Interprocedural Control Flow Reconstruction

Andrea Flexeder, Bogdan Mihaila, Michael Petter and Helmut Seidl

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany,
{flexeder, mihaila, petter, seidl}@cs.tum.edu,

Abstract. In this paper we provide an interprocedural algorithm for reconstructing the control flow of assembly code in presence of indirect jumps, call instructions and returns. In case that the underlying assembly code is the output of a compiler, indirect jumps primarily originate from high-level switch statements. For these, our methods succeed in resolving indirect jumps with high accuracy. We show that by explicitly handling procedure calls, additional precision is gained at calls to procedures exiting the program as well as through the analysis of side-effects of procedures onto the local state of the caller. Our prototypical implementation applied to real-world examples shows that this approach yields reliable and meaningful results with decent efficiency.

Key words: static analysis, binary analysis, control flow reconstruction, reverse engineering

1 Introduction

In contrast to high-level languages as e.g. C at the assembler level the semantics of a program can be fully specified, i.e. the effect of every assembler instruction is formally given by the instruction manual of the processor vendor. Consequently, an analysis of executables can provide more reliable results than a source-code analysis [5]. Full information about the behaviour of an assembly program is required e.g. for *reverse engineering* [8], i.e. to obtain an understanding of the structure of an executable, or when *analysing safety-critical real-time applications*. For instance, determining tight bounds for the worst-case execution time of an application [12] or checking safety properties of micro-controllers [7] for flawless functionality, demand for analysing the compiler output. Sometimes only the executable is available or the compiler may even contain bugs, s.t. the executable provides the basis for a static analysis. However, in order to perform analyses on assembly code, the control flow has to be reconstructed first. Resolving the jump targets for *indirect calls* and *indirect jumps* requires an analysis of the values of registers as well as of memory locations. Many architectures have specific instructions for both local jumps and procedure calls. However, not all occurrences of call instructions semantically denote procedure calls in the sense of temporary transfer of control to a subroutine. For instance consider the call to procedure exit in figure 1, which never returns control back to the caller. An analysis that assumes that every function returns can be misled by a call to such a non-returning function, since the immediately following program point should not be influenced by such a call. Thus, it is essential to deal with procedure calls for reconstructing meaningful control flow graphs.

Moreover, switch-statements at the assembler level are often translated to indirect jumps [9], as e.g. demonstrated by example 1. These jumps are controlled by a *jump table* containing relative jump targets for each case statement. In our setting this table is located in the read-only data segment of the executable and thus its entries are never changed. The target address of such an indirect jump is computed via the following instruction sequence: First via a comparison instruction (cf. instruction `0x08`) the value range of the index register is restricted. For example 1 register $r0$ is restricted to $0 \leq r0 \leq 5$. Note that the unsigned comparison instruction cmplwi treats its operands as unsigned integers (cf. [23]), i.e. all the negative numbers are rejected because their two‘s complement representation is larger than that of any positive number. If the value of register $r0$ is not inside these bounds, then the default case is executed, which results in a call to the exit function (cf. instruction `0x14`). Otherwise the instructions starting at `0x18` will be considered. Here, register $r0$ serves as an index into the jump table. Before the indirect jump is performed (cf. instruction `0x30`) the address offset read from the jump table is added to the table base address (cf. instruction `0x2C`).

```
int i = read();
switch(i)
{
  case 1: i += 11;   break;
  case 2: i += 22;   break;
  case 3: f(i);      break;
  case 4: i += 44;   break;
  case 5: i += 55;   break;
 default: exit(1);   break;
}
```

The jump table is given by:

```
.ro_data
51c60: ff fa e3 d4 // 51c60 - 34
51c64: ff fa e3 dc // 51c60 - 3c
51c68: ff fa e3 e4 // 51c60 - 44
51c6C: ff fa e3 ec // 51c60 - 4c
51c70: ff fa e3 f4 // 51c60 - 54
```

```
// i = read();
 00:    call     0x70
 04:    mr       r0,r3
//  switch(i) {
 08:    cmplwi   cr7,r0,5
 0C:    jle      cr7,0x18
 10:    li       r3,1
 14:    call     0x80 <exit>
 18:    mulli    r2,r0,4
 1C:    lis      r9,5
 20:    addi     r10,r9,7264
 24:    add      r9,r2,r10
 28:    lwz      r11,0(r9)
 2C:    add      r11,r11,r10
 30:    jump     r11
//    case 1: i += 11; break;
 34:    addi     r0,r0,11
 38:    jump     0x64 <postswitch>
//    case 2: i += 22; break;
 3C:    addi     r0,r0,22
 40:    jump     0x64 <postswitch>
...
//<exit>:
80:  li      r10,99
84:  halt
```

Fig. 1: Switch statement in PPC assembler

In this paper we present our example programs and our implementation for the *PowerPC* architecture (PPC) [23] which is still broadly used in embedded systems, as e.g. automotive industry, aeronautics or robotics. However, our approach can be applied to arbitrary architectures as well.

Before elaborating our framework in detail we present related approaches in the area of control flow reconstruction.

Related Work. Several tools tackle the problem of reconstructing the control flow from executables. Using a simple linear-sweep disassembler, as e.g. gcc's *objdump*, is not sufficient for identifying the code sections of an executable [21]. Therefore modern control flow reconstruction additionally relies on extra information either through code patterns used by compilers or static program analysis.

The first category of tools using *compiler patterns* for control flow reconstruction are e.g. *exec2crl* by AbsInt [25,24], *dcc* by Cifuentes et al. [1,10] and IDAPro [2]. exec2crl is a tool which extracts the control flow from well-formed compiler-generated assembly code of time-critical embedded systems. There, special coding conventions must be adhered to which prevent the use of function pointers and dynamic data structures. Additionally precise knowledge about the compiler and the target architecture is given. Under these restrictions a complete and sound control flow reconstruction is possible. The drawback of such a compiler-pattern driven approach is that for every compiler and change in the code generation schemes the set of patterns has to be adjusted.

Cifuentes et al. [10] propose slicing and substitution of expressions for obtaining normal forms for indirect jumps and calls. This normal form is matched against their repository of compiler patterns to recover high-level data flow information from executables. They only use heuristics if local memory is used to compute the address expression for an indirect jump or call. These heuristics make their tool unsound.

Our experiments with IDAPro [2] lead us to assume that the latest version 5.5 also uses compiler patterns to resolve those indirect jumps that represent switch statements.

The second category relies on *static analyses*. These approaches are used by tools such as e.g. *CodeSurfer* by Reps et al. [20,5], *Jakstab* by Kinder and Veith [14,15] and the work of Myreen [18]. In [15] Kinder and Veith present an analysis framework based on partial control flow graphs to resolve indirect jumps. They present a generic worklist algorithm to dynamically extend the control flow of a program. In [15] they claim that their approach yields *"the most precise overapproximation of the control flow graph w.r.t. the precision of the provided abstract domain."* However, they rely on an intra-procedural framework only, inlining newly detected procedures. Recursive procedures may lead to assembly code which is not manageable by their framework.

CodeSurfer works upon the control flow reconstruction of IDAPro. Reps et al. are aware of the fact that IDAPro yields an unsafe as well as incomplete control flow graph in presence of indirect jumps and calls. Thus, they attempt to augment and correct the information provided by IDAPro [5]. Their practical tool CodeSurfer, however, is not available to us.

In the context of program proving, Myreen [18] has presented a semantics-based control flow reconstruction via a translation into tail-recursive functions.

In this paper we present an analysis that safely reconstructs an overapproximation of the control flow for compiler-generated assembler programs by carefully examining call and jump instructions. Here, we follow the approach of Kinder and Veith [15] for dealing with indirect jumps and extend it with a treatment of procedure calls. One particular problem we deal with are abort and exit functions, which do *not return* to the corresponding call site, but terminate the whole program whenever they are called.

The structure of the paper is as follows: In Section 2 we present the concrete semantics our control flow reconstruction analysis builds on. Then, in Section 3 we describe a general interprocedural framework to overapproximate the control flow and call graph of an executable. Additionally we present a concrete instantiation of this framework in order to resolve common indirect jump instructions. We present experimental results and discuss several practical issues when analysing real-world code in Section 4. And finally we conclude.

2 The Concrete Semantics

Here, we present an instrumented concrete semantics w.r.t. which the control flow graph is defined. Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of registers of the processor. The instructions of the program are stored at the set of addresses $\mathbf{N} \subseteq \mathbb{N}$. For every executable we assume that we are given a unique start address $\mathsf{start} \in \mathbf{N}$ where program execution starts. The mapping $I : \mathbf{N} \to \mathbf{Instr}$ provides the processor instruction for a given program address from $\mathbf{N}$. Depending on the architecture, the width of an instruction may vary. For the PPC architecture, however, all instructions have equal width 4. In this paper, we consider the set **Instr** of processor instructions consisting of:

- stm: assignment statements $\mathbf{x}_i := e$, i.e. the value of expression e is assigned to register $\mathbf{x}_i \in \mathbf{X}$, memory read instructions $\mathbf{x}_i := M[e]$, where the content of the memory location specified by e is assigned to register $\mathbf{x}_i$ and memory write instructions $M[e_1] := e_2$, where the value of e_2 is assigned to the memory location specified by e_1;
- call $\mathbf{x}_i$: procedure calls, where the value of register $\mathbf{x}_i$ denotes the start address of a procedure;
- jump e $\mathbf{x}_i$: jump instructions, which transfer control to the address specified by $\mathbf{x}_i$ iff e evaluates to 0;
- return: the return-instruction transfers control back to the caller and
- halt: the program exit instruction which terminates execution of the whole program and transfers control back to the operating system.

Here, e, e_1, e_2 denote expressions as provided by the syntax of assembler instructions.

A procedure call call $\mathbf{x}_i$ transfers control to the callee whose address is given by the value of $\mathbf{x}_i$. We consider every address which is jumped to by a call instruction as the start address of a procedure. The address of the instruction directly following the procedure call is saved in a dedicated register, the *link* register of the processor. For instance, instruction `0x00` from figure 1 sets the link register to address `0x04`. The instruction return is nothing but an indirect jump to the address currently stored in the link register. For our control flow reconstruction, we only consider programs where return-statements transfer control back to the caller. This means that it is up to the callee to save the content of the link register (if necessary) and to restore it before executing the return. We leave it for supplementary analyses to verify that the link register is handled correctly.

For the sake of our analysis, we combine the comparison instruction and the succeeding branch instruction to the guarded jump instruction jump e $\mathbf{x}_i$. In concrete machine architectures, these instructions need not follow each other directly (see Section 4).

In the concrete semantics, we consider states σ assigning values to registers $\mathbf{X}$ and to memory locations from some address space $\mathbf{N}'$ which is disjoint from $\mathbf{N}$. Let V denote the set of all possible values. The set of all such states then is given by $\Sigma = (\mathbf{X} \cup \mathbf{N}') \to V$. Additionally, the instrumented operational semantics maintains a pair (c, f) where c is the address of the last call and f is the start address of the current procedure, with $c, f \in \mathbf{N}$. Processor instructions from $\mathbf{Instr}$ modify the current state $\sigma \in \Sigma$. The semantics of a single processor instruction s on a given program state σ is defined via the semantic function $[\![s]\!] : \Sigma \to \Sigma$. Besides modifying the state, it transfers control to another instruction (if it is an assignment or a jump), to another procedure (if it is a call) or to the environment (if it is a halt instruction). The transfer of control is provided by the partial function $\mathsf{next}_I : (\mathbf{N} \times \Sigma) \to \mathbf{N}$ which computes for every program point u with state σ, the next program point according to the semantics of the processor instruction $I(u)$:

$$\mathsf{next}_I(u, \sigma) = \begin{cases} \sigma(\mathbf{x}_i) \text{ with } [\![e]\!]\sigma = 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ u + 4 \text{ with } [\![e]\!]\sigma \neq 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ u + 4 & \text{otherwise} \end{cases}$$

Here, the function $[\![e]\!]\sigma$ evaluates an expression e and returns a value which is interpreted as an integer. In case of a jump-instruction the successor node is either the immediately following program point, if condition e does not evaluate to 0, or the value of the jump target register $\mathbf{x}_i$, otherwise. For procedure calls, the successor node is the immediately following program point (given that the called procedure returns).

Due to the presence of procedures, the small-step operational semantics is based on the two transition relations $\vdash_S$ and $\vdash_R$ denoting one step of intra-procedural and inter-procedural execution, respectively. These relations are defined by:

$$\begin{array}{ll} (u, \sigma, (c, f)) \vdash_S (\mathsf{next}_I(u, \sigma), \sigma', (c, f)) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge f' = \sigma(\mathbf{x}_i) \wedge \\ & (f', \sigma, (u, f')) \vdash_S^* (r, \sigma', (u, f')), \\ & I(r) = \mathsf{return} \\ (u, \sigma, (c, f)) \vdash_S (\mathsf{next}_I(u, \sigma), [\![I(u)]\!]\sigma, (c, f)) & \text{if } I(u) \text{ not a call} \\ (u, \sigma, (c, f)) \vdash_R (f', \sigma, (u, f')) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge f' = \sigma(\mathbf{x}_i) \\ (u, \sigma, (c, f)) \vdash_R (u', \sigma', (c, f)) & \text{if } (u, \sigma, (c, f)) \vdash_S (u', \sigma', (c, f)) \end{array}$$

An initial program state is given by $(\mathsf{start}, \sigma, (\mathsf{start}, \mathsf{start}))$ for suitable $\sigma \in \Sigma$.

Given this operational semantics, an approximation of the control flow of the program is a pair $(N, \mathsf{next}_I^{\sharp\sharp})$ where $N \subseteq \mathbf{N}$ and $\mathsf{next}_I^{\sharp\sharp} : \mathbf{N} \to 2^{\mathbf{N}}$ is a mapping such that for every initial configuration $\mathsf{conf} = (\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start}))$ the following holds:

- If $\mathsf{conf} \vdash_R^* (u, \sigma, (c, f))$ then $u \in N$;
- If $\mathsf{conf} \vdash_R^* (u, \sigma, (c, f)) \vdash_S (u', \sigma', (c, f))$ then $u' \in \mathsf{next}_I^{\sharp\sharp}(u)$.

Let $\mathsf{calls} \subseteq \mathbf{N}$ denote the subset of program points u where $I(u)$ is a call instruction. Then an approximation of the call graph of the program is a pair $(F, \mathsf{fun}_I^{\sharp\sharp})$ where $F \subseteq \mathbf{N}$ and $\mathsf{fun}_I^{\sharp\sharp} : \mathsf{calls} \to 2^F$ is a mapping such that for every initial configuration $\mathsf{conf} = (\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start}))$, $\mathsf{conf} \vdash_R^* (u, \sigma, (c, f))$ for some $I(u) = \mathsf{call}\, \mathbf{x}_i$, implies that $\sigma(\mathbf{x}_i) \subseteq \mathsf{fun}_I^{\sharp\sharp}(u)$.

3 Interprocedural Control Flow Reconstruction

Our goal is to construct sufficiently small pairs $(N, \mathsf{next}_I^{\sharp\sharp})$ and $(F, \mathsf{fun}_I^{\sharp\sharp})$. For that, we must determine tight approximations to the values of registers $\mathbf{x}_i$ occurring in indirect jump instructions $\mathsf{jump}\ e\ \mathbf{x}_i$ and indirect call instructions $\mathsf{call}\ \mathbf{x}_i$. In the following, we abstract from the concrete contents of the main memory and concentrate on the values of registers only. In order to be as precise as possible with the values of registers, we directly use the powerset domain 2^V ordered by subset inclusion as our abstract domain. Thus, an abstract state is described by a mapping $\sigma^\sharp$ from registers to the abstract domain $\mathbf{X} \to 2^V$. Only when sets of values grow, we may insert a widening to an enclosing interval [19]. However, the interval domain also requires a widening operation [11] to ensure termination of the fixpoint iteration. Typically, loops and recursive functions may lead to infinitely ascending chains. In our analysis framework, we therefore insert widening operators at back-edges and at procedure entries.

The general framework relies on an arbitrary complete lattice $\Sigma^\sharp$ of abstract states together with a concretisation $\gamma : \Sigma^\sharp \to 2^\Sigma$ where $\gamma(\sigma^\sharp)$ returns the set of concrete states described by $\sigma^\sharp$. Additionally, we require for every instruction s the corresponding abstract transformer $[\![s]\!]^\sharp : \Sigma^\sharp \to \Sigma^\sharp$ which safely approximates the concrete semantics of s, i.e., which satisfies:

$$[\![s]\!]\sigma \in \gamma([\![s]\!]^\sharp \sigma^\sharp) \ \text{ whenever } \ \sigma \in \gamma(\sigma^\sharp)$$

Given the abstract lattice $\Sigma^\sharp$ and the concretisation γ, we define the abstract next function $\mathsf{next}_I^\sharp : \mathbf{N} \times \Sigma^\sharp \to 2^{\mathbf{N}}$ by:

$$\mathsf{next}_I^\sharp(u, \sigma^\sharp) = \begin{cases} \gamma(\sigma^\sharp(\mathbf{x}_i)) \cap \mathbf{N} \text{ with } ([\![e]\!]^\sharp \sigma^\sharp) = \{0\} & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} \text{ with } ([\![e]\!]^\sharp \sigma) \not\ni 0 & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} \cup \gamma(\sigma^\sharp(\mathbf{x}_i)) \cap \mathbf{N} \text{ otherwise} & \text{if } I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \\ \{u+4\} & \text{otherwise} \end{cases}$$

Here, the abstract evaluation function $[\![e]\!]^\sharp$ takes an expression and returns a set of possible values of e.

For guarded jump instructions the set of successor program points is specified by the value of register $\mathbf{x}_i$ in the current register valuation if condition e is fulfilled, by the immediately following program point if e is not fulfilled or both sets otherwise. For all other processor instructions $\mathsf{next}_I^\sharp$ yields the immediately following program point.

Our analysis determines for each possible procedure entry node f a pair $\mu(f) = (\sigma^\sharp, C)$ where $\sigma^\sharp \in \Sigma^\sharp$ describes all possible concrete states at return points reachable from f on the same level (i.e., through $\vdash_S$), and $C \subseteq \mathbf{N}$ is the superset of all possible call sites for f. Additionally, the analysis determines for every program point u a pair $\eta(u) = (\sigma^\sharp, R)$ where $\sigma^\sharp$ describes the set of all states attained at u when reaching u from an initial state, and $R \subseteq \mathbf{N} \times \mathbf{N}$ is a set of pairs (c, f) of call sites c for procedure entry points f such that the current program point is reachable from f on the same level (i.e., w.r.t. $\vdash_S$).

Assume that $\eta(u) = (\sigma^\sharp, R)$. Then we refer to the i-th component of the pair $\eta(u)$ via $\eta_i(u)$. The components of the pair $\mu(f)$ will be accessed analogously.

The values $\mu(f)$ and $\eta(u)$ can be characterised as a solution of the following constraint system:

$$
\begin{array}{lll}
(1)\ \mu(f) & \sqsupseteq ((c,f) \in \eta_2(u)); (\eta_1(u), \{c\}) & \text{if } I(u) = \mathsf{return} \\
(2)\ \eta(\mathsf{start}) & \sqsupseteq (\top, \{(\mathsf{start}, \mathsf{start})\}) & \\
(3)\ \eta(v) & \sqsupseteq (f \in \gamma(\eta_1(u)(\mathbf{x}_i)) \wedge (u \in \mu_2(f))); & \\
 & \quad (H^\sharp(\eta_1(u), \mu_1(f)), \eta_2(u)) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \wedge v = u + 4 \\
(4)\ \eta(f) & \sqsupseteq (f \in \gamma(\eta_1(u)(\mathbf{x}_i))); (E^\sharp(\eta_1(u)), \{(u,f)\}) & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i \\
(5)\ \eta(v) & \sqsupseteq (v \in \mathsf{next}^\sharp_I(u, \eta_1(u))); ([\![s]\!]^\sharp(\eta_1(u)), \eta_2(u)) & \text{if } I(u) = s \in \mathsf{stm}
\end{array}
$$

Here, the operator ";" is defined by:

$$
(x \in A); B = \begin{cases} B & \text{if } x \in A \\ \bot & \text{otherwise} \end{cases}
$$

Constraint (1) describes the effect of a possibly called procedure f which may reach a return point. For constraint system η, initially at the start point start no information about possible variable valuations is known. Additionally we mark the start point as reachable by managing the relation $(\mathsf{start}, \mathsf{start})$, as constraint (2) specifies. Constraint (3) treats the case of a procedure call call $\mathbf{x}_i$. There, on the one hand the set of successor nodes is specified by the set of possible values of register $\mathbf{x}_i$, i.e., the set of entry points of the callees, and on the other hand, by the immediately following program point — given that any of the possibly called procedures returns. The value after the procedure call $u + 4$ consists of the set of call site - callee - relations valid before the call to procedure f together with the combination of the data flow value before the procedure call with the procedure summary $\mu_1(f)$. This combination is computed by the function $H^\sharp$. The function $E^\sharp$ computes the contribution of the abstract state of the current call site to the start point f of the callee. Additionally we relate the current call site u to the entry point of procedure f. This is defined by constraint (4). Constraint (5) treats all other forms of statements, which have no influence on the call site - callee relations. The successor node is computed by the abstract next function.

Note that for a procedure f which does not return, $\mu(f)$ yields $\bot$. Thus, in case of a call instruction at program point u the directly following program point $u + 4$ will not be reached.

A safe approximation of $E^\sharp$ and $H^\sharp$ independent of the abstraction $\Sigma^\sharp$ is:

$$
\begin{array}{ll}
E^\sharp(\sigma^\sharp) & = \sigma^\sharp \\
H^\sharp(\sigma^\sharp_c, \sigma^\sharp) & = \sigma^\sharp
\end{array}
$$

Assume we are given a (not necessarily least) solution (μ, η) of the constraint system. Then we can extract both an approximate control flow $(N, \mathsf{next}^{\sharp\sharp}_I)$ and an approximate call graph $(F, \mathsf{fun}^{\sharp\sharp}_I)$ by:

$$
\begin{array}{lll}
N & = \{u \mid \eta(u) \neq (\bot, \emptyset)\} & \\
\mathsf{next}^{\sharp\sharp}_I(u) & = \mathsf{next}^\sharp_I(u, \eta_1(u)) & \\
F & = \bigcup \{f \mid \eta_2(u) = \{\_, f\}\} & \\
\mathsf{fun}^{\sharp\sharp}_I(u) & = \gamma(\eta_1(u))(\mathbf{x}_i) \cap \mathbf{N} & \text{if } I(u) = \mathsf{call}\ \mathbf{x}_i
\end{array}
$$

F captures all possible procedure entry points of both functions that may return to the caller and functions that do definitely not return.

The following theorem relates the least solution of our constraint system with the (instrumented) operational semantics of the program as specified through the relations $\vdash_S$ and $\vdash_R$.

Theorem 1. (Correctness) *Let (μ, η) denote the least solution of the constraint system. Then the following holds:*

1. *Assume that $\eta(u) = (\sigma^\sharp, R)$ and $(\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start})) \vdash_R^* (u, \sigma, (c, f))$. Then $(c, f) \in R$ and $\sigma \in \gamma(\sigma^\sharp)$.*
2. *Assume that $\mu(f) = (\sigma^\sharp, C)$ and $(\mathsf{start}, \sigma_0, (\mathsf{start}, \mathsf{start})) \vdash_R^* (f, \sigma, (c, f)) \vdash_S^* (u, \sigma_u, (c, f))$ where $I(u) = \mathsf{return}$. Then $c \in C$ and $\sigma_u \in \gamma(\sigma^\sharp)$.*

The proof of theorem 1 is by induction on the length of the respective execution steps $\vdash_S$ and $\vdash_R$, respectively. As an immediate corollary, we obtain:

Corollary 1. *The pairs $(N, \mathsf{next}_I^{\sharp\sharp})$ and $(F, \mathsf{fun}_I^{\sharp\sharp})$ are approximations of the control flow and call graph of the input program.* $\square$

Instead of abstracting the state at a program point u only, we may also abstract the transformer along the path to program point u. The abstract domain $\Sigma^\sharp$ can be enhanced by additionally accumulating an abstraction of the state transformer from $\mathbb{T}$ corresponding to the current procedure. Thus, we consider the abstract domain $\Sigma^\natural = \Sigma^\sharp \times \mathbb{T}$. Accordingly we have to adjust the abstract semantic function $[\![s]\!]^\natural : \Sigma^\natural \to \Sigma^\natural$ to elements from $\Sigma^\natural$:

$$[\![s]\!]^\natural(\sigma^\sharp, \tau) = ([\![s]\!]^\sharp \sigma^\sharp, [\![s]\!]_{\mathbb{T}}^\natural \circ^\natural \tau)$$

where $\circ^\natural$ denotes the composition of transformers τ from $\mathbb{T}$ and $[\![s]\!]_{\mathbb{T}}^\natural : \mathbb{T}$ denotes the abstract semantic function on a processor instruction s, i.e. is a state transformer from $\mathbb{T}$.

This enhancement by abstracting the state transformers enables a more precise definition of the function $H^\sharp$ w.r.t. the domain $\Sigma^\natural$.

$$H^\sharp((\sigma_1^\sharp, \tau_1), (\sigma_2^\sharp, \tau_2)) = (\iota(\tau_2, \iota(\tau_1, \sigma_1^\sharp)), \tau_2 \circ^\natural \tau_1)$$

with $\iota : \mathbb{T} \to \Sigma^\sharp \to \Sigma^\sharp$. $\iota(\tau, \sigma^\sharp)$ transforms an abstract state $\sigma^\sharp$ by means of the state transformer $\tau \in \mathbb{T}$ which is interpreted in the context of the abstract domain $\Sigma^\sharp$. Additionally the function $E^\sharp$ is given by:

$$E^\sharp((\sigma^\sharp, \_)) = (\sigma^\sharp, \mathsf{Id}^\sharp)$$

with $\mathsf{Id}^\sharp$ the identity mapping.

One specific instance of this abstraction $\mathbb{T}$ records e.g. the set of registers which have definitely not been modified since procedure entry. For that, we choose $\Sigma^\natural = (\mathbf{X} \to 2^V) \times 2^{\mathbf{X}}$. Then $H^\sharp$ can be refined to:

$$\begin{aligned} H^\sharp((\sigma_c^\sharp, X), (\sigma^\sharp, X')) &= (\tilde{\sigma}^\sharp, X' \cap X) \qquad \text{where} \\ \tilde{\sigma}^\sharp(\mathbf{x}) &= \begin{cases} \sigma_c^\sharp(\mathbf{x}) & \text{if } \mathbf{x} \in X' \\ \sigma^\sharp(\mathbf{x}) & \text{otherwise} \end{cases} \end{aligned}$$

where for the instantiation of $H^\sharp$, $\circ^\natural$ is the intersection of both register sets. Combining the effect of a called procedure with the state of the call site results in a register valuation $\tilde{\sigma}^\sharp$ which takes its values from the register valuation before the call for all registers which are not modified by the called procedure and the values at procedure return for the remaining ones. Additionally, the set of definitely not modified registers for the caller after the call is given by the intersection of the respective sets of the caller before the call and the callee.

The value for the start point of a procedure is given by the register valuation $\sigma^\sharp$ at the call site for the procedure together with the set of all registers:

$$E^\sharp(\sigma^\sharp, X) = (\sigma^\sharp, \mathbf{X})$$

In this instance of domain $\mathbb{T}$, $\mathsf{Id}^\sharp$ is given by the set of all registers.

The constraint system as specified above, is not really tractable. In particular, the set of program locations is not known beforehand. In order to overcome this obstacle, we extend the approach of [15] and explore the reachable program locations as they are encountered during fixpoint computation. Besides indirect jumps, our extension also handles calls and returns.

In case of a return instruction at program point r, we rely on the fixpoint algorithm for updating the summaries $\mu(f)$ of procedure entries f from which r is (intra-procedurally) reachable, and let it re-consider the call sites of f if the summary $\mu(f)$ has changed. This results in the worklist-based fixpoint algorithm 1.

Fixpoint Algorithm 1

$W \leftarrow \{(\mathsf{start}, (\top, \{(\mathsf{start}, \mathsf{start})\}))\}$;
while $(W \neq \emptyset)$ **do**
  $(u, s) = \mathsf{extract}(W)$;
  if $(s \not\sqsubseteq \eta(u))$ **then**
    $\eta(u) \leftarrow \eta(u) \sqcup s$;
    $(\sigma^\sharp, R) = \eta(u)$;
    if $(I(u) = \mathsf{jump}\ e\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) = \top)$ **then**
      $\mathsf{abort}()$;
    else if $(I(u) = \mathsf{call}\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) = \top)$ **then**
      $W \leftarrow W \cup \{(u + 4, (\top, R))\}$;
    else if $(I(u) = \mathsf{call}\ \mathbf{x}_i \wedge \sigma^\sharp(\mathbf{x}_i) \neq \top)$ **then**
      $W \leftarrow W \cup \{(f, (E^\sharp(\sigma^\sharp), \{(u, f)\})) \mid f \in \gamma(\sigma^\sharp(\mathbf{x}_i))\}$;
    else if $(I(u) = \mathsf{return})$ **then**
      for all $((\_, f) \in R)$ **do**
        if $((\sigma^\sharp, \{c \mid (c, f) \in R\}) \not\sqsubseteq \mu(f))$ **then**
          $\mu(f) \leftarrow \mu(f) \sqcup (\sigma^\sharp, \{c \mid (c, f) \in R\})$;
          $(\sigma^\sharp_f, R_f) = \mu(f)$;
          $W \leftarrow W \cup \{(c + 4, (H^\sharp(\eta_1(c), \sigma^\sharp_f), \eta_2(c))) \mid (c, \_) \in R_f\}$;
    else
      $W \leftarrow W \cup \{(v, ([\![I(u)]\!]^\sharp \sigma^\sharp, R)) \mid v \in \mathsf{next}^\sharp_I(u, \sigma^\sharp)\}$;

Initially, we assume that $\eta(u)$ is (implicitly) initialised with the least possible value $(\bot, \emptyset)$ for all possible values of u. Likewise, we assume that μ assigns $(\bot, \emptyset)$ to all possible entry points of procedures.

Algorithm 1 maintains a worklist W consisting of all pairs (u, s) of program points together with a potential update s for the value $\eta(u)$. The algorithm terminates when all these updates have been processed. For processing one pair (u, s), the algorithm first checks whether s is already subsumed by the current value of $\eta(u)$. If this is not the case, s is added to $\eta(u)$, and this change is propagated to all consumers of the value $\eta(u)$. Here, a case distinction on the instruction at program point u is performed.

In case the target addresses of a call-instruction are not known (cf. line 9), we at least assume that the called function returns and overapproximate the return state with $\top$. Otherwise, we extend the worklist by pairs, consisting of all the targets f that may be called and their corresponding states (cf. line 12). In case of a return-instruction in procedure f we propagate the effect of f to all its call sites (cf. line 18). For all other kinds of program instructions the worklist is extended by pairs, consisting of all the successor nodes (computed via the abstract next function) and the corresponding state update (computed via the abstract semantic evaluation function).

With our current instantiation of $\Sigma^\sharp$ which only keeps track of the values of registers, we are only able to resolve *static* procedure calls. A more sophisticated instantiation, however, which additionally analyses the memory in greater detail, would also allow to compute a safe approximation of the control flow of a larger class of programs.

An assembly program can be either *stripped*, i.e. symbol table and debugging information is missing, or *unstripped*. The symbol table contains all the start addresses of the procedures F provided by the executable. In case we have a symbol table we start our analysis from all procedure start points. Furthermore, we can make the assumption that only those procedures may be called, which are listed in the symbol table in case of a call-instruction whose target addresses are unknown. In case of analysing a stripped executable, procedure start addresses are uncovered on the fly. Every executable is provided with a unique start address, specified in the header of the executable. Typically, the entry point of an executable is the start address of the .text section. If the target address of a call instruction call $\mathbf{x}_i$ is unknown, we must assume that an unknown procedure is called, which may call any other procedure in any state. Thus, a safe approximation of $E^\sharp$ and $H^\sharp$ is only given by:

$$\begin{aligned} E^\sharp(\sigma^\sharp) \quad &= \top \\ H^\sharp(\sigma^\sharp_c, \sigma^\sharp) &= \top \end{aligned}$$

The function abort (cf. line 8 of algorithm 1) indicates that the reconstruction of the control flow graph has failed. For unknown target addresses of jump-instructions (cf. line 7 of algorithm 1) we abort control flow reconstruction. Section 4 shows that an instantiation of our framework which tracks both the values of registers and memory locations is able to resolve all indirect jumps (resolvable by a static analysis) on all our benchmark programs. We fail in resolving some of the indirect calls since we do not track code addresses stored in the heap.

On regular termination, let (μ, η) be the variable valuations computed by algorithm 1, and let $F = \bigcup\{f \mid \eta_2(u) = \{(\_, f)\}\}$ and $N = \{u \in \mathbf{N} \mid \eta(u) \neq (\bot, \emptyset)\}$. Then the pair $(\mu|_F, \eta|_N)$ is a solution of our constraint system when restricted to procedure entries from F and program points from N. In particular, this means that the control flow graph $(N, \mathsf{next}_I^{\sharp\sharp})$ and the call graph $(F, \mathsf{fun}_I^{\sharp\sharp})$ constructed from (η, μ) are indeed approximations of the control flow and the call graph of the program.

Our experiments show that in case of switch-statements which are realised by jump table look-ups, we have to take memory into account. The jump table can either contain absolute addresses or address offsets, as e.g. is the case in our example in figure 1. Jump tables $T : \mathbf{N}'' \to V$ are located in the read-only memory $\mathbf{N}'' \subset \mathbf{N}'$ of an executable. Thus, in our instantiation of the framework we handle all those memory read accesses $\mathbf{x}_i := M[\mathbf{x}_j]$ to the read-only data section only. Then the abstract semantic function on memory access expressions is defined by:

$$[\![M[\mathbf{x}_j]]\!]^\sharp \sigma^\sharp = \begin{cases} \{T[c] \mid c \in \sigma^\sharp(\mathbf{x}_i)\} & \text{if } \sigma^\sharp(\mathbf{x}_i) \setminus \mathbf{N}'' = \emptyset \\ V & \text{otherwise} \end{cases}$$

In compiler-generated switch statements, typically no procedure calls are involved in the address computation for the jump target. Nevertheless, our experiments with real-world applications reveal that procedure calls may occur in-between this address computation, as figure 1 illustrates. The compiler omits a jump to the end of the switch statements, if an exit-procedure is called within the default branch of the switch-statement. Only a sufficiently precise treatment of procedure calls can avoid the loss of essential information for resolving the jump instruction at address `0x30` in figure 1.

4 Practical Issues and Experiments

Based on our theoretical approach, we implemented a prototypical control flow reconstruction tool to explore the quality of the resulting control flow graph and identify the next challenges by means of real-world programs. Our current implementation tracks the values of registers and memory locations but completely neglects the heap.

We conducted our experiments on a $2,2$ GHz quad-core machine equipped with physical memory of 16GB. All our benchmark programs have been compiled with GCC version $4.4.3$ using optimisation levels 0 and 2 *without* debug information for the *PowerPC* architecture. For the moment we only inspect fully statically linked and stripped executable programs. Hence our benchmark programs contain the whole *GNU C library* code. Our prototypical implementation (VoTUM [4]) consists in the following steps: First GCC's *objdump* is applied to the binaries to extract the assembler instructions. Then, we parse these assembler instructions and use them as the basis for our control flow reconstruction. The following two tables present the performance of our analyser on the benchmark programs.

Within these tables we specify: the binary file size **`Size`**; the number of procedure entries **`Procs`** (which is provided by the symbol table of the corresponding unstripped version of the binary) and in parentheses the number of procedures identified by our analyser; the number of assembler instructions **`Instr`**; the number of indirect jumps **`bctr`** and indirect calls **`bctrl`**; the number of *unresolved* indirect jumps **`ures`** and in parentheses the number of statically not resolvable indirect jumps due to runtime linkage;

Table 1: Benchmark suite for programs with optimisation level 0

Program	Size	Procs	Instr	bctr	ures	ureac	bctrl	res	ureac	bl	M(GB)	T(s)
openSSL	3.8MB	6708(375)	769511	163	0(4)	129	1352	20	1219	35709	4	203
thttpd	884kB	1197(464)	196493	77	0(5)	42	321	21	189	6092	1.2	67
switches	636kB	825(364)	138178	82	0(4)	42	302	20	184	3680	0.8	45
control	633kB	817(354)	139917	83	0(4)	49	302	16	184	3670	0.8	42
coreutils	3.9MB	5671(2371)	852322	431	0(26)	219	1648	101	1004	24159	1.3	527
gzip	0.7MB	1076(472)	166213	79	0(4)	44	310	20	188	4634	1.1	132

Table 2: Benchmark suite for programs with optimisation level 2

Program	Size	Procs	Instr	bctr	ures	ureac	bctrl	res	ureac	bl	M(GB)	T(s)
openSSL	2.9MB	6232(380)	613882	150	0(4)	116	1355	20	1217	34405	3	156
thttpd	852kB	1147(469)	189034	77	0(5)	42	320	17	190	5890	1	60
switches	625kB	826(358)	137833	77	0(4)	41	302	17	184	3673	0.8	44
control	629kB	817(354)	138589	81	0(4)	47	302	20	184	3670	0.8	40
coreutils	3.8MB	5372(2534)	830407	424	0(28)	202	1634	104	959	23504	1.3	459
gzip	0.7MB	1026(384)	162380	83	0(5)	44	309	20	190	4587	1	117

the number of *resolved* indirect calls **`res`**; **`ureac`** denotes the number of *unreachable* indirect jump and call instructions which the analyser did not reach when starting from the entry point of the stripped binary; the number of static call instructions **`bl`**; the memory consumption **`M(GB)`** in GB and the time consumption **`T(s)`** in seconds of our analyser.

For our benchmark suite on the one hand we concentrate on applications from embedded systems, as e.g. communication protocols **`openSSL`**, lightweight HTTP servers **`thttpd`** and a SCADE generated vehicle control program **`control`** from [3]. On the other hand we took a home-made example program **`switches`** with several characteristics of switches: nested switches, switches in loops, etc. **`coreutils`** consists of five selected programs (**`ls`**, **`basename`**, **`vdir`**, **`chmod`**, **`chgrp`**) taken from the *GNU Coreutils* package of Unix in order to demonstrate the applicability of our approach to ordinary desktop software and **`gzip`** to be comparable to other tools which refer to *SPECint*.

Some of our benchmark programs use lazy binding of procedure addresses via indirect jumps within the trampoline code to the so-called Procedure Linkage Table (**`PLT`**) [16]. The absolute address of such a dynamically loaded procedure is loaded from a constant memory location in the **`PLT`** section and then branched to via a **`bctr`**-instruction. If this location is not yet initialised, the trampoline branches to the runtime linker, which provides the dynamic address of the corresponding procedure. However, the address of this runtime linker is not present in the binary – it is only provided after loading the binary. Thus, *no static value* for the target of such a **`bctr`** instruction can be determined. Consequently, we list this kind of unresolvable **bctr** instructions in parentheses within our benchmark tables.

Summarising, our instantiation of the framework is able to provide tight bounds for all of the statically resolvable indirect jumps within the benchmark programs. We fail in resolving some of the indirect call instructions due to the fact that we have not modelled bit operations in our semantics yet and do not take the heap into account.

Position-independent Code. We examined switch-constructs within position-independent code (PIC), which is common in shared libraries [16]. Such code accesses all constant addresses through the global offset table (GOT), located in the read/write data section of the program. Consider the following example:

```
04:  call    0x08
08:  mflr    r30
0C:  lwz     r0,-24(r30)
10:  add     r30,r0,r30
....
2C:  lwz     r0, 24(r1)
30:  cmplwi  cr7,r0,5
34:  bgt     cr7,0x70<default>
38:  mulli   r9,r0,4
3C:  lwz     r0,-32764(r30)
40:  add     r9,r9,r0
44:  lwz     r9,0(r9)
48:  add     r9,r9,r0
4C:  jump    r9
```

After instruction `0`x10 register $r30$ contains the address of the GOT (cf. instructions `0`x04–0x10). Typically, in order to obtain the address of the GOT, *instruction pointer relative addressing* is used. This is realised via a call instruction to the immediately following location. The effect of this local jump is that the continuation address is saved in the link register. This continuation address serves as a fixed point in the code section and via a constant difference the GOT can be addressed, although its absolute address is not known until runtime.

After instruction `0`x38 register $r9$ holds the value of the switch index variable. The base address of the switch table is computed via a look-up in the GOT, as instruction `0`x3C illustrates. Finally, an access into the jump table (in the read-only data section) is performed at instruction `0`x44. Under the assumption that the location with offset 32764 to the GOT (cf. instruction `0`x3c) is definitely not overwritten, we can safely infer the base address of the jump table.

Control Flow Splitting. For our semantics we assumed that the `compare`- and `branch`-instructions are either directly following each other (cf. instructions `0x08,0x0C` in example 1) or the processor instructions in between the `compare` and the `branch`-instructions do not modify the register the compare is based on. This assumption, however, need not always be satisfied. In order to deal with this case we propose the technique of control flow splitting, as described in [22].

Function Pointers. At the assembler level, function pointers are realised via indirect calls.

Consider the following example code motivated by a Linux kernel driver, as for instance `linux-2.6.33/drivers/md/md.c`, where a bunch of initialisation functions is managed in a global array. Procedure `global_init` sequentially calls all the initialisation functions.

```c
const fptr inits[] =
    {init1,init2,init3};
void global_init() {
  int j = sizeof(inits);
  int i;
  for (i=0; i<j; i++)
    inits[i]();
}
```

```
//for (i=0; i<j; i++)
00:     li      r0,0
04:     stw     r0,8(r1)
08:     jump    0x30
//inits[i]();
0C:     lis     r9,10
10:     addi    r9,r9,6908
14:     mulli   r0,r0,4
18:     add     r9,r0,r9
1C:     lwz     r0,0(r9)
20:     call    r0
24:     lwz     r9,8(r1)
28:     addi    r0,r9,1
2C:     stw     r0,8(r1)
30:     cmpwi   cr7,r0,12
34:     blt     cr7,0x0C
```

Assuming that the global array `inits` is located in the read-only memory, our control flow reconstruction analysis allows to infer the targets for the call-instruction `0x20`. There are common compilers arranging all constant global data in the read-only memory. However, if this is not the case we either have to enhance our (theoretical) analysis framework with a memory analysis or rely on a may-analysis of modified memory locations. Let B denote such a set of possibly modified memory locations. Then, in our analysis framework we only have to adjust the abstract effect function for memory read accesses:

$$[\![M[\mathbf{x}_j]]\!]^\sharp \sigma^\sharp = \begin{cases} \{M[c] \mid c \in \sigma^\sharp(\mathbf{x}_i)\} & \text{if } (\sigma^\sharp(\mathbf{x}_i) \setminus \mathbf{N}') \cap B = \emptyset \\ V & \text{otherwise} \end{cases}$$

Our benchmark examples show that the number of indirect call-instructions (column **`bctrl`** in table 1) is significantly smaller than the number of static call-instructions (column **`bl`** in table 2). Our current implementation of the framework neither supports a precise handling of bit operations nor of the heap memory and thus fails in resolving some of the indirect calls.

Optimisation Levels. Our instantiation of the framework speaks about register valuations, only. Thus, the control flow reconstruction yields precise results as long as values are kept in registers only. This is the case for assembly code generated by compilers with a higher optimisation level. However, in case of unoptimised code or register pressure compilers store values on the stack. In order to analyse unoptimised assembly code, we extended our implementation of the framework by a stack analysis. Via the approach of inferring linear relations as presented in [17], we detect local and global memory locations. Since in such code the values of stack locations are temporarily cached in registers [13], also an analysis of relations between registers and memory locations is mandatory to precisely track the values of both registers and memory locations.

5 Conclusion

We have presented a framework for static analysis to jointly approximate the control flow and the call graph in presence of indirect jumps and calls. Such an approach is less restrictive than approaches relying on compiler patterns only. Furthermore, we discussed the challenges and possible solutions for code generated via different optimisation levels. In order to precisely reconstruct the control flow in presence of indirect calls, abstract domains are required which capture side-effects of procedures, and possibly also track code addresses which are stored in the heap [6].

For our prototypical implementation, we have assumed that the code to be analysed adheres to the coding conventions for calls to and returns from procedures. It remains for future work to extend these techniques to deal with code which deliberately violates these conventions. On the one hand, it remains to show that the executable to analyse adheres to our assumptions, such as e.g. a correct management of the return address. On the other hand, there are several areas for which code that does not adhere to the calling conventions offers interesting challenges, as e.g. self-modifying code, self-extracting executables or hand-made assembly code, as e.g. malicious code or optimised library code.

References

1. *DCC decompiler*. `http://www.itee.uq.edu.au/~cristina/dcc.html`.
2. *IDAPro disassembler*. `http://www.hex-rays.com/idapro/`.
3. Sicherheitsgarantien Unter REALzeitanforderungen. `http://www.sureal-projekt.org/`.
4. *VoTUM*. `http://www2.in.tum.de/votum`.
5. G. Balakrishnan. *WYSINWYX: What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin, Madison,WI,USA, August 2007.
6. G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis, 13th International Symposium, SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.
7. J. Brauer and A. King. Automatic abstraction for intervals using boolean formulae. In *Static Analysis Symposium (SAS 2010), Perpignan, France*, Lecture Notes in Computer Science. Springer, 2010.
8. C. Cifuentes. *Reverse Compilation Techniques*. Ph.D. thesis, Queensland University of Technology, July 1994.
9. C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, 2001.
10. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *ICSM*, pages 228–237, 1998.
11. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.
12. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 469–485. Springer-Verlag, 2001.

13. A. Flexeder, M. Petter, and H. Seidl. Analysis of Executables for WCET Concerns. Technical Report TUM-I0838, Technische Universität München, 2008.
14. J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.
15. J. Kinder, H. Veith, and F. Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, Jan 2009.
16. J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
17. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symp. on Principles of Programming Languages (POPL)*, pages 330–341, 2004.
18. M. O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.
19. F. B. Ramon E. Moore. *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2.)*. Soc for Industrial & Applied Math, 1979.
20. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, 2006.
21. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45, Washington, DC, USA, 2002. IEEE Computer Society.
22. A. Simon. Splitting the control flow with boolean flags. In *SAS '08: Proceedings of the 15th international symposium on Static Analysis*, pages 315–331, Berlin, Heidelberg, 2008. Springer-Verlag.
23. S. Sobek and K. Burke. *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*. Freescale Semiconductor, 2004. `http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf`.
24. H. Theiling. Extracting safe and precise control flow from binaries. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.
25. H. Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Universität des Saarlandes, 2003.