Jakstab: A Static Analysis Platform for Binaries ⋆
(Tool Paper)

Johannes Kinder[1,2] and Helmut Veith[1,2]

[1] Technische Universität Darmstadt, 64289 Darmstadt, Germany
[2] Technische Universität München, 85748 Garching bei München, Germany

Abstract. For processing compiled code, model checkers require accurate model extraction from binaries. We present our fully configurable binary analysis platform JAKSTAB, which resolves indirect branches by multiple rounds of disassembly interleaved with dataflow analysis. We demonstrate that this iterative disassembling strategy achieves better results than the state-of-the-art tool IDA Pro.

Introduction. While most of today's model checkers operate on source code, there are various settings where we need to verify binary code. First, when source code is not available, e.g., when a software manufacturer wants to verify the conformance of third party modules, such as drivers or plugins, to the API specification. Second, to be able to detect errors introduced in the compiling process [1], which is of particular importance in the field of embedded systems, where compilers can be unreliable. Third, binary level analysis results can supplement execution traces collected by testing and vice versa, as demonstrated by the SYNERGY algorithm [2]. And finally, our original motivation for this research stems from using model checking to detect malicious code inside executables [3].

Extracting a control flow graph (CFG) from an executable is not simply a matter of implementing a language front-end for assembly. Compiled code lacks many comfortable properties of structured high level languages and poses several challenges for analysis tools. Function pointers are only seldom handled by source-level verification tools, but on assembly level, calls and jumps to pointers are too abundant to be ignored. The treatment of function pointers requires dataflow analysis on an incomplete CFG. Thus, the traditional sequence, in which an analyzer builds the CFG first and only then performs dataflow analysis, has to be replaced by an iterative process. Another challenge is the loss of structure in compiled code. For accurate analysis results, procedures, along with their calling conventions, need to be explicitly detected. Compiler optimizations and, worse, obfuscation techniques can further mangle the control flow structure of an executable and impede correct disassembly and control flow extraction [4].

Existing disassemblers can be divided into two categories [4]: Linear sweep disassemblers, such as GNU objdump, simply sequentially translate machine code into assembly instructions. Recursive traversal disassemblers, such as IDA Pro, follow direct branches and decode the program by depth first search. We extend this classification by

⋆ Supported by DFG grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1) and the European Commission under Contract IST-2002-507932 ECRYPT.

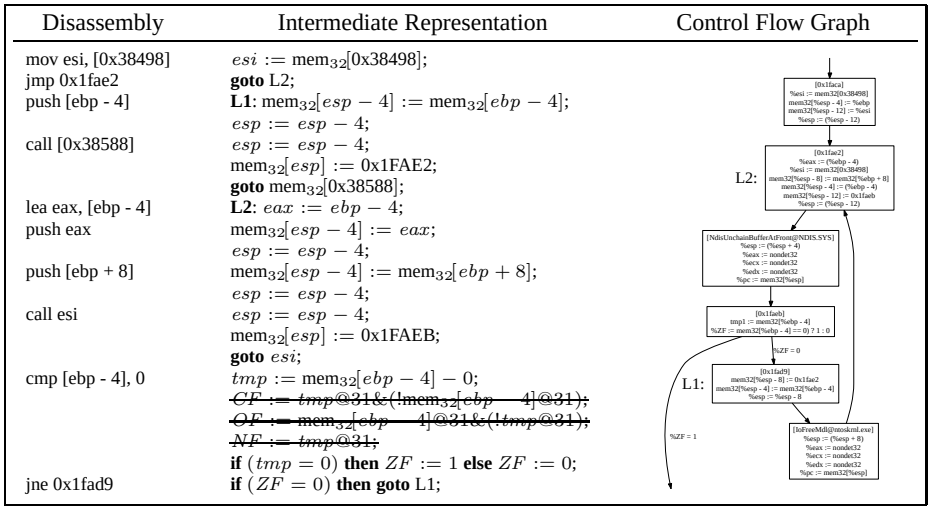

Fig. 1. Part of procedure 0x1FACA in fwdrv.sys. The second call is not resolved by IDA Pro.

defining an *iterative disassembler* as one that interleaves multiple disassembly rounds with dataflow analysis to achieve accurate and complete CFG extraction.

Our tool JAKSTAB[3] (**Ja**va tool**k**it for **st**atic **a**nalysis of **b**inaries) serves as a flexible front end to make executables accessible to static analysis and model checking. To this end, JAKSTAB contains an iterative disassembler and a library of semantic descriptions that translates assembly instructions to an RTL-style intermediate representation. Disassembler and semantic descriptions are fully configurable to support multiple target platforms. Using the intermediate representation, JAKSTAB iteratively creates the CFG, calculating and resolving indirect branch targets using results from dataflow analysis. JAKSTAB is implemented in Java and can be either used as a library or via its command line interface, which outputs plain disassembly or the intermediate representation as a CFG in graphviz-format. The intermediate representation, consisting of assignments, if, and goto statements, is independent of the target hardware and provides a natural interface to model checkers and program analysis tools.

Today's de facto industry standard for disassembly is IDA Pro. Its heuristic matches common prologue bytes to identify procedures and assumes that every call returns to its original site, regardless of the call target, which can lead to erroneous fall-through edges. Furthermore, the CFG is usually incomplete, since IDA Pro has only a very basic ability to resolve indirect branch instructions (function pointers): It propagates constants just within a basic block, and decorates calls to such constants with comments containing the actual target. While this is enough to aid human engineers, it is insufficient for automated analysis. Figure 1 shows an exemplary piece of assembly code from a Windows driver executable (fwdrv.sys from *Sunbelt Personal Firewall*), where IDA Pro (v4.7) fails to identify an indirect call to an imported function, whose address

[3] Project page online at `http://www.jakstab.org`

is stored at a memory location pointed to by the register *esi*. Finally, even though IDA Pro offers an (unsupported) SDK for plugin development, it is closed source software and thus cannot be easily integrated with an analysis tool.

To the best of our knowledge, the most successful approach to static analysis of executables currently is the CodeSurfer/x86 project [5]. CodeSurfer/x86 uses IDA Pro to access binaries, and combines two program analysis algorithms, value set analysis (VSA) and aggregate structure identification (ASI). In recent work, they combined VSA with a property automaton that encodes certain usage rules for the Windows driver API [6]. Generally, they assume a standard compilation model for binaries, which guarantees correct disassembly by IDA Pro. They acknowledge that IDA Pro's output can be incomplete and do connect missing edges from indirect calls, yet they lack a complete loop to disassemble previously unprocessed branch targets.

Closely related to executable analysis is the idea of building a *decompiler*, which transforms an executable back to source code [7, 8]. Chang et al. describe an architecture of communicating decompilers at different language levels [9]. Their implementation propagates static analysis facts through all language levels one instruction at a time, instead of strictly separating decompilation stages by language level. The prototype targets assembly source files generated by a set of compilers, and thus requires access to source code. We believe that JAKSTAB would fit nicely into this tool-chain as a provider of well-formed CFGs from generic executables.

Control Flow Reconstruction. In most assembly languages, instructions can affect multiple registers and status flags. The x86 architecture, which we first focused on, features an especially rich instruction set where instructions often represent non-trivial operation sequences. To fully capture instruction semantics and enable easy extensibility, JAKSTAB is designed to read Semantic Specification Language (SSL) files supplied with the Boomerang decompiler, which are available for several architectures including x86, PowerPC, 68K, and SPARC [10, 8]. Figure 1 shows the intermediate representation JAKSTAB produces from the assembly snippet using SSL definitions for the x86 architecture. Mapping every assembly instruction to its semantic specification creates a program representation with obvious pieces of dead code. In particular, most of the status flags are not used but simply overwritten by later instructions. To reduce the program size, our tool executes a live variable analysis and afterward removes any dead code. In our experiments, usually about 30% of the statements are identified as dead code and removed from the control flow graph. In the example in Figure 1, three flag updates are removed (crossed out text), and only one relevant update remains.

JAKSTAB recreates the control flow graph in an iterative process. Starting from the entry point of the executable, it propagates and folds constants through registers and memory cells to resolve indirect branch targets. JAKSTAB supports indirect memory access, which is common for local variables stored on the stack. Whenever Jakstab cannot resolve the address of an indirect write, it currently assumes that every memory cell can become undefined. Calls to shared libraries, which, in the Windows PE-format, appear as indirect calls to memory locations, are handled by creating stub procedures in the control flow graph. Constant propagation and folding is performed on all parts of the CFG already known, which allows JAKSTAB, in contrast to IDA Pro, to successfully recover the CFG of the example in Figure 1. Note that the results of constant propaga-

	cmd.exe		dnsrslvr.dll		faultrep.dll		ftp.exe		nmnt.sys		rcp.exe		svchost.exe	
IDA Pro	74%	9.4s	81%	36.2s	73%	5.4s	88%	2.4s	74%	3.1s	42%	1.4s	56%	1.5s
JAKSTAB	91%	32.4s	92%	3.2s	98%	9.0s	94%	2.7s	96%	4.5s	100%	1.1s	88%	1.0s

Fig. 2. Success rates and processing times for resolving indirect branches in executables.

tion can theoretically be incorrect if incoming edges to *existing* nodes are discovered in later iterations. In such cases, the CFG reconstruction process has to be restarted.

Any target location that has been successfully resolved in one iteration is scheduled for disassembly in the next one. Newly detected procedures are inlined to ensure correct interprocedural results in the next round of constant propagation. Figure 1 shows the CFG extracted from the example code, including stubs for imported library functions. The stubs non-deterministically assign those registers which might be overwritten by library functions (*eax*, *ecx*, *edx* according to the Intel application binary interface).

We compared JAKSTAB's and IDA Pro's capabilities in resolving indirect branches on Microsoft Windows system binaries. The results we present in Fig. 2 clearly show that JAKSTAB is able to provide significantly more accurate CFGs than IDA Pro at similar, and in some cases even faster, execution speeds.

Applications and Future Work. Our goal is to use JAKSTAB as a versatile platform for different verification tasks on binary level. Currently, we are building a bounded model checker on top of the existing framework to allow better resolution of indirect jumps and the extraction of all targets from jump tables. Besides the internal use of the bounded model checker for improving the CFG, we will investigate what kind of specifications can be verified on binary level, with particular focus on API usage specifications.

JAKSTAB, unlike IDA Pro, does not assume a standard compilation model. Therefore it is well suited to process code protected against disassembly, in particular malicious code. Anti-disassembly patterns that obscure the control flow of a program will thwart traditional recursive traversal disassemblers [4]. For example, return instructions are commonly misused as generic jumps by pushing the desired target address on the stack immediately beforehand. Since JAKSTAB supports local constant propagation through the stack, it can retarget disassembly correctly in these cases and is able to recover the real control flow. A CFG extracted from such a potentially malicious program can then be used as input to a semantic malware detector [3].

References

1. Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: WYSINWYX: What You See Is Not What You eXecute. In: VSTTE, Zurich, Switzerland (2005)
2. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: a new algorithm for property checking. In: SIGSOFT FSE'06, ACM (2006) 117–127
3. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: DIMVA'05. Volume 3548 of LNCS., Springer (2005) 174–187
4. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS'03, ACM (2003) 290–299

5. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: CC'04. Volume 2985 of LNCS., Springer (2004) 5–23
6. Balakrishnan, G., Reps, T.: Analyzing stripped device-driver executables. In: TACAS'08. LNCS, Springer (2008) 124–140
7. Cifuentes, C.: Reverse Compilation Techniques. PhD thesis, Queensland University of Technology (1994)
8. van Emmerik, M., Waddington, T.: Using a decompiler for real-world source recovery. In: WCRE 2004, IEEE Computer Society (2004) 27–36
9. Chang, B., Harren, M., Necula, G.: Analysis of low-level code using cooperating decompilers. In: SAS. Volume 4134 of LNCS., Springer (2006) 318–335
10. Cifuentes, C., Sendall, S.: Specifying the semantics of machine instructions. In: International Workshop on Program Comprehension (IWPC'98), IEEE Computer Society (1998) 126–133