Acta Informatica Pragensia
2(1), 2013, 18–29, DOI: 10.18267/j.aip.10
Online: aip.vse.cz

Section:
Peer-reviewed papers

Approaching Retargetable Static, Dynamic, and Hybrid Executable-Code Analysis

Jakub Křoustek[1], Dušan Kolář[1]

[1] IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2, 612 66 Brno, Czech Republic

{ikroustek, kolar}@fit.vutbr.cz

Abstract: Program comprehension and reverse engineering are two large domains of computer science that have one common goal – analysis of existing programs and understanding their behaviour. In present, methods of source-code analysis are well established and used in practice by software engineers. On the other hand, analysis of executable code is a more challenging task that is not fully covered by existing tools. Furthermore, methods of retargetable executable-code analysis are rare because of their complexity. In this paper, we present a complex platform-independent toolchain for executable-code analysis that supports both static and dynamic analysis. This toolchain, developed within the Lissom project, exploits several previously designed methods and it can be used for debugging user's applications as well as malware analysis, etc. The main contribution of this paper is to interconnect the existing methods and illustrate their usage on the real-world scenarios. Furthermore, we introduce a concept of a new retargetable method – the hybrid analysis. It can eliminate the shortcomings of the static and dynamic analysis in future.

Keywords: Debugger, Decompiler, Reverse Engineering, Lissom

1 INTRODUCTION

As E. W. Dijkstra said years ago: "*If debugging is the process of removing bugs, then programming must be the process of putting them in.*". Moreover, software development is getting more tricky since applications are being developed for a wide range of target platforms (computers running x86(-64) processors, smart devices with ARM multi-cores, consumer electronics with smaller chips, etc.) where the toolchain (e.g. compiler, disassemble, simulator) can be incomplete or not properly tested (e.g. automatically generated compiler, experimental target-specific optimizations), especially for the newly created platforms such as application-specific instruction-set processors (ASIPs).

With this diversity of target architectures and operating systems, it is not easy to properly analyze and debug your code because it is highly probable that the appropriate analytical tool do not support such particular target platform.

Roughly speaking, analysis of the source code is easier since it is more or less platform independent. Only the basic information about the target architecture (e.g. bit-width, endianity) is usually enough. As an example we can mention Coverity SAVE [14] that can be used for retargetable source code verification. However, there are two major scenarios where source-code analysis cannot be used.

> (1) Analysis of applications distributed in the form of executable files (abbreviated as executables in the following text) without source codes. This kind of analysis is useful for malware analysis, testing of the third-party commercial software, etc.
>
> (2) Whenever we need to examine executable versions of our own software, e.g. testing automatically generated compiler used for application's compilation, validation of the code optimized after compilation (optimizing linkers, postpass code optimization tools like AbsInt [5], etc.). Within this analysis, source code is available as well as additional information like symbols or debugging information.

In the following text, we present a complex retargetable framework for executable-code analysis, which is capable of both static and dynamic analysis. This framework was successfully implemented and tested within the Lissom project [10].

The motivation of this paper is to demonstrate both approaches on the real-world scenarios described in the previous paragraph. Furthermore, we highlight their drawbacks and we present their enhancement – the hybrid analysis. This approach combines advantages of the current methods but it is not implemented yet.

The paper is organized as follows. Section 2 discusses the state of the art of executable-code analysis. The description of the retargetable approach developed within the Lissom project is given in Section 3. Afterwards, we evaluate our methods of static and dynamic analysis on the real-world scenarios in Section 4. In Section 5, we introduce a new type of analysis, the hybrid analysis, and examples of its usage. Finally, Section 6 contains conclusion of this paper and discussion about the future research.

2 STATE OF THE ART

In this section, we discuss the state of the art of executable-code analysis. In present, we can find several common executable file formats – UNIX ELF, WinPE, Mac OS-X Mach-O, etc. These files may contain debugging information in one of the modern existing standards – DWARF or Microsoft PDB, for more details about theses formats and their processing see [7].

In order to support their accurate and comfortable analysis, it is important to provide appropriate tools. We can find two basic groups of tools – static and dynamic.

(1) Static executable-code analysis examines applications without their execution and it is usually focused on the program transformation from a machine code into a human-readable form of representation. The most common tools are disassemblers, which produce assembly code, or the more modern decompilers producing a high-level language (HLL) code (C, Java, etc.). Static analysis can also perform control-flow and data-flow checking, see [6] for more details.

(2) Dynamic analysis executes the application and it analyses its behavior during the run-time. The execution can be done either on the target architecture (e.g. remote debugging) or via simulation or emulation. Profilers, simulators, and debuggers are the most often used tools within this category.

The platform-specific analysis of executables is a partially solved problem, because implementation of such tool is a straightforward task [8, 15]. The existing toolchains usually contain both dynamic and static tools. For example the GNU Compiler Collection together with the Binutils package contains a profiler, disassembler, and debugger. Another example is the Microsoft Visual Studio. Nevertheless, program decompilation is still rare because even platform-dependent decompilers are hard to create [2, 3].

The retargetable analysis (i.e. the target architecture can be easily changed) is a more challenging task. We can find several projects focused on a rapid ASIP design that supports quality dynamic analysis (i.e. simulation and debugging of the ASIP's applications), but with a very limited static analysis (only the disassemblers are supported in general). All of these projects exploit its own architecture description language (ADL), which has been developed within the project, for the toolchain generation.

For example, the xADL project, developed at the Vienna University of Technology, supports generation of three types of simulators, but the support of debugging and static-analysis is missing. In other projects (ArchC, Sim-nML, EXPRESSION, LISA, etc), the situation is very similar – they support various types of automatically-generated simulators and debuggers, but the static analysis is very limited [8].

In past, there were only two attempts to create a retargetable decompiler – the PILER System (1976) [1] and the Boomerang project (2007) [16]. However, none of them was entirely completed and they are no longer actively developed.

3 RETARGETABLE STATIC AND DYNAMIC EXECUTABLE-CODE ANALYSIS

Our solution is developed as a part of the Lissom project located at Brno University of Technology [10]. The project is primarily focused on the design and development of the new ASIPs, their applications, and their interconnection within large Multiprocessor Systems-on-Chips (MPSoC).

A complete toolchain is developed within this project. It is automatically generated based on the target architecture description. The ISAC architecture description language [10] is used for this purpose. Each ISAC processor model consists of several parts. In the resource part, processor resources, such as registers or memories, are declared. In the operation part, processor instruction set with behavior of instructions is specified. The *assembler* and *coding* sections capture the format of instructions in the assembly and machine language, respectively. *Behavior* section is used for description of the instruction semantics via the ANSI C code, see Fig. 1 for illustration.

```
RESOURCES {                       // HW resources
  PC REGISTER bit[32] pc;         // program counter
  REGISTER bit[32] gpregs[16];    // register file
  RAM bit[32] memory {...};       // memory, etc.
}

// instruction set description
OPERATION op_sub {
    INSTANCE gpregs ALIAS {rd, rs, rt};
    ASSEMBLER { "SUB" rd "," rs "," rt };
    CODING { 0b1111 rs rt rd };
    // instruction’s behavior description
    BEHAVIOR { regs[rd] = regs[rt] - regs[rs]; }
}
```

Fig. 1. Example of a simple processor description in the ISAC language.

Size of the ISAC models varies based on the architecture’s complexity; see Tab. 1 several examples. The RISC (Reduced Instruction Set Computer) architectures (MIPS, ARM, SPARC, etc.) are usually easier to describe than CISC (Complex Instruction Set Computer), such as Intel x86. ISAC is also capable to describe VLIW (Very Long Instruction Word) architectures (VEX, ADI Blackfin, etc.) as well as the highly-parallel architectures like multi-core processors, or multiprocessor systems on chip (MPSoC) [12].

	MIPS	ARM	VEX	x86
Lines of ISAC code	3300	3400	1500	7000
Lines of auxiliary C code	1500	2000	800	2600
Total	4800	5400	2300	9600

Tab. 1. Complexity of the ISAC models for different architecture types.

The ISAC models are used for an automatic toolchain generation, e.g. C compiler, assembler, and analytical tools – decompiler, debugger, and simulators, see Fig. 2.

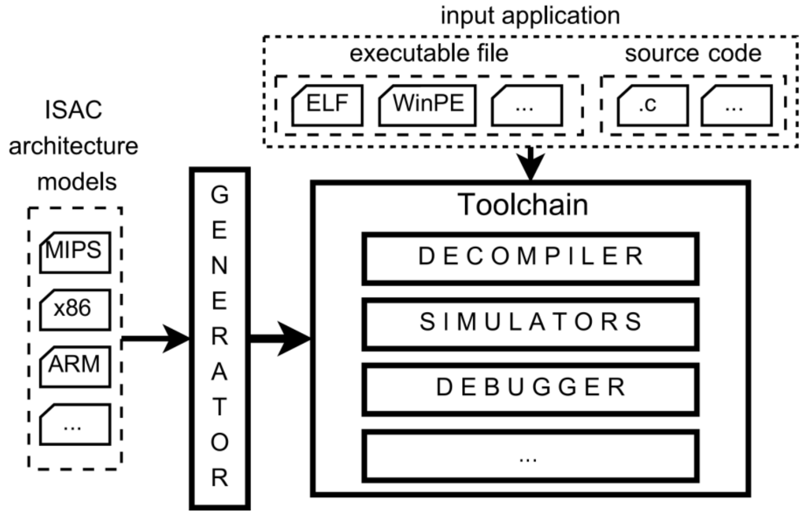

Fig. 2. Tools automatically generated based on the ISAC models.

In present, three main types of simulators are supported – interpreted, compiled, and translated. The *interpreted simulator* performs constant fetching, decoding, and executing instructions from memory. This approach is relatively slow but it is independent on the simulated application. On the contrary, the *compiled simulator* is dependent on a particular application because it is generated based on its content. However, it is faster than the former one because the repetitive tasks (e.g. fetching, decoding) are pre-computed during simulator creation. Finally, the *translated simulator* further enhances the concept of the compiled simulator via usage of debugging information [7]. Based on this information, we are able to extract information about the *basic blocks* (BB) contained within the application. Afterwards, the translated simulator is capable to execute instructions within the same BB in a burst mode (e.g. it is possible to skip several checks for these instructions). Therefore, its speed is even higher than the compiled simulator. However, the presence of debugging information is mandatory in this case.

Each simulator type suites for a different usage, see [12, 13] for details. The debugger allows dynamic analysis on the different levels (e.g. microarchitecture, instruction level, and source-level) [8]. The first two levels do not need any additional information, while the source-level debugger needs debugging information.

The retargetable decompiler is depicted in Fig. 3. At first, the input binary executable is transformed into an internal COFF representation. Afterwards, such file is processed in the *front-end* part, which is partially automatically generated based on the ISAC model. This phase decodes machine-code instructions into the LLVM IR representation [9], which is used as an internal code representation of decompiled applications in the remaining decompilation phases. In the *middle-end* part, this LLVM IR code is heavily optimized using optimization passes. Finally, the resulting HLL code is emitted in the *back-end* part. Currently, the Python-like language and the C language are supported. In present, the retargetable decompiler allows decompilation of MIPS, ARM, and Intel x86 executables. The detailed description can be found in [3, 4].

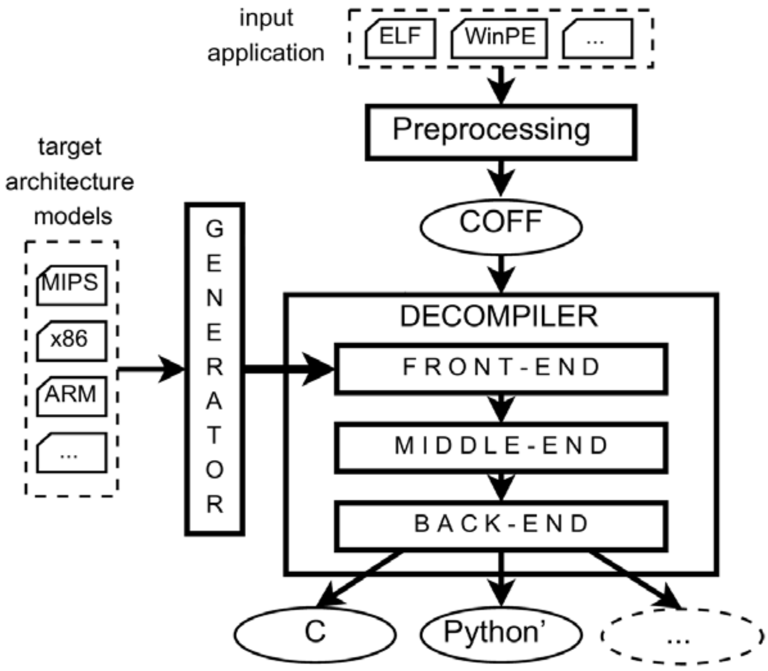

Fig. 3. Retargetable decompiler developed within the Lissom project.

4 STATIC AND DYNAMIC ANALYSIS IN A REAL-WORLD SCENARIOS

In this section, we present several scenarios of a retargetable executable-code analysis. We use the previously described tools for the static and dynamic analysis of these executables.

For evaluation, we use a MIPS architecture that is described using the ISAC language on the instruction-accurate level. MIPS is a 32-bit RISC processor architecture. The processor description is based on the MIPS32 Release 2 specification [11].

Several open-source applications and algorithms from benchmarking test-suites were used for testing. The Minimalist PSPSDK compiler (v 4.3.5) was used for their compilation into the MIPS-ELF executables. Different versions of optimizations were used. The testing was performed on Intel Core2 Quad with 2.8 GHz, 4 GB RAM running a Linux-based 64-bit operating system. The gcc (v4.6) compiler with optimizations enabled was used for the creation of all tools.

4.1 ONLY THE EXECUTABLE AVAILABLE

In this situation, we only have a naked (stripped) executable. Neither additional information nor its source code is available. This is the usual scenario of the third-party-software distribution or malware. Dynamic analysis can be done using interpreted or compiled simulation. The average speeds of interpreted and compiled simulators for MIPS are 27MHz and 55MHz, respectively [12].

In the scenario of the interpreted simulation, the simulator is automatically generated based on the target architecture model in ISAC (i.e. MIPS model). This simulator is capable to simulate the input application instruction by instruction. According to the description in Section 3, the compiled simulator needs to be generated for this particular input file. No additional information (e.g. debugging information) is necessary for creation and usage of these two simulators.

On the other hand, the translated simulator is not available, since we do not have information about basic blocks of the original program. Debugging is also limited to the instruction level. The source-level debugging is disabled because the debugging information is also unavailable.

Decompilation of the input binary executable is always possible, no matter if the source code or debugging information is present. The input application is analyzed and the target architecture, file format, and operating system are detected in preprocessing phase at first. In the same phase, we try to detect the originally used compiler using the signature-based detection. This is useful in later phases for generation of a more accurate code. The final step of the preprocessing phase is the conversion into the COFF format as described in Section 3.

Afterwards, the front-end phase is automatically generated based on the detected target architecture using the appropriate ISAC model. The machine code stored in the COFF file is transformed into its semantics representation in the LLVM IR format using the instruction decoder. In the next-step, the front-end performs several static analyses over the LLVM IR code, e.g. detection of functions, removal of statically-linked code, control-flow analysis, recovery of data types [3]. The final phase of front-end is the generation of the proper LLVM IR code that is passed to the middle-end.

Middle-end is responsible for code reduction and elimination of unneeded parts via several built-in optimization passes. Finally, the results, still in the LLVM IR form, are processed in the back-end part. It tries to detect and reconstruct the HLL constructions, such as loops, function calls, switch statements, etc.

However, the output can be further enhanced using debugging information for the better readability (e.g. variable and function names, position of code, source-code names, etc.), see the next scenario for details.

4.2 EXECUTABLE WITH DEBUGGING INFORMATION

This situation is less common, but still plausible, e.g. distribution of beta-versions of applications to testers. Debugging information can be exploited for generating translated simulator that can achieve higher speeds (up to 120 MHz, in average 113MHz [12]). The simulation speed drop off in debugging mode is about 1-12% based on the number of breakpoints and their positions within machine-code [8].

In theory, source-level debugging is allowed because debugging information is available (e.g. mapping of code to the HLL statements, locations of variables); however, the source code is unavailable and therefore, it is not possible to visualise the run-time information to the user. This is one of the limitations of the state of the art. However, this issue is addressed in Section 5 and its solution is proposed.

Decompilation can achieve the best results in this case. An example of source-code recovery using debugging information can be seen in Fig. 4. In this example, the decompiler was able to reconstruct even most of the local variable names, which are usually not available because they are removed by compiler. The retargetable decompiler is capable to extract and reconstruct many program constructions, e.g. file names (modules), function names, names and types of function arguments, global and local variable names and types, positions of these constructions within modules (e.g. order of originally used functions).

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[])
{
  int iter;
  double x, y, z;

  printf("Iterations: ");
  scanf("%d",&iter);

  /* initialize random numbers */
  srand(123456789);

  /* cnt = number of points in the
     1st quadrant of unit circle */
  int cnt = 0;
  int i = 0;
  while (i++ < iter){
    x = (double)rand() / RAND_MAX;
    y = (double)rand() / RAND_MAX;
    z = x*x + y*y;
    if (z <= 1) cnt++;
  }
  double pi=(double)cnt / iter * 4;
  printf("pi is %g\n", pi);
  return 0;
}
```

```c
#include <stdint.h>

    #include <stdio.h>

    #include <stdlib.h>

typedef int int32_t;
typedef double float64_t;

int main(int argc, char **argv) {
  int32_t cnt, i, iter;
  float64_t y, x, y2, x2, x3, lemon;
  cnt = i = iter = 0;

  printf("Iterations: ");
  scanf("%d", &iter);
  lemon = (float64_t) 2147483647;
  srand(123456789);
  while (iter >= i) {
    x3 = (float64_t) rand() / lemon;
    y = (float64_t) rand() / lemon;
    x = x3 * x3;
    y2 = y * y;
    x2 = x + y2;
    cnt = x2 > 1.0 ? cnt : cnt + 1;
    i = i + 1;
  }
  printf("pi is %g\n",
         (float64_t)cnt / iter*4.0);
  return 0;
```

Fig. 4: Computing Pi using Monte Carlo method (left – original C code, right – decompiled C code). The debugging information in the DWARF format was used for decompilation.

4.3 SOURCE CODE AND DEBUGGING INFORMATION AVAILABLE

This is the ideal scenario of the executable-code analysis and it is also to most common one (e.g. software development, precompiled open-source applications). All the previously mentioned features are available as well as the source-level debugging. All the common debugging features are supported (e.g. highlighting position within the source code, stack backtrace, inspecting variable values). Although it is not obvious, the decompiler can be used too. It can be utilized for compiler testing via comparison of the original source code and the code decompiled from the compiler-generated executables.

Finally, the decompilation results can be further enhanced via information gathered during simulation. This can be done in all scenarios, i.e. with or without source-code or debugging information. The description of this approach is given in the following section.

5 HYBRID ANALYSIS

In present, the static and dynamic analyses are isolated from each other; each analysis can be used independently of the other one, e.g. we can use decompilation without running simulation and vice versa. However, the outputs of each of them can be enhanced by outputs obtained in the other ones. This concept is called *hybrid analysis*. Currently, we can find two fundamental techniques of hybrid analysis:

- Exploitation of the run-time information (generated during simulation) within decompilation. This can be applied in all three scenarios described in Section 4.
- Decompilation results (i.e. a HLL C code) can be used to support the source-code debugging in scenarios where the source code or debugging information is unavailable (i.e. scenarios described in Sections 4.1 and 4.2).

5.1 EXPLOITATION OF RUN-TIME INFORMATION IN DECOMPILATION

It is possible to gather a considerable amount of valuable information during the application's run-time, e.g. a list of called functions, reachable instructions, values read and written into variables. On the other hand, it will be complicated or even impossible to get them during static analysis (e.g. during decompilation).

The mentioned run-time information can be gathered by the simulator. Such mode of execution is often referred to as a *trace mode*. The simulator stores all demanded information in an internal database that aggregates the results (e.g. removal of duplicate values) and the database's content is passed to the decompiler at the end of simulation. One of the open problems is simulation of applications that use dynamically linked code because the simulator must handle such situations (e.g. syscalls, invocation of standard-library functions). This task is related to *code emulation*.

Afterwards, all of these values can be used during the decompilation. For example, the list of called functions is usable in the function-detection algorithm (e.g. revelation of function's called by pointers), the list of positions can be employed for extracting targets of the indirect-branches (e.g. branch to the address stored in some register), variable's content during execution is important for the value-range analysis.

5.2 SOURCE-CODE DEBUGGING USING DECOMPILATION

The second proposed technique can be done in two ways. (1) In the easier one, the debugging information is available and the decompiler has to generate the target HLL code that corresponds to given debugging information. The HLL code must satisfy the integrity, e.g. mapping of machine-code instructions to HLL statements, mapping of variables to memory addresses, storage of local variables on stack in a given order.

(2) The other case (i.e. both source-code and debugging information are unavailable) is a more challenging task because both information have to be recreated during decompilation. Recreation of the source code can be done in a classical way described in the previous sections. Recreation of the debugging information related to the decompiled program can be tricky because the existing debugging standards are complex and the size of debugging information is usually several times larger than the application's code. Furthermore, the debugging standard can be undocumented and proprietary, such as Microsoft PDB.

The easiest solution is to generate a HLL code and re-compile it with debugging support enabled (e.g. "`-g`" in GCC) using an appropriate compiler. Output of this process is a new executable with debugging information that corresponds to the decompiled HLL code. However this may be an unwanted behavior because it implies debugging of a different application. The other possible solution is to recreate only a simplified version of debugging information by the decompiler itself (e.g. only mapping of machine code to HLL statements).

The debugger must be prepared for this situation and allow a lightweight-debugging mode with only a limited functionality. The DWARF standard suits this purpose better than PDB. Each type of debugging information (e.g. position mapping, variable information) is stored in a separated file section and its presence is optional. Furthermore, several existing libraries can be used for DWARF generation (e.g. libDWARF).

6 CONCLUSION

This paper was aimed on the retargetable analysis of executables. Several methods of static and dynamic analysis were presented, e.g. simulation, debugging, and decompilation. These methods are implemented within the Lissom project as the automatically generated toolchain. These tools are created based on the ISAC ADL models of the target architectures.

Using these tools, it is possible to inspect behavior of our own applications as well as the third-party applications, which may be harmful (e.g. malware). The presented approach has been tested on the MIPS architecture, while other architectures such as ARM or Intel x86 are supported too.

We close the paper by proposing an idea for the future research. We would like to interconnect the static and dynamic analysis within the hybrid analysis. Two possible ways of its usage in the given scenarios have been proposed; their implementation and evaluation is the primary target of the future research.

ACKNOWLEDGMENTS

This work was supported by the project TAČR TA01010667 System for Support of Platform Independent Malware Analysis in Executable Files, BUT grant FEKT/FIT-J-13-2000 Validation of Executable Code for Industrial Automation Devices using Decompilation, BUT FIT grant FIT-S-11-2, by the project CEZ MSM0021630528 Security-Oriented Research in Information Technology, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

7 REFERENCES

[1] BARBE, P. *The PILER system of computer program translation,* Technical report, Probe Consultants Inc., 1974.

[2] CIFUENTES, C. *Reverse compilation techniques*, PhD thesis, School of Computing Science, Queensland University of Technology, Brisbane, AU-QLD, 1994.

[3] 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), Kingston, Ontario, CA, IEEE CS, 2012, pp. 51-60, ISBN 978-0-7695-4891-3.

[4] ĎURFINA, L., KŘOUSTEK, J., ZEMEK, P., KOLÁŘ, D., HRUŠKA, T., MASAŘÍK, K., MEDUNA, A. *Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis*, In: International Journal of Security and Its Applications, Vol. 5, No. 4, 2011, Daejeon, KR, pp. 91-106, ISSN 1738-9976.

[5] KÄSTNER D., WILHELM S. *Generic control flow reconstruction from assembly code*, In Proceedings of the joint conference on Languages, compilers and tools for embedded systems: Software and compilers for embedded systems (LCTES/SCOPES '02), ACM, New York, NY, USA, pp. 46-55. 2002. URL http://www.absint.com

[6] KINDER, J., VEITH, H. *Jakstab: A static analysis platform for binaries*, In Computer Aided Verification, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, vol. 5123, pp. 423–427, 2008.

[7] KŘOUSTEK, J., MATULA, P., KONČICKÝ, J., KOLÁŘ, D. *Accurate Retargetable Decompilation Using Additional Debugging Information*, In: Proceedings of the Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'12), Rome, IT, IARIA, pp. 79-84, ISBN 978 1 61208-209-7, 2012.

[8] KŘOUSTEK, J., PŘIKRYL, Z., KOLÁŘ, D., HRUŠKA, T. *Retargetable Multi-level Debugging in HW/SW Codesign*, In: The 23rd International Conference on Microelectronics (ICM 2011), Hammamet, TN, IEEE, pp. 6, ISBN 978-1-4577-2209-7, 2011.

[9] LATTNER C. *LLVM: An Infrastructure for Multi-Stage Optimization*, Master's Thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. URL http://llvm.org/

[10] MASAŘÍK, K. *System for Hardware-Software Co-Design*, FIT BUT, ISBN 978-80-214-3863-7, Brno, CZ, 2008, URL http://www.fit.vutbr.cz/research/groups/lissom/

[11] MIPS Technologies Inc., *MIPS32 Architecture for Programmers Volume II-A: The MIPS32 Instruction Set*, 2010.

[12] PŘIKRYL, Z. *Advanced Methods of Microprocessor Simulation*, PhD thesis, Brno University of Technology, Faculty of Information Technology, Brno, CZ, p. 103, 2011.

[13] PŘIKRYL, Z., KŘOUSTEK, J., HRUŠKA, T., KOLÁŘ, D. *Fast Translated Simulation of ASIPs*, In: OpenAccess Series in Informatics (OASIcs), Vol. 16, No. 1, Wadern, DE, pp. 93-100, ISSN 2190-6807, 2011.

[14] RAMOS D. A., ENGLER, D. R. *Practical, low-effort equivalence verification of real code*, In *Proceedings of the 23rd international conference on Computer aided verification* (CAV'11), Springer-Verlag, Berlin, Heidelberg, pp. 669-685, 2011. URL http://www.coverity.com/

[15] ROSENBERG, B. J. *How Debuggers Work – Algorithms, Data Structures, and Architecture*, Wiley Computer Publishing, 1996.

[16] VAN EMMERIK, M. *Static Single Assignment for Decompilation*, PhD thesis, School of ITEE, University of Queensland, Brisbane, AU-QLD, 2007.