An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries

Johannes Kinder[1], Florian Zuleger[1,2,*], and Helmut Veith[1]

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

Abstract. Due to indirect branch instructions, analyses on executables commonly suffer from the problem that a complete control flow graph of the program is not available. Data flow analysis has been proposed before to statically determine branch targets in many cases, yet a generic strategy without assumptions on compiler idioms or debug information is lacking.

We have devised an abstract interpretation-based framework for generic low level programs with indirect jumps which safely combines a pluggable abstract domain with the notion of partial control flow graphs. Using our framework, we are able to show that the control flow reconstruction algorithm of our disassembly tool Jakstab produces the most precise overapproximation of the control flow graph with respect to the used abstract domain.

1 Introduction

One of the main problems when analyzing low level code, such as x86 assembly language, are indirect branch instructions. These correspond to goto statements where the target is calculated at runtime, or the use of function pointers combined with pointer arithmetic in high level languages. In executables, any address in the code is a potential target of an indirect branch, since in general there are no explicit labels. Failure to statically resolve the target of an indirect branch instruction thus leads to an either incomplete or grossly overapproximated control flow graph. Often, data flow analysis can aid in resolving such indirect branches; however, data flow analysis already requires a precise control flow graph to work on. This seemingly paradox situation has been referred to as an inherent "chicken and egg" problem in the literature [1,2].

In this paper, we show that this notion is overly pessimistic. We present a framework to construct a safe overapproximation of the control flow graph of low level programs by effectively combining control and data flow analysis by means of abstract interpretation. Existing approaches to control flow extraction from binaries usually either make a best effort attempt and accept possible unsoundness [3,4], or they make optimistic assumptions on clean code layout [5] or on the presence of additional information such as symbol tables or relocation information [2]. Our approach is designed to be generic in the sense that it does not require any additional information besides the actual instructions and is still able to compute a sound and precise overapproximation of the control flow graph. In particular, our paper makes the following contributions:

* The research of the second author was supported in part by Microsoft Research through its PhD Scholarship Programme.

– We define an abstract interpretation that reconstructs the control flow graph and is parameterized by a given abstract domain. We achieve this by extending the given abstract domain with a representation of the partial control flow graph. To this end, we define the notion of a control flow graph for a low level assembly-like language based on a concrete semantics in Section 3.2. We construct a *resolve* operator, based on conditions imposed on the provided abstract domain, for calculating branch targets. Using this operator, our analysis is able to safely overapproximate the control flow graph (Section 3.3).
– We present a general extension of the classical worklist algorithm met in program analysis which empowers control flow reconstruction by data flow analyses under very general assumptions. The algorithm overcomes the "chicken and egg" problem by computing the a priori unknown edges on the fly by using the *resolve* operator. We prove that the algorithm always returns the most precise overapproximation of the program's actual control flow graph with respect to the precision of the provided abstract domain used by the data flow analysis (Section 3.4).
– In earlier work, we presented our disassembly tool JAKSTAB [6], which employs constant propagation for an iterative disassembly strategy. JAKSTAB uses an abstract domain which supports symbolic memory addresses to achieve constant propagation through local variables and yielded better results than the most widely used commercial disassembler IDA Pro. We describe in Section 4 how the control flow reconstruction algorithm implemented in JAKSTAB instantiates our newly defined abstract interpretation. Thus, without the need to restart constant propagation, it always computes a safe overapproximation of the control flow graph.

2 Overview

In this section we describe current disassembly techniques and their shortcomings. We explain why proposed augmentations of disassembly with data flow analysis suffer from imprecision and we motivate how to overcome these difficulties by an intertwined control and data flow analysis.

2.1 Disassembly

Disassembly is the process of translating a sequence of bytes into an assembly language program. Simple *linear sweep* disassemblers, such as GNU objdump, sequentially map all bytes to instructions. Especially on architectures with varying instruction length (e.g. Intel x86) this leads to erroneous assembly programs, as these disassemblers easily lose the correct alignment of instructions because of data or padding bytes between code blocks. *Recursive traversal* disassemblers interpret branch instructions in the program to translate only those bytes which can actually be reached by control flow. The disassembler, however, cannot always determine the target of a branch instruction and can thus miss parts of the program.

To avoid this problem, disassemblers usually augment recursive traversal by heuristics to detect potential pieces of code in the executable. These heuristics exploit the presence of known compiler idioms, such as recurring procedure prologues or common patterns in the calculation of switch-jumps from jump tables [4]. While this works well for the majority of compiler generated code, the presence of hand-crafted assembly code and the effects of aggressive compiler optimizations can thwart heuristic methods. Moreover, heuristics in disassembly are prone to creating false positives, i.e., to misinterpret data as code. Because of these well known problems, improved methods of disassembly that incorporate data flow analysis have been subject to research.

2.2 Augmenting Disassembly with Data Flow Analysis

Data flow analysis statically calculates information about the program variables from a given program representation. Earlier work [1,5,3,7,6] has shown that data flow analysis can be used to augment the results of disassembly, but no conclusive answer was given on the best way to handle states with unresolved control flow successors during data flow analysis. Further, updating the control flow graph could render previous data flow information invalid, which would require backtracking and could cause the analysis to diverge.

De Sutter et al. [7] suggested to initially connect all indirect jumps to a virtual *unknown* node for indirect jumps, which effectively overapproximates the control flow graph. In an iterative process, they use constant propagation on the overapproximated graph to show infeasibility of most overapproximated edges, which can then be removed. This approach is inspired by the solution of Chang et al. [8] to the similar problem of treating unknown external library functions in the analysis of C programs. We exemplify De Sutter et al.'s method by applying it to the snippet of pseudo-assembly code shown in the left of Figure 1. The center of the figure depicts the corresponding initial control flow graph, where the indirect jump at line 2 is connected to the unknown node ($\top$). There are outgoing edges from the unknown node to all statements, since every address is a potential jump target in the general case of stripped code without relocation information. Calculating the possible values of x, we see that x can in fact take the concrete values $5, 3, 1, -1, \ldots$ at the entry of line 2 in the overapproximated program. Thus a program analysis operating on this initial overapproximation can only conclude that addresses 2 and 4 are no targets of the jump, but cannot remove the overapproximated edges to addresses 1 and 3. The final CFG reconstructed by this method,

```
1: x := 5
2: jmp x
3: x := x − 2
4: jmp 2
5: halt
```

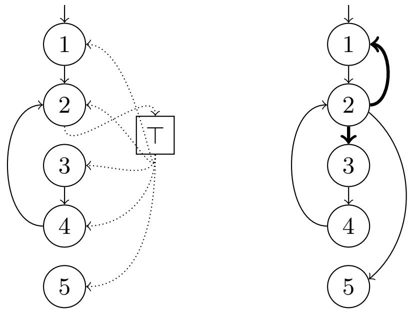

Fig. 1. Overapproximation of the CFG by adding an unknown node ($\top$) leads to additional possible values for x at the indirect jump

shown on the right of Figure 1, consequently contains the infeasible edges (2,1) and (2,3) (drawn in bold).

This example shows that integration of data flow algorithms with control flow reconstruction is non-trivial and can lead to suboptimal results. In the rest of this paper we will demonstrate how to successfully design program analyses which reconstruct the control flow of a disassembled low level program.

2.3 Integrating Fixed Point Equations for Control and Data Flow Analysis

Our approach to control flow reconstruction is based on the idea of executing data flow analysis and branch resolution simultaneously. A data flow problem is characterized by a constraint system derived from an overapproximation of the program semantics. The solution to a data flow problem is calculated by iteratively applying the constraints until a fixed point is reached. These constraints encode the complete control flow graph (by an edge relation), which, however, is unavailable as our task exactly is the control flow reconstruction.

The intuition of our approach is that we can grow the edge relation during the fixed point iteration, until a simultaneous least fixed point of both data and control flow is reached. For growing the edge relation, we introduce a $\mathbf{resolve}$ operator that uses data flow information to calculate branch targets of instructions. In our combined analysis, we will ensure that

– the quality of the fixed point, and thus of the reconstructed control flow graph, does not depend on the order in which the constraints are applied.
– the fixed point of control and data flow is a valid abstraction of the concrete program behavior.

3 Abstract Interpretation of Low Level Languages

In this section we formally define our combined analysis and prove the above properties. First, we introduce our low level language, then its concrete semantics, and finally we state our abstract interpretation framework. Our notation follows standard program analysis literature [9].

3.1 A Simple Low Level Language

We restrict ourselves to a simple low level language, JUMP, which captures the specifics of assembly language. JUMP uses the *program counter pc*, a finite set of *integer variables* $V = \{v_1, \ldots, v_n\}$, and a *store* $m[\cdot]$. For generality we do not further specify the *expressions* in JUMP, even though we explicitly note that expressions may contain the program counter, the variables and the store. We denote the set of expressions by $\mathbf{Exp}$. A *statement* in JUMP can be either

– a *variable assignment* $v := e$, where $v \in V$ and $e \in \mathbf{Exp}$, which assigns the value of an expression e to the variable v,
– a *store assignment* $m[e_1] := e_2$, where $e_1, e_2 \in \mathbf{Exp}$, which assigns the value of e_2 to the store location specified by e_1,

- a *guarded jump* statement $\mathsf{jmp}\ e_1, e_2$, with $e_1, e_2 \in \mathbf{Exp}$, which transfers control to the statement at the address calculated from expression e_2 if the expression e_1 evaluates to 0,
- or the *program exit* statement halt, which terminates execution of the program.

We denote the set of statements by $\mathbf{Stmt}$. The set of program *addresses* $\mathbf{A} \subseteq \mathbb{Z}$ is a *finite* subset of the integers. A program in JUMP is a finite partial mapping of addresses to statements. The idea is that every program has a fixed finite representation. At first we will assume that all addresses in $\mathbf{A}$ correspond to statements of the program. After we finish control flow reconstruction, we establish that some statements are not reachable and we can conclude that they are not part of the program (e.g., pieces of data intermixed with code blocks or misaligned statements on architectures with variable length instructions). Every program in our language JUMP has a unique starting address $\mathbf{start}$. The mapping between addresses and statements is expressed by $[\mathit{stmt}]^a$, where $\mathit{stmt} \in \mathbf{Stmt}$ and $a \in \mathbf{A}$. We present the formal semantics of JUMP in the next section.

JUMP is representative for assembly languages, since the most problematic features of machine code, indirect jumps and indirect addressing, are fully supported. Intuitively, it forms a minimalist intermediate representation for machine code. For simplicity JUMP does not implement explicit call and return instructions as these can be implemented by storing the program counter and jumping to the procedure, and jumping back to the stored value, respectively.

3.2 Semantics of JUMP

The semantics of JUMP is defined in terms of *states*. The set of states $\mathbf{State} := \mathbf{Loc} \times \mathbf{Val} \times \mathbf{Store}$ is the product of the *location valuations* $\mathbf{Loc} := \{pc\} \to \mathbf{A}$, the *variable valuations* $\mathbf{Val} := V \to \mathbb{Z}$ and the *store valuations* $\mathbf{Store} := \mathbb{Z} \to \mathbb{Z}$. We refer to the part of a state that represents an element of $\mathbf{Store}$ by a function $m[\cdot]$. As a state s is a function, we denote by $s(pc)$ the value of the program counter, by $s(v_i)$ the value of a variable v_i, and by $s(m[c])$ the value of the store mapping for an integer $c \in \mathbb{Z}$. We denote by $s[\cdot \mapsto \cdot]$ the state we obtain after substituting a new value for either the program counter, a variable, or a store mapping in s. We assume the existence of a deterministic *evaluation* function $\mathbf{eval} : \mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ ($\to$ is right-associative, i.e., $\mathbf{Exp} \to \mathbf{State} \to \mathbb{Z}$ stands for $\mathbf{Exp} \to (\mathbf{State} \to \mathbb{Z})$). We now define the operator $\mathbf{post} : \mathbf{Stmt} \to \mathbf{State} \to \mathbf{State}$:

$$\begin{aligned}
\mathbf{post}[\![v := e]\!](s) &:= s[v \mapsto \mathbf{eval}[\![e]\!](s)][pc \mapsto s(pc)+1] \\
\mathbf{post}[\![m[e_1] := e_2]\!](s) &:= s[m[\mathbf{eval}[\![e_1]\!](s)] \mapsto \mathbf{eval}[\![e_2]\!](s)][pc \mapsto s(pc)+1] \\
\mathbf{post}[\![\mathsf{jmp}\ e_1, e_2]\!](s) &:= \begin{cases} s[pc \mapsto \mathbf{eval}[\![e_2]\!](s)] & \text{if } \mathbf{eval}[\![e_1]\!](s) = 0 \\ s[pc \mapsto s(pc)+1] & \text{otherwise} \end{cases}
\end{aligned}$$

Remark 1. For the ease of explanation we have chosen to assume that all statements are of length 1, and therefore the program counter is increased by 1 for fall-through edges. Note that it would make no conceptual difference to introduce a length function that calculates the appropriate next location for every statement.

For later use in the definition of control flow graphs and in control flow reconstruction we define a language $\mathbf{Stmt}^{\#}$ derived from the language $\mathbf{Stmt}$, which consists of *assignments* $v := e$, $m[e_1] := e_2$ and labeled *assume* statements $\mathsf{assume}_a(e = 0)$, $\mathsf{assume}_a(e \neq 0)$, where $e, e_1, e_2 \in \mathbf{Exp}, a \in \mathbf{A}$, but which does not contain guarded jump statements. The intuition is that the assume statements correspond to resolved jump statements of the language $\mathbf{Stmt}$, where the labels specify resolved target addresses of the jump statements. We overload the operator $\mathbf{post} : \mathbf{Stmt}^{\#} \to \mathbf{2}^{\mathbf{State}} \to \mathbf{2}^{\mathbf{State}}$ to work on statements of the derived language and sets of states $S \subseteq \mathbf{State}$:

$$\begin{aligned}
\mathbf{post}[\![v := e]\!](S) \quad & := \{\mathbf{post}[\![v := e]\!](s) \mid s \in S\}, \\
\mathbf{post}[\![m[e_1] := e_2]\!](S) \quad & := \{\mathbf{post}[\![m[e_1] := e_2]\!](s) \mid s \in S\}, \\
\mathbf{post}[\![\mathsf{assume}_a(e = 0)]\!](S) \quad & := \{s[pc \mapsto a] \mid \mathbf{eval}[\![e]\!](s) = 0, s \in S\}, \\
\mathbf{post}[\![\mathsf{assume}_a(e \neq 0)]\!](S) \quad & := \{s[pc \mapsto a] \mid \mathbf{eval}[\![e]\!](s) \neq 0, s \in S\}.
\end{aligned}$$

Note that the definition of the $\mathbf{post}$ operator over sets makes use of the $\mathbf{post}$ operator for single elements in the case of assignments. We will need $\mathbf{Stmt}^{\#}$ and the transfer function $\mathbf{post}$ when stating the conditions we require from the abstract domain for our control flow reconstruction in Section 3.3.

A *trace* σ of a program is a finite sequence of states $(s_i)_{0 \leq i \leq n}$, such that $s_0(pc) = \mathbf{start}$, *stmt* is not halt for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$, and $s_{i+1} = \mathbf{post}[\![stmt]\!](s_i)$ for all $[stmt]^{s_i(pc)}$ with $0 \leq i < n$. Note that we do not impose conditions on variable or store valuations for state s_0. We denote the set of all traces of a program by $\mathbf{Traces}$. Further, we assume the program counter of all states in all traces to only map into the finite set of addresses $\mathbf{A}$, as every program has a fixed finite representation.

The definition of control flow graphs of programs in JUMP is based on our definition of traces and uses labeled edges. We define the set of labeled edges $\mathbf{Edge}$ to be $\mathbf{A} \times \mathbf{Stmt}^{\#} \times \mathbf{A}$.

Definition 1 ((Trace) Control Flow Graph). *Given a trace $\sigma = (s_i)_{0 \leq i \leq n}$, the trace control flow graph (TCFG) of σ is*

$$\begin{aligned}
TCFG(\sigma) = \{ & (s_i(pc), stmt, s_{i+1}(pc)) \mid \\
& \qquad 0 \leq i < n \text{ with } [stmt]^{s_i(pc)}, \text{ where } stmt \text{ is } v := e \text{ or } m[e_1] := e_2\} \\
\cup \{ & (s_i(pc), \mathsf{assume}_{s_{i+1}(pc)}(e_1 = 0), s_{i+1}(pc)) \mid \\
& \qquad 0 \leq i < n \text{ with } [\mathsf{jmp}\ e_1, e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) = 0\} \\
\cup \{ & (s_i(pc), \mathsf{assume}_{s_{i+1}(pc)}(e_1 \neq 0), s_{i+1}(pc)) \mid \\
& \qquad 0 \leq i < n \text{ with } [\mathsf{jmp}\ e_1, e_2]^{s_i(pc)} \text{ and } \mathbf{eval}[\![e_1]\!](s_i) \neq 0\}.
\end{aligned}$$

The control flow graph (CFG) is the union of the TCFGs of all traces

$$CFG = \bigcup_{\sigma \in \mathbf{Traces}} TCFG(\sigma).$$

As stated in the above definition, the CFG of a program is a semantic property, not a syntactic one, because it depends on the possible executions.

3.3 Control Flow Reconstruction by Abstract Interpretation

For the purpose of CFG reconstruction we are interested in abstract domains $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$, where

- $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq)$ is a complete lattice,
- the *concretization* function $\gamma : L \to \mathbf{2}^{\mathbf{State}}$ is monotone, i.e.,

$$\ell_1 \sqsubseteq \ell_2 \Rightarrow \gamma(\ell_1) \subseteq \gamma(\ell_2) \quad \text{for all } \ell_1, \ell_2 \in L,$$

  and maps the least element to the empty set, i.e., $\gamma(\bot) = \emptyset$,
- the *abstract operator* $\widehat{\mathbf{post}} : \mathbf{Stmt}^{\#} \to L \to L$ overapproximates the concrete transfer function $\mathbf{post}$, i.e.,

$$\mathbf{post}[\![stmt]\!](\gamma(\ell)) \subseteq \gamma(\widehat{\mathbf{post}}[\![stmt]\!](\ell)) \quad \text{for all } stmt \in \mathbf{Stmt}^{\#}, \ell \in L, \text{ and}$$

- the *abstract evaluation* function $\widehat{\mathbf{eval}} : \mathbf{Exp} \to L \to L$ overapproximates the concrete evaluation function, i.e.,

$$\mathbf{eval}[\![e]\!](\gamma(\ell)) \subseteq \gamma(\widehat{\mathbf{eval}}[\![e]\!](\ell)) \quad \text{for all } e \in \mathbf{Exp}, \ell \in L.$$

In the following we define a control flow analysis based on an abstract domain $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$. Our control flow analysis works on a *Cartesian abstract domain* $D : \mathbf{A} \to L$ and a *partial control flow graph* $F \subseteq \mathbf{Edge}$. The fact that edges are labeled with statements from $\mathbf{Stmt}^{\#}$ enables us to combine the abstract domain with the control flow reconstruction nicely.

A control flow analysis must have the ability to detect the (possibly overapproximated) set of targets of guarded jumps based on the knowledge it acquires. To this end, we define the operator $\mathbf{resolve} : \mathbf{A} \to L \to \mathbf{2}^{\mathbf{Edge}}$, using the functions available in the abstract domain. For a given address a and a lattice element ℓ, $\mathbf{resolve}$ returns a set of labeled control flow graph edges. If ℓ is the least element $\bot$, the location a has not been reached by the abstract interpretation yet, therefore no edge needs to be created and the empty set is returned. Otherwise, $\mathbf{resolve}$ labels fall-through edges with their respective source statements, or it calculates the targets of guarded jumps based on the information gained from the lattice element ℓ and labels the determined edges with their respective conditions:

$$\mathbf{resolve}_a(\ell) :=$$
$$:= \begin{cases} \emptyset & \text{if } \ell = \bot \\ \{(a, stmt, a+1)\} & \text{if } \ell \neq \bot \text{ and } ([stmt]^a \text{ is } [v := e]^a \\ & \qquad \text{or } [m[e_1] := e_2]^a) \\ \{(a, \mathsf{assume}_{a'}(e_1 = 0), a') \mid & \\ \quad a' \in \gamma\Big(\widehat{\mathbf{eval}}[\![e_2]\!]\big(\widehat{\mathbf{post}}[\![\mathsf{assume}_a(e_1 = 0)]\!](\ell)\big)\Big) \cap \mathbf{A}\} & \\ \qquad \cup\{(a, \mathsf{assume}_{a+1}(e_1 \neq 0), a+1)\} & \text{if } \ell \neq \bot \text{ and } [\mathsf{jmp}\ e_1, e_2]^a \end{cases}$$

The crucial part in the definition of the $\mathbf{resolve}$ operator is the last case, where the abstract operator $\widehat{\mathbf{post}}$ and the abstract $\widehat{\mathbf{eval}}$ are used to calculate possible jump targets.

Note that the precision of the abstract domain influences the precision of the control flow analysis.

We are now ready to state constraints such that all solutions of these constraints are solutions to the control flow analysis. The first component is the Cartesian abstract domain $D : \mathbf{A} \to L$, which maps addresses to elements of the abstract domain. The idea is that D captures the data flow facts derived from the program. The second component is the set of edges $F \subseteq \mathbf{Edge}$ which stores the edges we produce by using the $\mathbf{resolve}$ operator. Finally, we provide initial abstract elements $\iota^a \in L$ for every location $a \in \mathbf{A}$. Then the constraints are as follows:

$$F \supseteq \bigcup_{a \in \mathbf{A}} \mathbf{resolve}_a(D(a)) \tag{1}$$

$$D(a) \sqsupseteq \bigsqcup_{(a', stmt, a) \in F} \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup \iota^a \tag{2}$$

Note how it pays off that edges are labeled. The partial control flow graph F does not only store the a priori unknown targets on the guarded jumps, but also the conditions (assume statements) which have to be satisfied to reach them. This information can be used by the abstract $\widehat{\mathbf{post}}$ to propagate precise information.

The system of constraints (1) and (2) corresponds to a function

$$\begin{gathered}
G : \left((\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}\right) \to \left((\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}\right) \\
G(D, F) \mapsto (D', F'), \text{ where} \\
F' = \bigcup_{a \in \mathbf{A}} \mathbf{resolve}_a(D(a)), \\
D'(a) = \bigsqcup_{(a', stmt, a) \in F} \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup \iota^a.
\end{gathered}$$

The connection between constraints (1) and (2) and control flow analysis is stated in the following theorem (detailed proof in Appendix A[1]), whereby correctness notably depends on $\iota^{\mathbf{start}} \in L$:

Theorem 1. *Given a program in the language* JUMP *and a trace $\sigma = (s_i)_{0 \le i \le n}$, such that $s_0(pc) = \mathbf{start}$ and $s_0 \in \gamma(\iota^{\mathbf{start}})$, every solution (D, F) of the constraints (1) and (2) satisfies $s_n \in \gamma(D(s_n(pc)))$ and $TCFG(\sigma) \subseteq F$.*

The proof is a straightforward induction on the length of traces using the properties we require from the abstract domain. We immediately obtain:

Corollary 1. *Given a program in the language* JUMP *and a solution (D, F) of the constraints (1) and (2), where $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$, F is a superset of the CFG.*

The Cartesian abstract domain $\mathbf{A} \to L$, equipped with pointwise ordering, i.e., $D_1 \sqsubseteq D_2 :\Leftrightarrow \forall a \in \mathbf{A}.\, D_1(a) \sqsubseteq D_2(a)$, is a complete lattice, because L is a complete

[1] Appendices included in the full version of this paper, available on `http://jakstab.org`

lattice. The power set $\mathbf{2}^{\mathbf{Edge}}$ ordered by the subset relation $\subseteq$ is a complete lattice. The product lattice $(\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}$, equipped with pointwise ordering, i.e., $(D_1, F_1) \sqsubseteq (D_2, F_2) :\Leftrightarrow D_1 \sqsubseteq D_2 \wedge F_1 \subseteq F_2$, is complete as both $\mathbf{A} \to L$ and $\mathbf{2}^{\mathbf{Edge}}$ are complete. It can be easily seen that G is a monotone function on $(\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}$. As $(\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}}$ is a complete lattice, we deduce from the Knaster-Tarski fixed point theorem [10] the existence of a least fixed point μ of the function G. Therefore, the following proposition immediately follows:

Proposition 1. *The combined control and data flow problem, i.e., the system of constraints (1) and (2), always has a unique best solution.*

3.4 Algorithms for Control Flow Reconstruction

For the purpose of algorithm design we will focus on abstract domains L satisfying the ascending chain condition (ACC). We now present two CFG-reconstruction algorithms. The first algorithm (Algorithm 1) is generic and gives an answer to the "chicken and egg" problem as it computes a sound overapproximation of the CFG by an intertwined control and data flow analysis. We stress the fact that the order in which the CFG reconstruction is done may only affect efficiency but not precision. The second algorithm (Algorithm 2) is an extension of the classical worklist algorithm and is geared towards practical implementation.

The generic algorithm maintains a Cartesian abstract domain $D : \mathbf{A} \to L$ and a partial control flow graph $F \subseteq \mathbf{Edge}$. $D(a)$ is initialized by $\iota^{\mathbf{start}}$ for $a = \mathbf{start}$ (line 3) and by $\perp$ for $a \neq \mathbf{start}$ (line 2). As we do not know anything about the control flow graph of the program yet, we start with F as the empty set (line 4). The algorithm iterates its main loop as long as it can find an unsatisfied inequality (line 7, 8). Thus the algorithm essentially searches for violations of constraints (1) and (2). If the generic algorithm finds at least one not yet satisfied inequality, it nondeterministically picks a single unsatisfied inequality and updates it (lines 9 to 14).

We now state the correctness of Algorithm 1 for abstract domains L that satisfy the ascending chain condition (detailed proof in Appendix B):

Theorem 2. *Given a program in the language* JUMP, *where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$, *the generic CFG-reconstruction algorithm (Algorithm 1) computes a sound overapproximation of the CFG and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain L regardless of the non-deterministic choices made in line 9.*

Proof (sketch). The algorithm terminates because $(\mathbf{A} \to L) \times \mathbf{2}^{\mathbf{Edge}'}$, where $\mathbf{Edge}'$ is the finite subset of $\mathbf{Edge}$ that consists of all the edges that are potentially part of the program, satisfies the ascending chain condition. The fact that the algorithm always computes the most precise result heavily depends on the existence of the unique least fixed point μ of G. It is easy to show that the generic algorithm computes this least fixed point μ. As the least fixed point is the best possible result with respect to the precision of the abstract domain, it is always the most precise regardless of the non-deterministic choices made in line 9.

```
Input: a JUMP-program, its set of addresses $\mathbf{A}$ including $\mathbf{start}$, and the abstract domain
       $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an initial value $\iota^{\mathbf{start}}$
Output: a control flow graph
1  begin
2      forall $a \in \mathbf{A} \setminus \{\mathbf{start}\}$ do $D(a) := \bot$;
3      $D(\mathbf{start}) := \iota^{\mathbf{start}}$;
4      $F := \emptyset$;
5      while true do
6          $Choices := \emptyset$;
7          if $\exists (a', stmt, a) \in F.\, \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \not\sqsubseteq D(a)$ then $Choices := \{\text{do\_p}\}$;
8          if $\exists a \in \mathbf{A}.\, \mathbf{resolve}_a(D(a)) \nsubseteq F$ then $Choices := Choices \cup \{\text{do\_r}\}$;
9          if $\exists u \in Choices$ choose $u \in U$    /* non-deterministic choice */
10             switch $u$ do
11                 case do_p choose $(a', stmt, a) \in F$ where
                   $\widehat{\mathbf{post}}[\![stmt]\!](D(a')) \not\sqsubseteq D(a)$
12                     $D(a) := \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup D(a)$;
13                 case do_r choose $a \in \mathbf{A}$ where $\mathbf{resolve}_a(D(a)) \nsubseteq F$
14                     $F := \mathbf{resolve}_a(D(a)) \cup F$;
15         else
16             return F;
17 end
```

Algorithm 1. Generic CFG-reconstruction Algorithm

The worklist algorithm (Algorithm 2) is a specific strategy for executing the generic algorithm, where the partial control flow graph $F \subseteq \mathbf{Edge}$ is not kept as a variable, but implicit in the abstract values of the program locations. The initialization of D (lines 2, 3) is the same as in the generic algorithm. The algorithm maintains a worklist W, where it stores the edges for which data flow facts should be propagated later on. Every time the algorithm updates the information $D(a)$ at a location a (lines 3, 8), it calls the $\mathbf{resolve}$ operator (lines 4, 9) to calculate the edges which should be added to W. In every iteration of the main loop (lines 5 to 9) the algorithm non-deterministically picks an edge from the worklist by calling choose (line 6), and then shortens the worklist by calling rest (line 6). Subsequently, it checks for the received edge $(a', stmt, a)$, if an update is necessary (line 7), and in the case it is, it proceeds as already described.

From the correctness of the generic algorithm (1) we obtain the correctness of the worklist algorithm (proof in Appendix C):

Corollary 2. *Given a program in our language* JUMP*, where* $\{s \in \mathbf{State} \mid s(pc) = \mathbf{start}\} \subseteq \gamma(\iota^{\mathbf{start}})$*, the worklist CFG-reconstruction algorithm (Algorithm 2) computes a sound overapproximation of the CFG and terminates in finite time. Furthermore it returns the most precise result with respect to the precision of the abstract domain* L *regardless of the non-deterministic choices made in line 6.*

```
Input: a JUMP-program, its set of addresses $\mathbf{A}$ including $\mathbf{start}$, and the abstract domain
    $(L, \bot, \top, \sqcap, \sqcup, \sqsubseteq, \widehat{\mathbf{post}}, \widehat{\mathbf{eval}}, \gamma)$ together with an initial value $\iota^{\mathbf{start}}$
Output: a control flow graph
begin
    forall $a \in \mathbf{A} \setminus \{\mathbf{start}\}$ do $D(a) := \bot$;
    $D(\mathbf{start}) := \iota^{\mathbf{start}}$;
    $W := \mathbf{resolve}_{\mathbf{start}}(D(\mathbf{start}))$;
    while $W \neq \emptyset$ do
        $((a', stmt, a), W) := (\mathtt{choose}(W), \mathtt{rest}(W))$;
        if $\widehat{\mathbf{post}}[\![stmt]\!](D(a')) \not\sqsubseteq D(a)$ then
            $D(a) := \widehat{\mathbf{post}}[\![stmt]\!](D(a')) \sqcup D(a)$;
            $W := \mathtt{add}(W, \mathbf{resolve}_a(D(a)))$;
    $F := \emptyset$;
    forall $a \in \mathbf{A}$ do
        $F := F \cup \mathbf{resolve}_a(D(a))$;
    return F;
end
```

Algorithm 2. Worklist CFG-Reconstruction Algorithm

Proof (sketch). The worklist terminates because L satisfies the ascending chain condition. As the generic algorithm can always simulate the updates made by the worklist algorithm, the result computed by the worklist algorithm is always less or equal to the result of the generic algorithm, which is the least fixed point of G. On the other hand it can be shown that if the algorithm terminates, the result is greater or equal to the least fixed point of G.

Note that if the abstract domain L does not satisfy the ascending chain condition, it is possible to enhance the algorithms by using a widening operator to guarantee termination of the analysis. Such an algorithm would achieve a valid overapproximation of the CFG but lose the best approximation result stated in the above theorems, due to the imprecision induced by widening.

4 Instantiation of the Framework in the JAKSTAB Tool

We implemented the worklist algorithm for control flow reconstruction (Algorithm 2) in our disassembly and static analysis tool JAKSTAB [6]. JAKSTAB works on X86 executables, and translates them into an intermediate language that is similar in style but more complex than JUMP. We designed an abstract domain supporting constant propagation through registers (globally) and indirect memory locations (local to basic blocks) to parameterize the analysis, which yielded better results than the most widely used commercial disassembler IDA Pro. In this section we demonstrate how JAKSTAB integrates with our framework and sketch its abstract domain.

For supporting memory constants, JAKSTAB has to maintain an abstract representation of the store. When only dealing with memory accesses through constant addresses

	x	y	$m[x+2]$
start: $x := x + 2$	$(x + 2)$	$\top$	$\top$
2: $m[x] := 5$	$(x + 2)$	$\top$	5
3: $x := x + 1$	$(x + 3)$	$\top$	5
4: $x := x + 3$	$(x + 6)$	$\top$	5
5: $y := m[x - 4]$	$(x + 6)$	5	5
6: halt	$(x + 6)$	5	5

Fig. 2. Simple example for constant propagation through symbolic store locations. Abstract values calculated by the analysis are shown on the right.

(which is the case for global variables), this is trivial, since the store then behaves just like additional variables/registers. In compiled code, however, local variables are laid out on the stack, relative to the top of the current stack frame. They are manipulated by indirect addressing through the stack base pointer. For example, the instruction `mov [ebp - 4], 0` assigns 0 to the local variable at the top of the current stack frame. The exact value of the stack pointer, however, is only determined at runtime. Therefore, to successfully propagate constants through stack variables, our analysis must be able to handle indirect memory accesses symbolically, i.e., use symbolic store mappings from expressions to arbitrary expressions. The same holds true for fields in dynamically allocated memory structures, whose addresses are not statically known, either.

Support for symbolic store locations requires symbolic constants. Consider the simple program in Figure 2. The value of x is non-constant (because it is part of the input) and thus initialized to $\top$. To still propagate the assignment of 5 to the store location pointed to by x from line 2 to 5, the value of x has to be propagated symbolically, by forward substituting and partially evaluating the expressions. To this end, the lattice of abstract variable values contains symbolic expressions as an additional level of abstraction between integers and $\top$. Consequently, the mapping from store indices to integers has to be extended to a mapping $\mathbf{Exp} \to \mathbf{Exp}$. The join $\sqcup$ of the lattice for two elements with distinct values of the program counter is implemented by removing all symbolic mappings, retaining only mappings of variables to integers and from integer store locations to integers. This causes the scope of symbolic constant propagation to be limited to a single basic block. It also has the effect that the lattice L, which is of infinite height and satisfies the ascending chain condition; join points in loops always cause removal of mappings, thus every abstract state can only hold a finite number of mappings. Since ascending chains in the lattice remove one mapping per height level, the chains will always reach $\top$ after a finite number of steps.

The use of symbolic values has other implications as well. For updating symbolic values, the abstract $\widehat{\mathbf{post}}$ uses a substitution function that substitutes variables and memory expressions recursively with symbolic values from the abstract state. For substituting memory values, an aliasing check of store indices has to be performed. The abstract evaluation function $\widehat{\mathbf{eval}}$, which is used by our framework to resolve branch targets, uses substitution of symbolic store locations as well but ignores resulting symbolic values and only returns either integers, $\top$, or $\bot$. The concretization function γ maps each element of L to all concrete valuations matching the integer constants, disregarding symbolic mappings.

Using this abstract domain, JAKSTAB has already achieved good precision in reconstructing the control flow of executables [6]. Note that in the analysis of compiled applications, there are some cases when calls cannot be resolved by our current implementation. Most of the instances of unresolved control flow are due to function pointers inside structures being passed as parameters through several procedure calls. The local propagation of memory values in the abstract domain is currently not precise enough to capture such cases. Improvement of the propagation of memory values is a particular focus of ongoing work. The number of function pointer structures greatly depends on the implementation language of the program and the API functions used. In low level C code, the overwhelming majority of indirect calls result from compiler optimizations storing the addresses of frequently used API functions in registers, which JAKSTAB can reliably resolve.

5 Related Work

The problem of extracting a precise control flow graph from binaries has risen in several different communities of computer science. An obvious area is *reverse engineering* and in particular *decompilation*, where one aims to recover information about the original program, or preferably a close approximation of the original source code, from a compiled binary [11,12,13]. The compiler literature knows the concept of *link-time-* and *post-link-optimizers* [7,14], which exploit the fact that the whole program including libraries and hand-written assembly routines can be analyzed and optimized at link-time, i.e., after all code has been translated to binary with the symbol information still present. Precise tools for determining worst case execution time (WCET) of programs running on real time systems also have to process machine code, since they need to take compiler optimizations into account, and thus face similar problems of reconstructing the control flow [1,5,15]. Other applications of binary analysis include binary instrumentation [16], binary translation [17], or profiling [4].

Another prominent area of research that requires executable analysis is advanced *malware detection*. While classical malware detection relied on searching executables for binary strings (*signatures*) of known viruses, more recent advanced techniques focus on detecting patterns of malicious *behavior* by means of static analysis and model checking [18,19]. In this application domain, independence of the analysis from symbol information and compiler idioms is imperative, since malicious code is especially likely to have its symbols removed or to even be specially protected from analysis.

Due to the interest from these different communities, there has been a number of contributions to improving the results from disassembly. The literature contains a number of practical approaches to disassembly, which do not try to formulate a generalizable strategy. Schwarz et al. [2] describe a technique that uses an improved linear sweep disassembly algorithm, using relocation information to avoid misinterpreting data in a code segment. Subsequently, they run a recursive traversal algorithm on each function and compare results, but no attempt is made to recover from mismatching disassembly results. Harris and Miller [4] rely on identifying compiler idioms to detect procedures in the binary and to resolve indirect jumps introduced by jump tables. Cifuentes and van Emmerik [20] present a method to analyze jump tables by backward slicing through register assignments and computing compound target expressions for the indirect jumps. These compound expressions are then matched against three compiler-specific patterns of implementing switch statements.

There have also been several proposals for more general frameworks for reconstructing the control flow from binaries. In Section 2.2, we already discussed the approach by De Sutter et al. [7], which targets code with symbol and relocation information and uses an overapproximating unknown-node for unresolved branch targets. In his bottom-up disassembly strategy, Theiling [1] assumes architectures in which all jump targets can be computed directly form the instruction, effectively disallowing indirect jumps. For extending his method to indirect jumps, he also suggests the use of an overapproximating unknown node.

Kästner and Wilhelm [5] describe a top-down strategy for structuring executables into procedures and basic blocks. For this to work, they require that code areas of procedures must not overlap, that there must be no data between or inside procedures, and that explicit labels for all possible targets of indirect jumps are present. Compilers, however, commonly generate procedures with overlapping entry and exit points, even if the control flow graphs of the procedures are completely separate, so their top-down structuring approach cannot be used in general without specific assumptions about the compiler or target architecture.

The advanced executable analysis tool Codesurfer/X86, presented by Balakrishnan and Reps [3], extracts information about relations between values and computes an approximation of possible values based on the abstract domain of value sets. For disassembly, they rely on the capabilities of the commercial disassembler IDA Pro. While they are able to resolve missing control flow edges through value set analysis, their separation from disassembly prevents that newly discovered code locations can be disassembled. Furthermore, CodeSurfer/X86 is vulnerable to errors introduced by the heuristics based disassembly strategy of IDA Pro.

Although operating at higher language levels, the decompilation approach of Chang et al. [13] is similar in spirit to our framework. They connect abstract interpreters operating at different language levels, executing them simultaneously. One can interpret our data flow analysis and control flow reconstruction as separate decompilation stages of their framework. However, we do not restrict the execution order but allow nondeterministic fixed point iteration over both analyses and are still able to prove that the resulting control flow graph is optimal.

6 Conclusion

We have built a solid foundation for the concept of disassembling binary code by defining a generic abstract interpretation framework for control flow reconstruction. While analysis of machine code often requires ad hoc solutions and has many pitfalls, we believe that it greatly helps in the design of disassemblers and binary analysis tools to know that data flow guided disassembly does not suffer from a “chicken and egg” problem. Based on our framework, we plan to further extend our own disassembler JAKSTAB with an improved abstract domain to further reduce the need for overapproximation of control flow.

References

1. Theiling, H.: Extracting safe and precise control flow from binaries. In: 7th Int'l. Workshop on Real-Time Computing and Applications Symp (RTCSA 2000), pp. 23–30. IEEE Computer Society, Los Alamitos (2000)
2. Schwarz, B., Debray, S.K., Andrews, G.R.: Disassembly of executable code revisited. In: 9th Working Conf. Reverse Engineering (WCRE 2002), pp. 45–54. IEEE Computer Society, Los Alamitos (2002)
3. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, pp. 5–23. Springer, Heidelberg (2004)
4. Harris, L.C., Miller, B.P.: Practical analysis of stripped binary code. SIGARCH Comput. Archit. News 33(5), 63–68 (2005)
5. Kästner, D., Wilhelm, S.: Generic control flow reconstruction from assembly code. In: 2002 Jt. Conf. Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES 2002-SCOPES 2002), pp. 46–55. ACM Press, New York (2002)
6. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
7. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Trans. Program. Lang. Syst. 27(5), 882–945 (2005)
8. Chang, P.P., Mahlke, S.A., Chen, W.Y., Hwu, W.W.: Profile-guided automatic inline expansion for C programs. Softw., Pract. Exper. 22(5), 349–369 (1992)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
10. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5(2), 285–309 (1955)
11. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw., Pract. Exper. 25(7), 811–829 (1995)
12. van Emmerik, M., Waddington, T.: Using a decompiler for real-world source recovery. In: 11th Working Conf. Reverse Engineering (WCRE 2004), pp. 27–36. IEEE Computer Society Press, Los Alamitos (2004)
13. Chang, B., Harren, M., Necula, G.: Analysis of low-level code using cooperating decompilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 318–335. Springer, Heidelberg (2006)
14. Schwarz, B., Debray, S.K., Andrews, G.R.: PLTO: A link-time optimizer for the intel IA-32 architecture. In: Proc. Workshop on Binary Translation, WBT 2001 (2001)
15. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
16. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM Press, New York (2007)
17. Cifuentes, C., van Emmerik, M.: UQBT: Adaptive binary translation at low cost. IEEE Computer 33(3), 60–66 (2000)
18. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 174–187. Springer, Heidelberg (2005)
19. Christodorescu, M., Jha, S., Seshia, S.A., Song, D.X., Bryant, R.E.: Semantics-aware malware detection. In: IEEE Symp. Security and Privacy (S&P 2005), pp. 32–46. IEEE Computer Society, Los Alamitos (2005)
20. Cifuentes, C., van Emmerik, M.: Recovery of jump table case statements from binary code. Sci. Comput. Program. 40(2-3), 171–188 (2001)