from the standard C library for the initialization of variables used in idioms. For example, this can be done by using statements `a = rand();` or `scanf("%d", &a);`. An example of an enhanced test is depicted in Figure 13c. Such code cannot be eliminated during compilation and the instruction idiom is successfully generated in the executable file, see Figure 13d.

The testing was performed on Intel Core i5 (3.3 GHz), 16 GB RAM running a Linux-based 64-bit operating system. The GCC compiler (v4.7.2) with optimizations enabled (`O2`) was used to build the decompiler.

Finally, we enabled the emission of debugging information in the DWARF standard [8] by using the `g` option because both decompilers exploit this information to produce more accurate code, see [19] for details. The debugging information helps to eliminate inaccuracy of decompilation (e.g. entry-point detection, function reconstruction) that may influence testing. However, the debugging information does not contain information about the usage of the idioms and therefore, its usage does not affect the idiom-detection accuracy.

All test cases are listed in Table 2. The results of our original approach are marked as **Lissom1**, the results of our new approach are marked as **Lissom2**, and finally, the results of the Hex-Rays decompiler are marked as **Hex-Rays**.

The first column represents the description of a particular idiom used within the test. The maximal number of points for each test on each architecture is five (i.e. one point for each optimization level—`O0`, `O1`, `O2`, `O3`, `Os`). Some idioms are not used by compilers based on the optimization level or target architecture. Therefore, the number of total points can be lower than five. For example, the MIPS and ARM architectures lack a floating-point unit (FPU) and the essential FPU operations are emulated via *soft-float* idioms. On the other hand, the Intel x86 architecture implements these operations via the x87

Table 2: Experimental results—the number of successfully detected and reconstructed instruction idioms. Note: several tests differ only in the used numeric constant; however, different instruction idioms are emitted based on this value.

	MIPS			ARM				Intel x86			
		Lissom1	Lissom2		Hex-Rays	Lissom1	Lissom2		Hex-Rays	Lissom1	Lissom2
Tested instruction idiom	tests	(%)	(%)	tests	(%)	(%)	(%)	tests	(%)	(%)	(%)
`intA = intB < 0`	5	100.0	100.0	5	0.0	100.0	100.0	5	0.0	0.0	100.0
`intA = intB >= 0`	5	100.0	100.0	5	0.0	100.0	100.0	5	0.0	100.0	100.0
`intA = 0`	0	-	-	0	-	-	-	1	100.0	100.0	100.0
`intA = intB * 4`	5	100.0	100.0	5	100.0	100.0	100.0	5	100.0	100.0	100.0
`intA = intB / -2`	0	-	-	5	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = intB / 4`	1	0.0	100.0	5	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = intB / 10`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = intB / 120`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`uintA = uintB / 7`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`uintA = uintB / 9`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = -intB - 1`	5	100.0	100.0	5	0.0	100.0	100.0	5	100.0	100.0	100.0
`intA = intB % 2`	0	-	-	5	40.0	80.0	80.0	4	0.0	0.0	100.0
`intA = intB % 3`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = intB % 5`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`intA = intB % 8`	0	-	-	4	75.0	100.0	100.0	4	25.0	0.0	100.0
`uintA = uintB % 3`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`uintA = uintB % 5`	0	-	-	4	100.0	100.0	100.0	4	100.0	0.0	100.0
`uintA = uintB % 8`	5	100.0	100.0	5	20.0	100.0	100.0	5	0.0	0.0	100.0
`floatA = -floatB`	5	100.0	100.0	5	0.0	100.0	100.0	0	-	-	-
`floatA = copysign(floatB, floatC)`	5	100.0	100.0	5	0.0	100.0	100.0	0	-	-	-
`floatA = fabs(floatB)`	5	100.0	100.0	5	0.0	100.0	100.0	0	-	-	-
Total	**41**	**97.6**	**100.0**	**91**	**58.2**	**98.9**	**98.9**	**74**	**63.5**	**21.6**	**100.0**

```c
int main(void)
{
  int a = 1;
  a = a * 8;
  return a;
}
```

(a) Test C code.

```c
int main(void)
{
  return 8;
}
```

(b) Compiler-optimized code without an instruction idiom.

```c
#include <stdlib.h>
int main(void)
{
  int a = rand();
  a = a * 8;
  return a;
}
```

(c) Enhanced test C code.

```c
#include <stdlib.h>
int main(void)
{
  int a = rand();
  a = a << 3;
  return a;
}
```

(d) Compiler-optimized code with an instruction idiom.

Fig. 13: Problem of idiom removal by compiler.

floating-point instruction extension. Therefore, the instruction idioms are not used in this case.

The overall decompilation results are depicted in Figure 14. We can observe four facts based on the results.

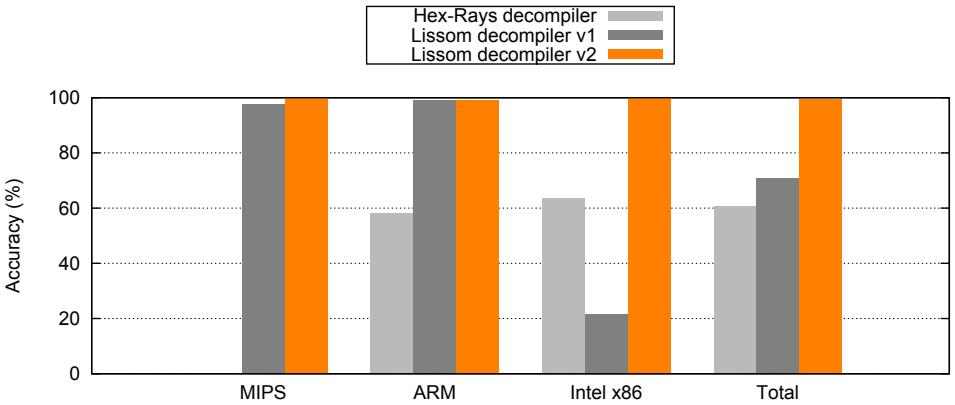

Fig. 14: Accuracy of instruction-idioms detection and code reconstruction. Note: the total accuracy of Hex-Rays decompiler is calculated based on ARM and Intel x86 only.

(1) As we have mentioned earlier, the Hex-Rays decompiler does not support the MIPS architecture. Therefore, we are unable to compare our results on this architecture.

(2) The results of the Hex-Rays decompiler on ARM and Intel x86 are very similar (approximately 60%). Its authors covered the most common idioms for both architectures (multiplication via bit shift, division by using magic-number multiplication, etc.). However, the non-traditional idioms are covered only partially or not at all (e.g. integer comparison to zero, floating-point idioms).

(3) Our original approach (i.e. idiom detection within the front-end phase) reaches its limits on the Intel x86 architecture, where the accuracy drops to 21%.

(4) Our new approach (i.e. idiom detection within the middle-end phase) achieved almost perfect results on all the architectures (99.5% in total); only one test for the ARM architecture failed. It was a test calculating a modulo operation by 2 compiled with the `Os` optimization. The generated machine code calculates this operation over multiple basic blocks. In present, this is not covered by our approach. However. it can be easily solved via a new `opt` pass. This is marked as a future work.

Furthermore, the new approach is even faster than the previous one—approximately 20% based on the target application, architecture, compiler, and compilation options. There are two reasons for such an speed improvement: (1) The LLVM `opt` (used in the middle-end phase) supports a framework for creating own passes over the input LLVM IR code. Such passes are heavily optimized for speed and they can achieve a higher speed than writing such a pass on your own (our previous approach). (2) In the new approach, the instruction idioms are detected on an already optimized code (e.g. dead-code elimination pass). Therefore, it is necessary to search in a smaller amount of code than in the previous approach.

7. Conclusion

In this paper, we presented a novel approach of instruction-idioms detection and code reconstruction during the decompilation process of an existing retargetable decompiler. This new approach is based on the previous one described in [21]. The novelty is implemented via delaying the instruction-idioms detection phase from the front-end part into the middle-end part. The novel instruction-idioms analysis has been successfully implemented in the middle-end phase. The only limit of the current implementation is a limitation to instruction idioms which operate on one basic block. In our study, we found only one idiom that operates on two basic blocks. Transforming such an instruction idiom in a single-basic-block-at-a-time pass is impossible.

To conclude the experimental results, our new approach is capable of detecting and reconstructing instruction idioms for the common RISC and CISC architectures with a high accuracy (i.e. more than 99%), which is better than existing non-retargetable decompilers (some of them lacks this analysis as is demonstrated in Section 5).

By using the novel approach, we were able to increase the accuracy of reconstruction from 21% to 100% on the Intel x86 architecture, and from 70% to 99% in total for all architectures.

The future research lies in a further testing of the retargetable idiom detection and code reconstruction by using executables created by different compilers and for different target architectures (e.g. PowerPC). There is always a room for improvement by adding

new instruction idioms into our database of supported idioms. Finally, a usage of control-flow analysis for instruction-idioms detection may be useful when dealing with more aggressive optimizations.

Acknowledgments. This work was supported by the BUT grant FIT-S-14-2299 Research and application of advanced methods in ICT, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Boston, 2nd edn. (2006)
2. ARM Limited: ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition, ARM DDI 0406C edn. (2011), https://silver.arm.com/download/download.tm?pv=1199569
3. Beeler, M., Gosper, R.W., Schroeppel, R.: HAKMEM. Massechusetts Institute Of Technology (1972)
4. Boomerang: http://boomerang.sourceforge.net/ (2013)
5. Cifuentes, C.: Reverse Compilation Techniques. Ph.D. thesis, School of Computing Science, Queensland University of Technology, Brisbane, QLD, AU (1994)
6. Davidson, J.W., Whalley, D.B.: Quick compilers using peephole optimization. Software: Practice and Experience 19(1), 79–97 (1989)
7. Decompile-It.com – Online C Decompiler: http://decompile-it.com/ (2013)
8. DWARF Debugging Information Committee: DWARF Debugging Information Format, 4 edn. (2010), http://www.dwarfstd.org/doc/DWARF4.pdf
9. Emmerik, M.J.V.: Static Single Assignment for Decompilation. Ph.D. thesis, University of Queensland, Brisbane, QLD, AU (2007)
10. GCC: the GNU Compiler Collection: http://gcc.gnu.org/ (2013)
11. GNU ARM Toolchain: http://www.gnuarm.com/ (2012)
12. von Hagen, W.: The Definitive Guide to GCC. Apress (2006)
13. Hex-Rays Decompiler: www.hex-rays.com/products/decompiler/ (2013)
14. Hyde, R.: The Art of Assembly Language. No Starch Press, San Francisco, US-CA (2003)
15. IDA Disassembler: www.hex-rays.com/products/ida/ (2013)
16. Intel Corporation: Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture (2013), http://download.intel.com/products/processor/manual/253665.pdf
17. Křoustek, J., Kolář, D.: Object-file-format description language and its usage in retargetable decompilation. In: AIP Conference Proceedings (SCLIT'12). vol. 1479, pp. 466–469. American Institute of Physics (AIP) (2012)
18. Křoustek, J., Kolář, D.: Preprocessing of binary executable files towards retargetable decompilation. In: 8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13). pp. 259–264. International Academy, Research, and Industry Association (IARIA), Nice, FR (2013)
19. Křoustek, J., Matula, P., Končický, J., Kolář, D.: Accurate retargetable decompilation using additional debugging information. In: 6th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'12). pp. 79–84. International Academy, Research, and Industry Association (IARIA) (2012)
20. Křoustek, J., Matula, P., Ďurfina, L.: Generic plugin-based convertor of executable file formats and its usage in retargetable decompilation. In: 6th International Scientific and Technical Conference (CSIT'11). pp. 127–130. Ministry of Education, Science, Youth and Sports of Ukraine,

Lviv Polytechnic National University, Institute of Computer Science and Information Technologies (2011)
21. Křoustek, J., Pokorný, F.: Reconstruction of instruction idioms in a retargetable decompiler. In: 4th Workshop on Advances in Programming Languages (WAPL'13). pp. 1507–1514. IEEE Computer Society, Krakow, PL (2013)
22. Lissom: http://www.fit.vutbr.cz/research/groups/lissom/ (2013)
23. LLVM Assembly Language Reference Manual: http://llvm.org/docs/LangRef.html (2013)
24. Masařík, K.: System for Hardware-Software Co-Design. VUTIUM, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 1st edn. (2008)
25. Microsoft Corporation: Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx (2013), version 8.3
26. Minimalist PSPSDK: http://sourceforge.net/projects/minpspw/ (2013)
27. MIPS Technologies Inc.: MIPS32 Architecture for Programmers Volume II-A: The MIPS32 Instruction Set, MIPS MD00086 edn. (2010), https://www.mips.com/products/architectures/mips32/
28. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, US-CA (1997)
29. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, Cambridge, UK, 3rd edn. (2007)
30. Reverse Engineering Compiler (REC): http://www.backerstreet.com/rec/rec.htm (2013)
31. SmartDec: http://decompilation.info/ (2013)
32. Stallman, R.M., the GCC Developer Community: GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint.pdf (2010)
33. The dcc Decompiler: http://itee.uq.edu.au/~cristina/dcc.html (2013)
34. The LLVM Compiler Infrastructure: http://llvm.org/ (2013)
35. TIS Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (1995), http://refspecs.freestandards.org/elf/elf.pdf
36. Ďurfina, L., Křoustek, J., Zemek, P., Kábele, B.: Detection and recovery of functions and their arguments in a retargetable decompiler. In: 19th Working Conference on Reverse Engineering (WCRE'12). pp. 51–60. IEEE Computer Society, Kingston, ON, CA (2012)
37. Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T., Masařík, K., Meduna, A.: Advanced static analysis for decompilation using scattered context grammars. In: Applied Computing Conference (ACC'11). pp. 164–169. World Scientific and Engineering Academy and Society (WSEAS) (2011)
38. Warren, H.S.: Hacker's Delight. Addison-Wesley, Boston, US-MA (2003)

Jakub Křoustek is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc degree from the same university in 2009. He is currently working on the Lissom research project as the leader of the retargetable decompiler. His current research interests include reverse engineering, malware detection, and compiler design, with special focus on code analysis and reverse translation.

Fridolín Pokorný is a MSc student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his bachelor's degree from the same university in 2013. His research is focused on compilers and their code optimizations.

Dušan Kolář went to Brno University of Technology, Czech Republic, where he studied computer science and cybernetics and obtained his degrees in 1994 and 1998. Since then, he has been working at the university, presently at the Faculty of Information Technology. His main research interests are formal languages and automata and formal models with focus on their usage in compilers and formal models transformation.

Received: December 3, 2013; Accepted: July 7, 2014.