Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis

Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška,
Karel Masařík, and Alexander Meduna

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno, Czech Republic
{idurfina, ikroustek, izemek, kolar, hruska, masarik,
meduna}@fit.vutbr.cz

Abstract. Together with the massive expansion of smartphones, tablets, and other smart devices, we can notice a growing number of malware threats targeting these platforms. Software security companies are not prepared for such diversity of target platforms and there are only few techniques for platform-independent malware analysis. This is a major security issue these days. In this paper, we propose a concept of a retargetable reverse compiler (i.e. a decompiler), which is in an early stage of development. The retargetable decompiler transforms platform-specific binary applications into a high-level language (HLL) representation, which can be further analyzed in a uniform way. This tool will help with a static platform-independent malware analysis. Our unique solution is based on an exploitation of two systems that were originally not intended for such an application—the architecture description language (ADL) ISAC for a platform description and the LLVM Compiler System as the core of the decompiler. In this study, we show that our tool can produce highly readable HLL code.

Keywords: decompilation, reverse engineering, malware, LLVM, Lissom, ISAC

1 Introduction

There are two basic types of malicious software (a.k.a. *malware*) analysis—*dynamic analysis* and *static analysis*. Even though both types have the same objective—investigation of malware behavior—they differ in its accomplishment. Both methods are usually performed together to gain a better understanding of malware samples. In dynamic analysis, we inspect a run-time malware behavior by its monitoring (e.g. monitoring of WinAPI calls [1]) and we track changes of the system and network (e.g. registry modification, installation of new services, network communication, etc. [2, 3]).

To gain further insight of malware (e.g. inspection of malware functions, shell-code detection, etc.), static analysis is used. In this case, the code viewing and walking is done by several reverse-engineering tools like disassemblers or decompilers [4, 5]. The reverse translation of malware gives the analyst an opportunity

to see its source code, either on the assembly language level (disassembly), or on a higher level, such as the C language (decompilation). With such knowledge, it is possible to create an appropriate protection (e.g. signature of a malware sample for an anti-virus or a new heuristic method).

Static malware analysis is crucial because some kinds of malware cannot be properly analysed using only dynamic analysis (e.g. polymorphic and metamorphic code [6, 7]), and we will focus on it in the further text.

Malware was primarily targeted at personal computers (i.e. architectures x86 and x86-64 [8]) for the past 20 years. The techniques for malware analysis were well optimized for this platform during this time and security companies were able to keep pace with malware authors [9, 6].

The expansion of smart devices (e.g. smartphones, tablets, etc.) is very rapid last years [10]. Such devices are powered by various processors and running several types of operation systems. Users often use these new platforms for manipulating sensitive user data (e.g. passwords, credit card information, etc.), which comes to the attention of malware authors. Furthermore, the variety of these platforms is problematic for security companies because their solutions are mostly oriented on the classical ones, and they are not capable to protect new platforms at the moment. Those are the main reasons why the amount of malware for these platforms increases steadily for the last years.

In this paper, we present an overview of a retargetable decompiler, which is currently in an early stage of development. Our approach is not tied to any particular target platform. The primarily utilization of this tool is a static platform-independent malware analysis. With its help, it will be possible to inspect malware code on a much more abstract and unified form of representation, while preserving the functional equivalence of the code. Therefore, malware analysts do not need to have a deep knowledge of the target platform (i.e. instruction set and processor architecture) and they can fully focus on malware analysis.

The retargetable decompiler is based on exploitation of the ADL ISAC [11], which is intended to be used for designing new application-specific instruction set processors (ASIPs). However, we use this formalism for the description of existing platforms. The front-end of the decompiler is generated from this description. The decompiler core is based on the LLVM Compiler System [12], which we use for a reverse translation from a binary form into a Python-like language.

The paper is organized as follows. Section 2 discusses the state of the art of machine code decompilers. Then, Sections 3 and 4 describe the two key concepts our decompiler is based on—the ISAC language and the LLVM Compiler System. The design of our decompiler is then presented in Section 5. Experimental results are given in Section 6. Section 7 closes the paper by discussing future research.

2 State of the Art

The era of machine code decompilers was established before more than 50 years. The D-Neliac decompiler [13], built in 1960, was the first decompiler which

showed that it is feasible to develop programs working in a reverse way to compilers.

From this period, there have been a lot of attempts to create various kinds of decompilers. A project with the closest idea to our project is the PILER System [14]. This system, consisting of three parts, uses two intermediate representations. The first part is an interpreter, which has to be run on the source machine. Its output is *Micro Form*. Micro Form is a three-address low-level intermediate representation. This representation is processed by the second part—an analyzer. The analyzer exploits more analyses, such as data flow or timing analyses, to produce *Intermediate Form*. This form is a high-level intermediate representation designed to be suitable for FORTRAN, COBOL, and other languages of that time. The last part is a converter. It emits the source of code of the HLL. In its time, the PILER System was designed to be flexible and general. However, according to the available information, it was never completed.

Currently, there exist several decompilers which deserve to be mentioned. There are the dcc decompiler [15] from C. Cifuentes, the open source Boomerang decompiler [16], the REC Decompiler [17], and the Hex-Rays Decompiler [5]. The first two decompilers are not developed any more, but the second two are constantly improving. They are shortly described in the following paragraphs. The summary is shown in Table 1.

The dcc decompiler aims only to a single architecture and to a single operating system (i80286 MS-DOS executables), but it is well structured and it complexly shows and implements the most important algorithms for the reverse engineering process. It has the same structure as a compiler: there are a front-end, a middle-end, and a back-end. Every part has its own separate tasks. This decompiler uses also two types of an intermediate representation. The low one is for the communication between the front-end and the middle-end, and the high one is sent from the middle-end to the back-end, which finally transforms this representation into a C source code. Except these intermediate representations, the decompiler creates a control flow graph in the front-end, and this graph is used in the both other parts. The dcc decompiler also contains other tools which help to create a more readable result in the target HLL. The most important features of these tools are recognitions of compilers and library routines. According to the recognized compiler, the start-up code does not need to be decompiled. The same effect applies for library routines.

After the dcc decompiler was published, there was an idea to create a retargetable decompiler. This idea resulted in an open source project called Boomerang. Its main developer was M. van Emmerik. Boomerang is a retargetable decompiler with a modular architecture. This architecture is a base for its retargetability. The design is similar to dcc, but there is an emphasis on the modular principle for an easy substitution of every part. Boomerang currently supports input executables for x86 (except SIMD instructions), Sparc, and PowerPC processors, where binary file format can be either ELF or PE. The target language is C.

The REC decompiler is not open source, but it is available for free, and also the authors published the description of its design. From nowadays decompilers, it supports the most number of architectures and binary file formats. The supported processors are x86, x86-64, MIPS, m68k, Sparc, and PowerPC. The file format of an input executable can be PE, ELF, or MachO. The REC decompiler implements complex algorithms for control flow graphs. Therefore, it is able to reconstruct some advanced constructs, such as `switch` statements.

The Hex-Rays Decompiler represents a proprietary solution which can be bought as an add-on to the IDA Pro Disassembler [4]. Therefore, we lack detailed information about this solution. This decompiler has a well-developed recognition of compilers, start-up and statically-linked code. This scope is covered by Fast Library Identification and Recognition Technology (FLIRT) [18]. It provides tools for an easy addition of new library and compiler signatures, which can be subsequently used by the decompiler.

	dcc	Boomerang	REC decompiler	Hex-Rays Decompiler
Supported architectures	i80286	x86, Sparc, PowerPC	x86, x86-64, MIPS, m68k, Sparc, PowerPC	x86, ARM
Supported file formats	MS-DOS format	PE, ELF	PE, ELF, MachO	ELF, PE
Target language	C	C	C-like	C
Intermediate representation	two types (high and low)	a single type	?	?
Detects statically-linked code	Yes	No	Yes	Yes

Table 1. A comparison of the described decompilers.

On the other hand, there are projects which are not complete decompilers, but they are specialized only for generating a source code from some intermediate representation. As a nice example, we can mention emscripten [19]. It is a compiler able to transform LLVM bitcode to Javascript. It is used for transforming C/C++ code for running on the web. A similar project is llvm-js-back-end [20], which is a general LLVM back-end for producing Javascript code.

3 ISAC Language

The ISAC language [11] was developed within the Lissom project at Brno University of Technology [21]. The project has two basic scopes. The first scope is a development of an ADL for the description of Multiprocessor Systems-on-Chip (MPSoC). The second scope is a transformation of MPSoC description into advanced software tools (e.g. a C compiler, a simulator, etc.) or into a hardware

realization of each processor. The ISAC language belongs into a so-called mixed ADL. It means that a processor model consists of several parts. In the resource part, processor resources, such as registers or memory hierarchy, are declared. In the operation part, processor instruction set with behavior of instructions and processor micro-architecture is described. Processor model can be written in two levels of accuracy—instruction-accurate or cycle-accurate. The retargetable decompiler currently uses the first one.

The *assembler* and *coding* sections capture the format of instructions in the assembly and machine language, so they define instructions in textual and binary forms. For the behavioral model, the *behavior* section is used. In this section, a subset of the ANSI C language can be used. The behavior section defines the semantics of each operation. For example, a simple instruction with its behavior is described using the assembler, coding, and behavior sections, see Figure 1.

```
RESOURCES {                    // HW resources
  PC REGISTER bit[32] pc;      // program counter
  REGISTER bit[32] regs[16]; // register file
  RAM bit[32] memory {SIZE(0x10000); FLAGS(R, W, X); };
}
OPERATION reg REPRESENTS regs
  { /* textual and binary description of registers */ }
OPERATION op_add {             // instruction description
  INSTANCE reg ALIAS {rd, rs, rt};
  ASSEMBLER { "ADD" rd "=" rs "," rt };
  CODING { 0b0001 rd rs rt };
                              // instruction behavior
  BEHAVIOR { regs[rd] = regs[rs] + regs[rt]; };
}
```

Fig. 1. Example of a ISAC language source code.

4 LLVM Compiler System

The LLVM Compiler System [12] was originally designed as a compiler framework to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time and in idle-time between runs [22]. Nowadays, the use of LLVM spans over many different areas, including compilation [23–26], video decoding [27], signal processing [28], static checking [29–32], and implementation of various programming languages [33–35]. The key features of LLVM include a universal, language-independent instruction set, type system,

intermediate representation (LLVM IR [36]), many built-in sophisticated optimization algorithms and passes, link-time optimizations, just-in-time (JIT) code generation, and application programming interface for several programming languages.

Consider the C source code in Figure 2. This straightforward implementation of the factorial function can be directly compiled into the LLVM IR. The output of this conversion is shown in Figure 3. This example shows us some of the properties of the LLVM IR:

- The used RISC-like instruction set captures the key operations of ordinary processors, but avoids most of machine-specific constraints. Most instructions are in the three-address form—they take either one or two operands and produce a single result. The instruction set includes arithmetic instructions (e.g. `add`, `mul`), bitwise instructions (e.g. `shl`, `and`), memory access instructions (e.g. `load`, `store`, `alloca`), conversion instructions (e.g. `trunc`, `zext`), and other instructions (e.g. `icmp`, `call`). Furthermore, every basic block ends with a terminator instruction (e.g. `br`, `ret`) which explicitly specifies its successor basic blocks.
- As can be seen from the presence of the `phi` instruction in Figure 3, the virtual registers are in the Static Single Assignment (SSA) form (see [37]), where each variable is assigned exactly once. The use of this form results in a simplification of many compiler optimizations.
- A language-independent type system is used. Every instruction and SSA register has an associated type and all operations obey strict type rules. This enables several optimizations which otherwise would not be possible (at least not in such a straightforward way). Primitive types include void, boolean, variable-sized integers, and floating-point types. Derived types include pointers, arrays, structures, and functions. The `cast` instruction can be used for type conversions (other ways of type conversions are not possible). Address computation and address arithmetic is done by the `getelementptr` instruction.
- The LLVM IR can exist in the following three forms: textual (as in Figure 3), binary (compiled textual representation), and in-memory (compiler internal representation). All of these representations are equivalent—that is, one can be transformed to the others without any loss of information.

```
int factorial(int n) {
    if (n == 0)
        return 1;
    return n*factorial(n-1);
}
```

Fig. 2. A simple implementation of the factorial function in C.

```
define i32 @factorial(i32 %n) {
entry:
    %0 = icmp eq i32 %n, 0
    br i1 %0, label %bb2, label %bb1

bb1:
    %1 = add i32 %n, -1
    %2 = icmp eq i32 %1, 0
    br i1 %2, label %factorial.exit, label %bb1.i

bb1.i:
    %3 = add i32 %n, -2
    %4 = call i32 @factorial(i32 %3)
    %5 = mul i32 %4, %1
    br label %factorial.exit

factorial.exit:
    %6 = phi i32 [ %5, %bb1.i ], [ 1, %bb1 ]
    %7 = mul i32 %6, %n
    ret i32 %7

bb2:
    ret i32 1
}
```

Fig. 3. The generated LLVM IR code from the code in Figure 2.

5 Design of a Retargetable Decompiler

The objective of the decompiler is an analysis of a binary code and its transformation into a HLL. It is important to preserve the functional equivalence of the transformed program; otherwise, further code analyses will be inaccurate. This is a very difficult task because we have to deal with missing information in the input code (e.g. because of compiler optimizations, malware obfuscation, etc.). The usage of the retargetable decompiler is straightforward—its user describes the target architecture in the ISAC ADL and the decompiler is automatically generated by a tool-chain generator based on this description. After that, it is possible to reversely translate binary executables for this architecture. The idea of this process is discussed in the following text.

The structure of the retargetable decompiler is similar to a classical compiler. It consists of a front-end, a middle-end, and a back-end, see Figure 4. The

only platform-specific part is the front-end. For this purpose, the binary coding and semantics of each processor instruction is extracted from the architecture model in ISAC. This is a major difference against other retargetable decompilers, because it is not necessary to manually reconfigure the decompiler for a new architecture. It should be noted that in present, there is no other competitive method of automatically-generated retargetable decompilation.

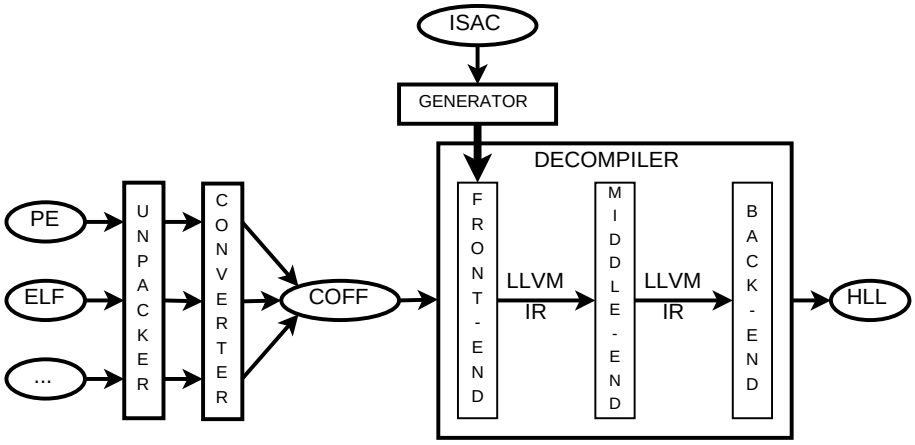

Fig. 4. The concept of the retargetable decompiler.

5.1 Front-end

The objective of the front-end is a translation from an architecture-specific machine code into a sequence of low-level LLVM IR instructions. The input binary file is stored in a platform-specific file format (e.g. Windows PE, Unix ELF, etc.). Furthermore, in the case of malware, the input file is usually packed and protected against reverse-engineering. Therefore, the first step of the reverse translation is a code unpacking phase. As this topic is well documented in the literature (see [38, 39]), we will not deal with it in our paper.

After that, it is necessary to convert the platform-specific file format into a unified form of representation. The internal COFF-based file format has been designed for this purpose, together with several conversion algorithms. At the moment, we support conversions from Windows PE, Unix ELF, Symbian E32, and Android DEX file formats.

To generate the instruction decoder (i.e. a part of the front-end responsible for the conversion from a machine code into LLVM IR), it is necessary to extract the binary coding and semantics of instructions from the ISAC model. This task is done via a tool called *semantics extractor* [40]. The semantics extractor transforms ANSI C code from the behavior section of the instruction into a sequence

of LLVM IR instructions, which properly describes its behavior. Therefore, we are able to map instruction semantic in LLVM IR to its machine code.

After that, it is possible to automatically generate an instruction decoder. Its functionality is similar to a disassembler, except its output is not an assembly language representation of the instruction, but rather a sequence of LLVM IR instructions (i.e. a basic block with several instructions). The instruction decoder is based on a formal model [41]. Whenever a statically-linked code is detected, its representation is emitted instead of simple instruction semantics.

The design of our solution for statically-linked code detection is inspired by FLIRT [18]. In the whole process, there are separate steps which start from taking the static library and finish at creating a signature for this library. Static libraries contain object files with different formats. This problem is solved by the same way as it is solved for executable files. Object files are extracted from libraries, they are transformed into our object file format and finally, they are packed into a single archive. Due to this action, we can then proceed with a single file format. In comparison with FLIRT, this clearly represents an advantage because we do not have to build separate tools for each object file format. Indeed, we can easily extend the tool for transformation.

The creation of signatures consists of two parts:

- extracting the pattern for each object file from a library,
- building the signature from a group of patterns.

The process is visualized in Figure 5.

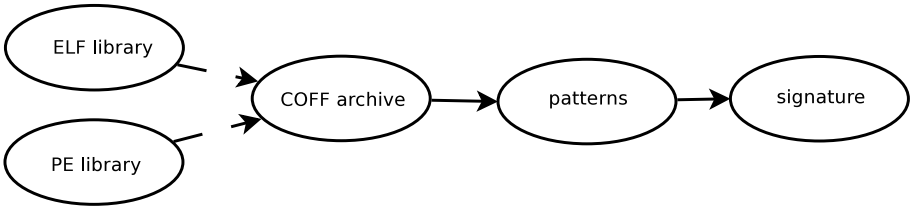

Fig. 5. The process of creating the signature for static library

The final part of the front-end is a static analysis of the emitted LLVM IR code. It is focused on control flow analyses and transformation of the LLVM IR code to produce more suitable code for the following parts of the decompiler. For example, this phase is responsible for the detection of functions, elimination of "jump register" instructions, and annotation of the resulting LLVM IR code. However, this part is not implemented yet, and more intensive research in this field is necessary.

5.2 Middle-end

In this stage, we have a very low-level LLVM IR of the input binary. Each basic block represents a single assembly instruction, there may be many redundant

instructions (recall that each assembly instruction is decompiled in isolation), and there is no evidence of high-level constructs, such as loops. The key role of the middle-end part of our decompiler is to improve the properties of the generated LLVM IR code and prepare it for the final emission of the output HLL.

The following passes are performed over the LLVM IR code:

- Search for idioms. These are sequences of code whose combined semantics is not immediately apparent from the instructions' individual semantics. For example, `xor`'ing a single register with itself can be replaced by assigning a zero into it. This form is clearly more readable than the original form. Idioms may, however, span over several instructions. In such cases, they may be replaced with a few equivalent instructions. This pass also includes several other types of program analysis, such as constant or expression propagation.
- Retrieval of high-level constructs, such as `if/else` statements, `switch`es, and loops. As we want to use unstructured `goto`'s as little as possible, it is necessary to identify proper structured equivalents. We identify headers and bodies of loops, common destination basic blocks after a branch instruction, and other information needed to generate natural and readable output code.
- Optimization of the code. Here, we connect successive basic blocks and perform optimizations which eliminate redundant instructions.

As LLVM already includes many worked-out and sophisticated optimization and analysis passes, we naturally prefer using them over our own passes whenever possible.

To improve this middle phase in terms of effectiveness, the LLVM IR code generated by our front-end is annotated. We utilize the fact that LLVM IR allows metadata to be attached to instructions in the program that can convey extra information about the code. The used annotations include markings of the entry point of function bodies and information about application binary interface (calling conventions, system calls, etc.).

5.3 Back-end

In this final decompilation stage, we transform the optimized LLVM IR into a HLL. We currently use a Python-like (see [42]) language as the target language, briefly described next. However, a support for different back-ends is planned.

Our HLL is non-typed, block structured, and uses whitespace indentation, rather than curly braces or keywords, to delimit blocks. Since we focus on code analysis by humans, the used language emphasises code readability. Whenever there is no support in Python for a specific construction, we use C-like constructs. For example, we use C-like `switch` statements to implement the fall-through feature of C. Instead of arrays, we use lists, and instead of structures, we utilize dictionaries. We also use the address and dereference operators from C. As there are cases when the code cannot be structured by high-level constructs only (for example, an irreducible subgraph of the control-flow graph is detected [15]), an explicit `goto` represents a necessary addition to our language.

The implemented back-end takes the optimized LLVM IR and converts it into our HLL by walking over its control-flow graph. Analysis information from the middle-end is used throughout the generation to emit proper high-level constructs.

After the generation is completed, an additional post-processing phase is done to further improve the readability of the code. These modifications are done on a textual level, and include the elimination of redundant brackets and expressions introduced by the back-end.

In the next section, we present an example of a generated output code.

6 Experimental Results

In this section, we present some experimental results. Since our decompiler is in an early stage of development, we just compare our back-end with the C back-end, currently included in LLVM. Even though we have several examples of a preliminary decompilation of a real-world code, the obtained results are not in a publishable form yet.

First, we compile the source code given in Figure 6 directly into LLVM IR using `llvm-gcc`[1]. Due to space requirements, we omit the listing of the obtained LLVM IR code. Then, we emit a high-level representation of it by the C back-end and by our back-end (see Figures 7 and 8, respectively).

Observe the following key differences between the two back-ends:

- We do not introduce any useless variables (see `llvm_cbe_tmp__1` in the output from the C back-end).
- Instead of using `goto`'s, we inline the bodies of the corresponding basic blocks wherever possible. This also holds for other high-level constructs, such as loops and `switch` statements.
- Instead of accessing string literals through a structure, we inline them wherever possible.
- We eliminate as much redundant code as possible, hence increasing the readability of the resulting code. For example, consider the redundant pairs of brackets in the output from the C back-end. As they are not needed and they also do not contribute to the understandability of the code (like when brackets are used in complex expressions with many different operators), we remove them.
- As our HLL is non-typed, there is no need for any sequences of castings which make the generated code—albeit type correct—less readable.
- Finally, we generate as little boilerplate code as possible (however, this cannot be seen from the figures).

```
int func(int a, int b) {
    if (a != b) {
        printf("%d != %d", a, b);
    }
    return a * (b + 10);
}
```

Fig. 6. An input source code for our comparison of back-ends.

```
// Removed boilerplate code
struct l_unnamed0 { unsigned char array[9]; };
static struct l_unnamed0 _OC_str = { "%d != %d" };

unsigned int func(unsigned int llvm_cbe_a,
                  unsigned int llvm_cbe_b) {
    unsigned int llvm_cbe_tmp__1;

    if ((llvm_cbe_a == llvm_cbe_b)) {
        goto llvm_cbe_bb1;
    } else {
        goto llvm_cbe_bb;
    }

    llvm_cbe_bb:
        llvm_cbe_tmp__1 = printf(((&_OC_str.array[((signed
            long long)0ull)])), llvm_cbe_a, llvm_cbe_b);
        goto llvm_cbe_bb1;

    llvm_cbe_bb1:
        return (((unsigned int)(((unsigned int)(((unsigned
            int)(((unsigned int)llvm_cbe_b) + ((unsigned
            int)10u))))) * ((unsigned int)llvm_cbe_a))));
}
```

Fig. 7. Truncated output from the C back-end.

```
def func(a, b):
    if not (a == b):
        printf("%d != %d", a, b)
    return (b + 10) * a
```

Fig. 8. Output from our back-end.

7 Conclusion

This paper proposed the concept of a retargetable decompiler for a static platform-independent malware analysis. The front-end of this tool is generated based on a processor description in the ISAC ADL. Its middle-end and back-end are build on top of the LLVM Compiler System. The idea of exploitation of these systems is innovative and it allows an automatic retargetability of our solution. This is a major advantage of our solution over other similar projects.

The functionality of each part of the decompiler was discussed and we presented results of the current state of the implementation. As can be seen, we are already able to convert a platform-specific machine code into its semantic representation as a platform independent LLVM IR code. Such code can be translated into a human-readable HLL code by our back-end after that. This back-end can achieve better results in a reverse translation than other existing solutions.

However, there is still a lot of space for improvements. In the first place, it is necessary to finish the implementation of the static analyser in the front-end. After that, we will be able to produce a more accurate LLVM IR code, which will improve the resulting HLL code.

The middle-end of our decompiler can be enhanced by several new analyses and transformations. For example, consider the `if` condition in Figure 8. A more natural way of representing the condition would be to transfer the negation into the expression, resulting into `if a != b`. Another transformation to be considered is a replacement of a chain of `if/else if` statements by a single `switch` statement. Furthermore, current analyses and transformations have to be improved to generate better HLL.

As for the back-end of our decompiler, emission of some HLL constructs can be improved. For example, loops are currently always generated as `while True` loops. A proper detection of induction variables in the middle-end is necessary to emit more natural constructions. Furthermore, other HLLs can be considered, possibly including type information and conversions between types.

Acknowledgments

This work was supported by the research fundings MPO ČR, No. FR-TI1/038, TAČR, No. TA01010667, BUT FIT grant FIT-S-11-2, by the Research Plan No. MSM0021630528, and by the SMECY European project.

[1] We used `llvm-gcc` version 4.2.1. The source code from Figure 6 was compiled with enabled optimizations (`-O3`) directly into LLVM IR (`-emit-llvm`). The used version of LLVM is 2.8.

References

1. Xu, J., Sung, A.H., Chavez, P., Mukkamala, S.: Polymorphic malicious executable scanner by API sequence analysis. In: Fourth International Conference on Hybrid Intelligent Systems. (2004) 378–383
2. Wagener, G., State, R., Dulaunoy, A.: Malware behaviour analysis. Journal in Computer Virology **4** (2008) 279–287
3. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security and Privacy **5**(2) (2007) 32–39
4. IDA Pro Disassembler. [Online] (Available: `http://www.hex-rays.com/idapro/`)
5. Hex-Rays Decompiler. [Online] (Available: `http://www.hex-rays.com/decompiler.shtml`)
6. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Upper Saddle River, US-NJ (2005)
7. Szor, P., Ferrie, P.: Hunting for metamorphic. In: Virus Bulletin Conference. (2001) 123–144
8. Intel Corporation: Intel 64 and ia-32 architectures software developer's manual volume 1: Basic architecture (2011)
9. Aquilina, J.: Malware Forensics Investigating and Analyzing Malicious Code. Syngress Publishing, Burlington, US-MA (2008)
10. International Data Corporation (IDC): Worldwide quarterly mobile phone tracker (2011)
11. Masařík, K.: System for Hardware-Software Co-Design. 1st edn. Faculty of Information Technology BUT, Brno, CZ (2008)
12. The LLVM Compiler System. [Online] (Available: `http://llvm.org/`)
13. Halstead, M.H.: Machine-Independent Computer Programming. Spartan Books (1962)
14. Barbe, P.: The PILER system of computer program translation. Technical report, Probe Consultants Inc (1974)
15. Cifuentes, C.: Reverse Compilation Techniques. PhD thesis, School of Computing Science, Queensland University of Technology, Brisbane, AU-QLD (1994)
16. Boomerang. [Online] (Available: `http://boomerang.sourceforge.net/`)
17. Reverse Engineering Compiler. [Online] (Available: `http://www.backerstreet.com/rec/rec.htm`)
18. Fast Library Identification and Recognition Technology. [Online] (Available: `http://www.hex-rays.com/idapro/flirt.htm`)
19. emscripten. [Online] (Available: `http://code.google.com/p/emscripten/`)
20. llvm-js-backend. [Online] (Available: `http://github.com/dmlap/llvm-js-backend`)
21. Lissom Project. [Online] (Available: `http://www.fit.vutbr.cz/research/groups/lissom/`)
22. Adve, V., Lattner, C., Brukman, M., Shukla, A., Gaeke, B.: LLVA: A low-level virtual instruction set architecture. In: Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture, San Diego, US-CA (2003)
23. Clang. [Online] (Available: `http://clang.llvm.org/`)
24. LDC: LLVM D Compiler. [Online] (Available: `http://www.dsource.org/projects/ldc`)
25. Trident Compiler. [Online] (Available: `http://trident.sourceforge.net/`)
26. Tripp, J.L., Gokhale, M.B., Peterson, K.D.: Trident: From high-level language to hardware circuitry. Computer **40**(3) (2007) 28–37

27. Just-In-Time Adaptive Decoder Engine (Jade). [Online] (Available: http://sourceforge.net/apps/trac/orcc/)
28. Faust: Signal Processing Language. [Online] (Available: http://sourceforge.net/projects/faudiostream/)
29. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: CAV'07: Proceedings of the 19th International Conference on Computer Aided Verification, Berlin, DE (2007) 371–383
30. Babić, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, Leipzig, DE (2008) 211–220
31. Calysto Extended Static Checker. [Online] (Available: http://www.domagoj-babic.com/index.php/ResearchProjects/Calysto)
32. Lewycky, N.: Checker: A static program checker. Master's thesis, Computer Science Department, Ryerson University, Toronto, CA-ON (2006)
33. unladen-swallow: A Faster Implementation of Python. [Online] (Available: http://code.google.com/p/unladen-swallow/)
34. Rubinius. [Online] (Available: http://rubini.us/)
35. The Pure Programming Language. [Online] (Available: http://code.google.com/p/pure-lang/)
36. Adve, V., Lattner, C.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, Palo Alto, US-CA (2004) 75–86
37. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13**(4) (1991) 451–490
38. Coogan, K., Debray, S.K., Kaochar, T., Townsend, G.M.: Automatic static unpacking of malware binaries. In: Working Conference on Reverse Engineering, Lille, FR (2009) 167–176
39. Yan, W., Zhang, Z., Ansari, N.: Revealing packed malware. IEEE Security and Privacy **6**(5) (2008) 65–69
40. Husár, A., Trmač, M., Hranáč, J., Hruška, T., Masařík, K., Kolář, D., Přikryl, Z.: Automatic C compiler generation from architecture description language ISAC. In: 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Brno, CZ, Masaryk University (2010) 84–91
41. Hruška, T., Kolář, D., Lukáš, R., Zámečníková, E.: Two-way coupled finite automaton and its usage in translators. In: New Aspects of Circuits. Volume 2008., Heraklion, GR (2008) 445–449
42. Python Programming Language. [Online] (Available: http://www.python.org/)