Alternating Control Flow Reconstruction

Johannes Kinder[1] and Dmitry Kravchenko[2]

[1] École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
johannes.kinder@epfl.ch
[2] Technische Universität Darmstadt, D-64277 Darmstadt, Germany
dmitry.kravchenko@cased.de

Abstract. Unresolved indirect branch instructions are a major obstacle for statically reconstructing a control flow graph (CFG) from machine code. If static analysis cannot compute a precise set of possible targets for a branch, the necessary conservative over-approximation introduces a large amount of spurious edges, leading to even more imprecision and a degenerate CFG.
In this paper, we propose to leverage under-approximation to handle this problem. We provide an abstract interpretation framework for control flow reconstruction that alternates between over- and under-approximation. Effectively, the framework imposes additional preconditions on the program on demand, allowing to avoid conservative over-approximation of indirect branches.
We give an example instantiation of our framework using dynamically observed execution traces and constant propagation. We report preliminary experimental results confirming that our alternating analysis yields CFGs closer to the concrete CFG than pure over- or under-approximation.

1 Introduction

Binary machine code is an attractive analysis target for several reasons: when working with a fully compiled and linked binary, no interfacing with the build process is required and all program parts are available. Inline assembly and differences between programming languages or dialects are all eliminated by the translation to machine code. Moreover, there is often no source code available for a program of interest, e.g., when analyzing proprietary code such as third party device drivers, plugins, or potentially malicious software.

An important first step for applying static analysis to binaries is to recover an accessible program representation in the form of a control flow graph (CFG). The major problem in computing a CFG for low-level machine code lies in the treatment of indirect branches, i.e., instructions such as `jmp eax`, whose targets are computed at runtime. In earlier work [15,14], we showed how abstract interpretation [7] of machine code coupled with on the fly disassembly can compute an over-approximation of the concrete CFG: starting with an empty CFG, data flow information is propagated one instruction at a time. On encountering an indirect branch instruction, possible targets are computed from the partial data flow information and added to the CFG as new edges.

In a purely static approach, the over-approximate analysis has to be precise enough to compute accurate sets of target address values for each indirect branch. Because of

imprecise reasoning or because parts of the runtime environment (libraries, operating system behavior) have been abstracted away, however, the static analysis may have to conservatively over-approximate the possible target addresses. In that case, the CFG has to be extended with edges from the branch instruction to the entire program address space (including the branch instruction itself). These mostly spurious edges introduce additional imprecision into the analysis as abstract states propagate across them, in the end yielding a degenerate CFG that is unusable for analysis. In practice, existing tools [13] therefore either immediately report an error whenever they cannot resolve an indirect branch or simply turn the indirect jump into a leaf without any successors.

In the context of dynamic instrumentation, Nanda et al. proposed to resolve indirect branches at runtime, when the concrete jump target has already been computed [16]. In this paper, we generalize this idea to combining over-approximate static analysis with under-approximation and alternating between the two for resolving jump targets. We provide a formalization of our approach in terms of abstract interpretation, which allows us to be precise about the nature of the combined analysis and its results. The resulting framework is parameterized by an over-approximate analysis, an under-approximate analysis, and a predicate for influencing alternation between the two. We make the following contributions:

- We give a new formalization of low-level control flow reconstruction as abstract interpretation of a parameterized semantics (Section 3). In Section 4, we show how this semantics can be instantiated to obtain (i) the concrete semantics of a low-level language, (ii) our existing over-approximate control flow reconstruction [15], and (iii) a purely under-approximate control flow reconstruction.
- Based on these definitions, we present a combined over- and under-approximation as an instance of our framework (Section 5). We split the rule for branch instructions to implement alternation between the two approximations. This yields a well-understood blueprint for using dynamic information in control flow reconstruction, whereas previous approaches relied on ad-hoc solutions.
- We prove that the algorithm for alternating control flow reconstruction terminates (i.e, the alternation semantics has a fixpoint) and computes an over-approximation of the original program restricted by additional preconditions. Subsequent static analysis on the reconstructed CFG is sound with respect to these preconditions.
- We present preliminary experimental results (Section 6) confirming that alternating control flow reconstruction yields a clear quality improvement compared to pure under- or over-approximation (in terms of false positives and negatives) and is essential for obtaining CFGs of realistically-sized programs.

2 Overview

Let us first informally illustrate our approach using an example. Consider the program fragment in Figure 1a. It starts at location 0 and contains two indirect branches at locations 6 and 22. For this example, we will use an over-approximating static analysis that collects for each location and variable a set of explicit values up to size, say, five. At the first indirect branch from location 6, control is transferred to the address stored in a variable x. Assume that x can take the values 10, 15, or 20 at runtime, but that this

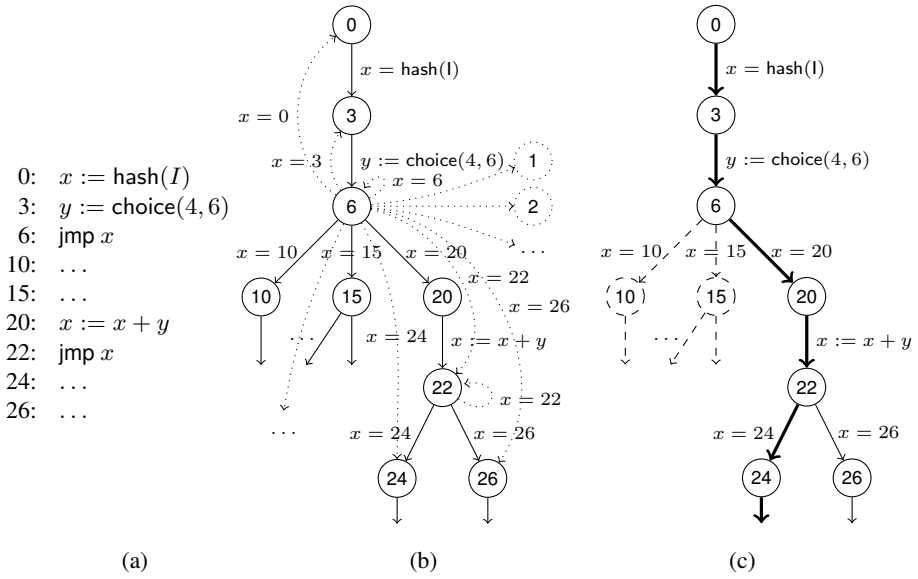

Fig. 1. Indirect jump example. (a) IL code, (b) CFG reconstructed by pure over-approximation, with spurious edges and nodes dotted, (c) CFG reconstructed by alternation using a single trace, with under-approximate edges in bold, over-approximate edges as regular lines, and missed edges dashed (assume edges are shown without an explicit "assume" keyword).

cannot be determined by our static analysis because x depends on the output of a hash function over some input value I. Assume further, that static analysis *can* determine the possible values of 4 and 6 for y, which are selected by some function choice. Note that our intermediate language used in the remainder of the paper does not contain an explicit function call but instead uses jumps. Here we use two calls for ease of exposition. Over-approximate control flow reconstruction by abstract interpretation proceeds on the program as follows. Starting from an empty CFG, the two edges $(0, x := \mathsf{hash}(I), 3)$ and $(3, y := \mathsf{choice}(4, 6), 6)$ are created (as simple syntactic fall-through edges) and interpreted, leading to the abstract state $(x = \top, y = \{4, 6\})$.

Upon reaching the indirect jump at location 6, the static analysis computes the set of concrete addresses corresponding to the current abstract value of x. This "downward" concretization yields a superset of the actual, concrete target addresses. For each possible concrete value V, an edge $(6, \mathsf{assume}\ x = V, V)$ is then added to the partial CFG. In general, this algorithm always produces a correct over-approximation of the concrete CFG, but its precision depends critically on whether the abstract value for x concretizes to an accurate set of concrete target addresses.

In our example, however, x is unknown and its concretization ranges over all values in the set $\mathbf{L}$ of program addresses. Thus, the control flow reconstruction algorithm has to add $|\mathbf{L}|$ edges from 6, which renders the resulting CFG useless for all practical purposes. Taking only the locations within the depicted program fragment into account, the CFG already degenerates to the form shown in Figure 1b. In this CFG, control flow can even

reach addresses such as 1 and 2, which lie between existing instructions.[1] Moreover, the imprecision of over-approximating the jump edges propagates: through the spurious edge $(6, \mathsf{assume}\ (x = 22), 22)$, the second indirect jump can be reached, leading to a self-loop at location 22.

To avoid such loss of precision, we propose *alternating control flow reconstruction*. As soon as a predefined condition χ is met, e.g., that the set of possible targets is unconstrained, the abstract interpreter switches to an under-approximate semantics. To this end, an under-approximation is maintained alongside the over-approximation. Under-approximate states concretize "upward" to a subset of the concrete values.

We can under-approximate the program semantics by simply executing the program with random input. Assume that, in our execution, we have in location 6 that $x = 20$ and $y = 6$. Then x concretizes upward to the singleton set $\{20\}$, and a corresponding edge $(6, \mathsf{assume}\ x = 20, 20)$ is created. The simultaneously over- and under-approximate analysis then continues along this edge, which under-approximates the set of concrete edges going out from location 6. From interpreting the edge, the over-approximation can deduce that $x = 20$, leading to the abstract state $(x = \{20\}, y = \{4, 6\})$. The next fall-through edge updates the state to $(x = \{24, 26\}, y = \{4, 6\})$, which allows the over-approximation to precisely resolve all edges from the indirect jump at location 22. The analysis thus explores the CFG by alternation, returning to over-approximation after passing over the unresolved first indirect jump using under-approximation. It missed two outgoing edges from location 6, but reconstructed a CFG (Figure 1c) that differs less from the concrete CFG than the over-approximation. The final CFG represents an over-approximation of the program restricted to $x = 20$ at location 2. Subsequent analyses on the CFG are then performed modulo the additional precondition $\mathsf{hash}(I) = 20$.

To obtain a fully over-approximate CFG without additional preconditions, the missing two indirect branches could be covered using dynamic test generation [9,10,17]. Note that complete control flow reconstruction by dynamic test generation alone requires full branch coverage, therefore it would need to create at least 5 tests. Together with alternating control flow reconstruction, both branches of the indirect jump at location 22 can be covered using just a single trace, requiring only 3 test cases for complete reconstruction.

3 Parameterized Semantics for Low-Level Control Flow

Computing the control flow graph in the presence of indirect jumps reduces to a reachability problem, since we want to find for each indirect jump all possible values that its target expression can evaluate to. Defining an analysis that is able to deal with indirect jumps is non-obvious, though, so in earlier work [15] we proposed a generic framework for control flow reconstruction that is parameterized by an over-approximate data flow analysis for a simplified language without indirect jumps. In the following, we provide a new and more flexible formalization of low-level control flow as a parameterized

[1] Such spurious inter-instruction locations cannot be simply ruled out, as "overlapping instructions" are actually legal. They lead to an alternative decoding of the same code bytes and are a popular anti-disassembly measure in hand-crafted assembly code.

semantics that allows to define concrete, under-, and over-approximating semantics in common terms.

3.1 Intermediate Language

To abstract from the details of assembly language while still capturing its nature, in particular the relevant low-level branching behavior, we define the intermediate language IL. The set of statements is denoted by $\mathbf{Stmt}$ and consists of assignments $m[e_1] := e_2$, conditional indirect jumps $\mathsf{if}\ e_1\ \mathsf{jmp}\ e_2$, assume statements $\mathsf{assume}\ e_1$, and the halt statement, where $e_1, e_2 \in \mathbf{Exp}$ denote expressions from a set $\mathbf{Exp}$ containing constants, memory dereferences $m[e]$, and the usual arithmetic, Boolean, and bitwise operators. For simplicity, we do not explicitly introduce variables; when we use identifiers such as x, these refer to fixed memory locations. The only IL data type are integers; Boolean true and false are represented by 1 and 0, respectively. The finite set of program *locations* $\mathbf{L} \subseteq \mathbb{N}$ denotes all addresses that are part of the program. Note that we could also use bit-vectors as data type (and indeed do so in our implementation); we choose to use integers in the exposition for greater generality. An IL program $\langle \ell_0, P \rangle$ consists of a unique start location ℓ_0 and a finite mapping $P :: \mathbf{L} \to (\mathbf{Stmt} \times \mathbf{L})$ from program locations to statements and successor locations. Successor locations identify the address of the syntactically next instruction and are used here to reflect variable length instruction sets. We will use the notation $[stmt]^{\ell}_{\ell'}$ to refer to the statement $stmt$ in P at location ℓ with successor location ℓ'.

Furthermore, we define a reduced language IL^- and its set of statements $\mathbf{Stmt}^-$, which consists only of assignments and assume statements.

3.2 Parameter Semantics Template

Our approach to control flow reconstruction is a parameterized framework in the form of an adapter semantics that lifts an existing semantics $\mathcal{S}$ for the simpler language IL^-. In a state that is about to execute an indirect jump next, the adapter semantics converts the jump into a set of assume edges pointing out to the targets that are feasible given the current state. Therefore $\mathcal{S}$ has to be defined only for assignments and assume statements, and a CFG is automatically created by collecting all the edges created by the adapter semantics. The parameter semantics $\mathcal{S} = (\mathcal{A}, \iota, f, \gamma[\![\cdot]\!], \cup^A)$ contains:

- A complete lattice $\mathcal{A} = (A, \sqcup, \sqcap, \sqsubseteq, \bot, \top)$.
- An initial lattice element ι.
- A transfer function $f[\![C]\!](S) :: (\mathbf{Stmt}^-, A) \to A$ for the statements in IL^-, i.e., for assignments and assume statements.
- A concretizing evaluation function $\gamma[\![e_1, \ldots, e_n]\!]S :: \mathbf{Exp}^n \times A \to \mathcal{P}(\mathbb{Z}^n)$ that abstractly evaluates and concretizes vectors of expressions $e_1, \ldots, e_n$ to a set of vectors of concrete values ($\mathcal{P}$ denotes the powerset). In principle, the concretizing evaluation function could be synthesized from a regular concretization function γ as $\gamma[\![e_1, \ldots, e_n]\!]S := \{\mathbf{eval}[\![e_1]\!](S^\natural), \ldots, \mathbf{eval}[\![e_n]\!](S^\natural) \mid S^\natural \in \gamma(S)\}$, where $\mathbf{eval}$ denotes concrete expression evaluation, but is usually implemented directly.
- An operator $\cup^A :: A \times A \to A$ that is used by the adapter semantics to combine elements of the domain A reaching the same location. $\cup^A$ may be identical to $\sqcup$ but does not have to be.

$$\frac{[m[A] := E]^{\ell}_{\ell'} \quad f[\![m[A] := E]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle \ell', S', G \uplus (\ell, \ell')\rangle}\ \textsc{Assignment}$$

$$\frac{[\textsf{if } B \textsf{ jmp } E]^{\ell}_{\ell'} \quad (1, V) \in \gamma[\![B, E]\!]S \quad f[\![\textsf{assume } (B \wedge E = V)]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle V, S', G \uplus (\ell, V)\rangle}\ \textsc{Jump-True}$$

$$\frac{[\textsf{if } B \textsf{ jmp } E]^{\ell}_{\ell'} \quad 0 \in \gamma[\![B]\!]S \quad f[\![\textsf{assume } (\neg B)]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle \ell', S', G \uplus (\ell, \ell')\rangle}\ \textsc{Jump-False}$$

$$\frac{[\textsf{assume } E]^{\ell}_{\ell'} \quad f[\![\textsf{assume } E]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle \ell', S', G \uplus (\ell, \ell')\rangle}\ \textsc{Assume}$$

$$\frac{[\textsf{halt}]^{\ell}_{\ell'}}{\langle \ell, S, G\rangle \longrightarrow \langle \ell, S, G\rangle}\ \textsc{Halt}$$

Fig. 2. Control flow resolving transition system for an IL program $\langle \ell_0, P\rangle$, parameterized by a forward transfer function f for IL$^-$.

3.3 Control Flow Semantics

Using these parameters, we can define the full IL transition relation $\longrightarrow$ shown in Figure 2. It relates states from the set $\mathbf{L} \times A \times \mathcal{P}(\mathbf{L} \times \mathbf{L})$ consisting of the value of the program counter $\ell \in \mathbf{L}$, the memory state $S \in A$, and the partial CFG $G \subseteq \mathbf{L} \times \mathbf{L}$.

Depending on the statement pointed to by the current program counter value, the rules invoke the transfer function f, update the program counter, and add the current edge to the partial CFG. Assignments and assume statements simply fall through to the syntactically next statement, and halt statements terminate execution. The rule JUMP-TRUE fires if the jump condition is possibly true in the current state and determines the next program counter value from the concretization of the target expression. The rule invokes the transfer function to assume the evaluations of the condition and target expressions, updates the program counter to the jump target, and adds the new edge. The rule JUMP-FALSE fires if the condition of a jump is possibly false, and transfers control to the fall-through successor, assuming the negative evaluation of the condition. The reachability semantics for IL then is the least fixpoint of a function F defined as

$$F(D) := \{\langle \ell_0, \iota, \varnothing\rangle\} \cup \dot{\bigsqcup}\{\langle \ell', S', G'\rangle \mid \exists \langle \ell, S, G\rangle \in D : \langle \ell, S, G\rangle \longrightarrow \langle \ell', S', G'\rangle\}$$

with

$$\langle \ell, S, G\rangle \,\dot{\cup}\, \langle \ell', S', G'\rangle = \begin{cases} \{\langle \ell,\ S \cup^A S',\ G \cup G'\rangle\} & \text{if } \ell = \ell' \\ \{\langle \ell, S, G\rangle, \langle \ell', S', G'\rangle\} & \text{otherwise} \end{cases}$$

as the pointwise merge function that combines states reaching the same location. The reconstructed CFG then results from

$$CFG = \bigcup\{G \mid \langle \ell, S, G\rangle \in \operatorname{lfp} F\}.$$

4 Instantiating the Parameterized Semantics

Using the template from Section 3.2, we instantiate our semantics framework with semantics for IL$^-$ to obtain concrete, over-approximate, and under-approximate control flow semantics for IL.

4.1 Concrete Semantics

We can now instantiate our framework with a concrete reachability semantics for IL$^-$ to define the concrete semantics of IL. The semantics of IL$^-$ is defined in terms of sets of concrete memory states from $\mathcal{P}(\mathbb{N} \to \mathbb{Z})$, where each memory state maps positive integers (representing memory addresses) to integers. The parameters required by our adapter semantics are given by

- the powerset lattice of concrete memory states $(\mathcal{P}(\mathbb{N} \to \mathbb{Z}), \cup, \cap, \subseteq, \varnothing, \mathbb{N} \to \mathbb{Z})$,
- the initial state set $\iota = \mathbb{N} \to \mathbb{Z}$,
- the concrete transfer function over sets of memory states

$$f^\natural[\![m[A] := E]\!](S) = \{s' \mid \exists s \in S : s' = s[\mathbf{eval}[\![A]\!](s) \mapsto \mathbf{eval}[\![E]\!](s)]\}$$
$$f^\natural[\![\mathsf{assume}\ E]\!](S) = \{s \in S \mid \mathbf{eval}[\![E]\!](s) = 1\},$$

- the concrete evaluation function $\gamma^\natural[\![e_1, \ldots, e_n]\!]S := \{v_1, \ldots, v_n \mid \exists s \in S : v_1 = \mathbf{eval}[\![e]\!](s), \ldots, v_n = \mathbf{eval}[\![e_n]\!](s)\}$ ($\gamma^\natural$ is the identity function),
- and the merging operator $\cup$ for taking the union of two sets of memory states.

This fully instantiated concrete semantics for IL is the reference point for the following over- and under-approximations. Defining the concrete semantics in terms of our framework allows to simplify the correctness proofs for abstract semantics. If the framework is shared by both semantics, it suffices to only establish a correctness relation between the concrete and abstract transfer functions for IL$^-$.

We define the *concrete control flow graph* of a program as the CFG constructed by the concrete semantics. This definition is equivalent to the one in [15].

4.2 Over-Approximate Semantics

If instantiated with over-approximate IL$^-$ semantics, our adapter semantics computes an over-approximation of the concrete CFG. The over-approximation $f^\sharp$ of the concrete transfer function $f^\natural$ has to satisfy $\gamma^\sharp \circ f^\sharp(S^\sharp) \supseteq f^\natural \circ \gamma^\sharp(S^\sharp)$ for a concretization function $\gamma^\sharp$ from abstract states to concrete sets of states. Any over-approximate forward analysis is suitable here; for useful results in control flow reconstruction, however, the concretizing evaluation function should be able to produce reasonably precise address values for jump targets, however.

The fixpoint of the fully instantiated over-approximate semantics can be computed using chaotic iteration over the locations in the partial CFG, which yields the algorithm for static control flow reconstruction [15]. The resulting CFG over-approximates the concrete CFG, i.e., is a superset of its edges. In the present formalization of the framework, the over-approximate domain must not contain infinite ascending chains to ensure termination. It is straightforward to remove this requirement by including widening (and possibly narrowing), but we omit this extension for simplicity.

4.3 Under-Approximate Semantics

An under-approximate semantics is given using exactly the same parameters as an over-approximate one is, but has dual requirements for correctness. The under-approximation $f^\flat$ of the concrete transfer function $f^\natural$ has to satisfy $\gamma^\flat \circ f^\flat(S^\flat) \subseteq f^\natural \circ \gamma^\flat(S^\flat)$ for the concretization function $\gamma^\flat$ from under-approximate states to sets of concrete states.

We now define an under-approximate semantics $(\mathcal{T}, \iota, f^\flat, \gamma^\flat[\![\cdot]\!], \cup^T)$ that replays a set of concrete execution traces. It is essentially equivalent to the concrete semantics but also maintains a total instruction counter for timing out after executing a predefined number N of instructions. The parameters instantiating our framework are

- the product lattice of (i) the chain of integer counters and (ii) the powerset lattice of memory states, $\mathcal{T} = ([0; N] \times \mathcal{P}(\mathbb{N} \to \mathbb{Z}), \sqcup^T, \sqcap^T, \sqsubseteq^T, (0, \varnothing), (N, \mathbb{N} \to \mathbb{Z}))$,
- the initial state $\iota = (0, S)$, which contains the zero counter and a finite set of memory configurations (e.g., command line parameters, file contents, ...) to initialize the executions.
- the under-approximate transfer function $f^\flat(S^\flat) = \begin{cases} (n+1, f^\natural(S^\natural)) & n < N \\ (N, \varnothing) & \text{otherwise} \end{cases}$ with $S^\flat = (n, S^\natural)$, which satisfies $\gamma^\flat \circ f^\flat(S^\flat) \subseteq f^\natural \circ \gamma^\flat(S^\flat)$.
- the concretizing evaluation function $\gamma^\flat[\![e_1, \ldots, e_n]\!](n, S^\natural) := \{v_1, \ldots, v_n \mid \exists s \in S^\natural : v_1 = \mathbf{eval}[\![e]\!](s), \ldots, v_n = \mathbf{eval}[\![e_n]\!](s)\}$.
- and the merging operator $\cup^T$ with $(n_1, S_1^\natural) \cup^T (n_2, S_2^\natural) = (\mathsf{max}(n_1, n_2), S_1^\natural \cup S_2^\natural)$ for taking the union of two sets of memory states and setting the counter to the maximum of both individual counters.

Fixpoint computation for the under-approximate semantics is an algorithm for purely dynamic control flow reconstruction. The resulting CFG under-approximates the concrete CFG, i.e., is a subset of its edges. Termination of the fixpoint computation is ensured by the instruction counter.

5 Alternation in Control Flow Reconstruction

To leverage alternation in control flow reconstruction, we perform a simultaneous over- and under-approximation that can be plugged into our parameterized framework in the same way as the concrete and over-approximate semantics before.

5.1 Combined Semantics

We now define an intermediate semantics that allows to plug an over- and an under-approximation into our framework simultaneously. It is itself parameterized by an over-approximation $(\mathcal{A}, \iota^\sharp, f^\sharp, \gamma^\sharp[\![\cdot]\!], \cup^\sharp)$ and an under-approximation $(\mathcal{U}, \iota^\flat, f^\flat, \gamma^\flat[\![\cdot]\!], \cup^\flat)$, with $\mathcal{A} = (A, \sqcup^\sharp, \sqcap^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp)$ and $\mathcal{U} = (U, \sqcup^\flat, \sqcap^\flat, \sqsubseteq^\flat, \bot^\flat, \top^\flat)$. The combined intermediate semantics is defined as $(\mathcal{C}, \iota^\diamond, f^\diamond, \gamma^\diamond[\![\cdot]\!], \cup^\diamond)$ with:

- The complete lattice $\mathcal{C} = (A \times U, \sqcup^\diamond, \sqcap^\diamond, \sqsubseteq^\diamond, (\bot^\sharp, \bot^\flat), (\top^\sharp, \top^\flat))$ with pairwise definitions of $\sqcup^\diamond$, $\sqcap^\diamond$, and $\sqsubseteq^\diamond$, e.g., $(a, u) \sqsubseteq^\diamond (a', u') \Leftrightarrow a \sqsubseteq^\sharp a' \wedge u \sqsubseteq^\flat u'$.

$$\frac{[\textsf{if } B \textsf{ jmp } E]^{\ell}_{\ell'} \quad \chi(S,E) \quad (1,V) \in \gamma^\sharp[\![B,E]\!]S^\sharp \quad f^\diamond[\![\textsf{assume } B \wedge E = V]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle V, S', G \uplus (\ell, V)\rangle}\ \text{JT}^\sharp$$

$$\frac{[\textsf{if } B \textsf{ jmp } E]^{\ell}_{\ell'} \quad \neg\chi(S,E) \quad (1,V) \in \gamma^\flat[\![B,E]\!]S^\flat \quad f^\diamond[\![\textsf{assume } B \wedge E = V]\!](S) = S'}{\langle \ell, S, G\rangle \longrightarrow \langle V, S', G \uplus (\ell, V)\rangle}\ \text{JT}^\flat$$

Fig. 3. Split rules for alternating jump target resolution. $\chi(S,E)$ controls which rule is enabled, with $S = (S^\sharp, S^\flat)$.

- The initial lattice element $\iota^\diamond = (\iota^\sharp, \iota^\flat)$.
- A combined transfer function $f^\diamond[\![C]\!](S^\sharp, S^\flat) := (f^\sharp[\![C]\!](S^\sharp), f^\flat[\![C]\!](S^\flat))$. The combined transfer function maps over- and under-approximate states to their respective successors. If the under-approximate state has no successor for a particular state (i.e., the condition of an assume statement is false), the under-approximate component is set to $\bot^\flat$.
- The concretizing evaluation function $\gamma^\diamond[\![\cdot]\!](S^\sharp, S^\flat) := \gamma^\sharp[\![\cdot]\!](S^\sharp)$, which ignores the under-approximate part of the state.
- The merging operator $\cup^\diamond$ such that $(S_1^\sharp, S_1^\flat) \cup^\diamond (S_2^\sharp, S_2^\flat) := (S_1^\sharp \cup^\sharp S_2^\sharp, S_1^\flat \cup^\flat S_2^\flat)$.

This combined semantics can instantiate our control flow reconstruction framework, but will yield the same CFG as the over-approximation alone, as the concretizing evaluation function will yield the same over-approximate branch targets when applying the JUMP-TRUE rule. We need to adapt our framework to make use of the additional under-approximate information.

5.2 Alternation Framework

To allow switching between over- and under-approximate information when resolving indirect jumps, we split the JMP-TRUE rule of our framework into the two rules shown in Figure 3. A definable predicate $\chi(S,E)$ for a state S and an expression E controls which of the two rules is enabled. When $\chi(S,E)$ is true, the rule $\text{JT}^\sharp$ resolves jumps exactly as the original JMP-TRUE rule before. Otherwise, $\text{JT}^\flat$ is enabled and uses the under-approximate portion of the state to evaluate the jump condition and the target expression.

The definition of χ allows to fine-tune the alternation. For instance, it can be defined as $\chi((S^\sharp, S^\flat), E) := \gamma^\sharp[\![E]\!]S^\sharp \subset \mathbb{N}$, such that the over-approximation is used as long as the concretizing evaluation function returns a finite set. We will use this definition of χ for the experiments in Section 6. Another possibility is to use $\chi'((S^\sharp, S^\flat), E) := S^\flat \sqsupset^\flat \bot^\flat \vee \gamma^\sharp[\![E]\!]S^\sharp \subset \mathbb{N}$, which only falls back to under-approximation if the under-approximation contains information at the location of the jump. By setting χ to always true or false, the alternation framework falls back to pure over- or under-approximation, respectively.

Note that the under-approximation can often default to the bottom element $\bot^\flat$. In the trace replay semantics defined in Section 4.3, this happens when the executions do

not cover a statically feasible control flow edge (e.g., the second branch of a conditional jump). Then $f^\flat$ yields the empty set (the $\bot^\flat$ element for the trace replay semantics), which is paired with the over-approximate state for that location. During fixpoint computation, this empty set propagates until it merges with under-approximate states from a covered path.

5.3 Algorithm

As for the other semantics, the combined analysis is performed by computing the least fixpoint by chaotic iteration. Even though our alternating semantics is a considerable detour from regular abstract interpretation, the transfer relation is monotonic, and a fixpoint exists for the full framework instance.

Theorem 1 (Termination). *The alternating control flow reconstruction algorithm terminates in finite time if the over- and under-approximate domains do not contain infinite ascending chains.*

Proof. As $f^\flat$ and $f^\sharp$ are required to be monotonic, $f^\diamond$ is monotonic, i.e., $S_1 \sqsubseteq^\diamond S_2 \Leftrightarrow f^\diamond(S_1) \sqsubseteq^\diamond f^\diamond(S_2)$. The transfer relation $\longrightarrow$ of the parameterized semantics is monotonic in its original form [15], i.e., $\forall S_1, S_1', S_2, S_2' : S_1 \sqsubseteq S_2 \wedge S_1 \longrightarrow S_1' \wedge S_2 \longrightarrow S_2' \Rightarrow S_1' \sqsubseteq S_2'$. It remains to prove that it stays monotonic when JUMP-TRUE is replaced by the two new rules. The critical part is the situation when $\chi(S_1, E) \wedge \neg\chi(S_2, E)$ or $\neg\chi(S_1, E) \wedge \chi(S_2, E)$ holds and $\longrightarrow$ switches between the two rules.

The only difference between the rules, however, is the more restricted concretization of jump targets in $\text{JT}^\flat$, since $|\gamma^\flat[\![B, E]\!]S^\flat| \leq |\gamma^\sharp[\![B, E]\!]S^\sharp|$. Therefore, $\text{JT}^\flat$ may fire fewer times than $\text{JT}^\sharp$, but the individual applications remain monotonic. Obviously, firing strictly more times does not affect the monotonicity of the semantic function F (see Section 3.3). Neither does firing fewer times, because (i) all remaining state updates are still guaranteed to be monotonic by virtue of $f^\diamond$'s monotonicity and (ii) no edges already created are removed from the partial control flow graph G. □

Again, the restriction to finite ascending chains in the over-approximate lattice can be lifted by introducing widening. In under-approximations, infinite ascending chains are easily avoided using counters (see Section 4.3).

5.4 Characterizing the Reconstructed CFG

Soundness. The transfer function of our alternating semantics neither over- nor under-approximates the concrete semantics, i.e., it is not a sound abstraction. After chaotic iteration reaches a fixpoint, the "over-approximate" part $S^\sharp$ of the combined state $S^\diamond = (S^\sharp, S^\flat)$ may not fully over-approximate the concrete reachability semantics. Consequently, the final CFG is neither an over- nor an under-approximation of the concrete CFG for the full program.

However, we can still characterize the kind of approximation that is performed: Alternating control flow reconstruction computes an over-approximation of the concrete CFG of a restricted version of its input program. The restriction asserts that at each

indirect jump where the under-approximation was used, control flow follows only the under-approximate set of branches. We define the restricted program as $\langle \ell'_0, P' \rangle$ with $P' = P[\ell'_0 \mapsto [\mathsf{assume}\ \phi]_{\ell_0}^{\ell'_0}]$ enforcing a precondition ϕ. Here, ϕ is the condition under which $\neg\chi((S^\sharp, S^\flat), E) \implies \gamma^\flat[\![B, E]\!]S^\flat = \gamma^\natural[\![B, E]\!]S^\natural$ at each state $\langle \ell, (S^\sharp, S^\flat), G \rangle$ with $[\mathsf{if}\ B\ \mathsf{jmp}\ E]_{\ell'}^{\ell}$. Then, we can state the following theorem:

Theorem 2 (Relative Soundness). *The CFG computed from the alternating semantics for a program $\langle \ell_0, P \rangle$ soundly over-approximates the concrete CFG of the restricted program version $\langle \ell'_0, P' \rangle$.*

Proof. All rules except JT$^\flat$ over-approximate G. For all applications of JT$^\flat$, ϕ enforces that $\gamma^\flat[\![B, E]\!]S^\flat = \gamma^\natural[\![B, E]\!]S^\natural$, which is a (trivial) over-approximation. □

If needed, the precondition ϕ could in principle be determined by backward symbolic execution along the paths leading to the jumps. When using directed test generation, ϕ is the disjunction of the input assignments of the test cases leading to each indirect jump. Note that when an indirect jump is not covered by a test case, ϕ asserts that the jump cannot be reached. A static analysis performed using the reconstructed CFG as program representation will in fact be performed on $\langle \ell'_0, P' \rangle$, thus proofs obtained this way hold for $\langle \ell, P \rangle$ only under the assumption ϕ.

Precision. Note that the fixpoint of the alternating semantics computed by chaotic iteration can depend on the iteration order. Even though the transfer relation "$\longrightarrow$" is monotonic, it does not distribute over the merge operator. That is, merging two states $\langle \ell, S_1, G_1 \rangle \dot{\cup} \langle \ell, S_2, G_2 \rangle = \langle \ell, S_{1,2}, G_{1,2} \rangle$ and computing the set of successors $\mathbf{T}'_{1,2} = \{\langle \ell', S'_{1,2}, G'_{1,2} \rangle \mid \langle \ell, S_{1,2}, G_{1,2} \rangle \longrightarrow \langle \ell', S'_{1,2}, G'_{1,2} \rangle\}$ can lead to a different result than first computing only, say, the successors $\mathbf{T}'_1$ of $\langle \ell_1, S_1, G_1 \rangle$ and then merging them with $\mathbf{T}'_{1,2}$. This happens if $\chi(S_1, E)$ but $\neg\chi(S_{1,2}, E)$ holds for a jump at ℓ, i.e., rule JT$^\sharp$ is applied for S_1 but rule JT$^\flat$ for $S_{1,2}$.

The *least* fixpoint of the alternating semantics can yield a CFG containing more edges than the one obtainable by applying the JT$^\flat$ rule from the beginning wherever it eventually becomes enabled, because this could avoid some over-approximate edges that are added while still using JT$^\sharp$. In principle, this "optimal" CFG could be computed by maintaining a list of locations at which to apply JT$^\flat$ and, each time a location is added to the list, resetting the analysis to the point where that location was reached first. In practice, however, precise target addresses initially resolved using JT$^\sharp$ are likely to be concrete. For instance, these addresses can be constants that were propagated along a particular path before a $\top$ element from another path overwrote them. Such addresses add concrete edges that could be missed by the under-approximation and can therefore *improve* the precision with respect to the original program.

6 Evaluation

We now report some preliminary results for reconstructing the control flow graphs of microbenchmarks and medium-sized compiled applications.

6.1 Implementation and Setup

We evaluated the proposed approach on our static analysis platform for x86 binaries, JAKSTAB [13].[2] We extended JAKSTAB with a combined adapter analysis modeled after the semantics described in Section 5.1 and implemented a new analysis module that replays pre-recorded execution traces. With the combined analysis, JAKSTAB resorts to under-approximation whenever the static analysis cannot produce a finite concretization for the set of target addresses of an indirect jump.

As analysis targets we used our own microbenchmarks and several C and C++ programs from the SPEC CPU 2006 benchmark, compiled using Visual C++. The benchmarks are single-threaded, so we only had to record and replay the main thread. Our microbenchmarks are hand-written assembly language programs, so we were able to manually construct the concrete CFG as the ground truth. For establishing a ground truth for the C/C++ programs, we enabled debug symbols and disassembled them using IDA PRO [11]. IDA PRO uses the debug symbols and several compiler-specific heuristics to create an accurate assembly language representation. Note that this does not necessarily reflect the concrete CFG, however. Due to its heuristic approach, IDA PRO disassembles *all* identifiable code in an executable and not only that which is actually reachable. This includes a large amount of library and error handling code. Therefore, the number of instructions explored by a non-heuristic approach is expected to be significantly lower.

We recorded concrete execution traces on a single-processor 32-bit Windows XP guest system running in the BitBlaze [17] version of QEMU. As soon as the target process is started, all user-mode instructions (including libraries) are recorded to a file. The SPEC benchmark is designed for long running CPU time evaluations and can yield instruction traces of hundreds of gigabytes. To obtain traces of tractable length while still exercising the same code, we had to remove redundancy from the benchmarking setup. We decreased the size of explicit input data, and if necessary, also reduced bounds of loops that repeatedly invoke the same program functionality. We recorded a single trace per executable.

6.2 Experimental Results

For determining the effectiveness of our approach, we measured the instruction coverage achieved by JAKSTAB using under-approximate, over-approximate, and alternating control flow reconstruction. In this setting, false positives refer to unreachable instructions that are wrongly classified as reachable, and false negatives refer to reachable instructions that have not been covered. We replayed a single trace for the under-approximation and used simple constant propagation for the over-approximation, as it scales to large binaries. The preliminary results of our experiments are shown in Table 1. For each program, we list its implementation language, the number of instructions we accepted as ground truth, and the length of the trace we recorded.

The ♭ column contains the results from enabling only trace replay, i.e., the reached instructions as an absolute number and as percentage of the ground truth, and the required execution time of the analysis. Since the analysis is under-approximate, there

[2] Source code available at `http://www.jakstab.org`.

				♭ only			♯ only			◇			
Program	Src	Inst	Trace	Inst	Cvrg	Time	Inst	FP	Time	Inst	Cvrg	FP	Time
demo1	asm	9	49K	7	78%	<1s	512	98%	1.5s	9	100%	0%	<1s
demo2	asm	38	87K	29	76%	<1s	512	93%	1.4s	35	92%	0%	<1s
demo3	asm	35	87K	17	49%	<1s	512	93%	2.2s	35	100%	0%	<1s
astar	C++	29645	1.0M	10738	36%	7.2s	*		*	15377	52%	0%	18.3s
bzip	C	29257	531K	9660	33%	5.3s	*		*	20453	70%	0.4%	33.2s
lbm	C	5057	840K	2943	58%	2.2s	*		*	4603	91%	0%	3.5s
omnetpp	C++	171592	4.6M	30627	18%	27.1s	*		*	61461	36%	0.05%	153.9s
milc	C	47382	12.0M	14085	30%	13.4s	*		*	24546	52%	0.3%	54.2s
specrand	C	16937	413K	4720	28%	2.9s	*		*	9166	54%	0%	8.1s

Table 1. Experimental results for control flow reconstruction. ♭ denotes under-approximation, ♯ over-approximation, and ◇ alternating control flow reconstruction. * denotes an out of memory error. Under-approximations follow a single trace.

are no false positives, only false negatives (i.e., uncovered instructions). The ♯ column contains the number of instructions reported by constant propagation when unresolved indirect jumps are truly over-approximated. For all programs but the microbenchmarks, the large number of spurious edges caused the analysis to run out of memory after using 2 GB of heap space. In the microbenchmarks, the number of 512 instructions results from padding in the code section. Since the analysis is over-approximate, there are no false negatives. The "FP" column shows the fraction of false positives in the total instructions discovered (i.e., unreachable instructions included in the CFG). The ◇ column shows the number of instructions reported by alternating trace replay and constant propagation. It is susceptible to both false positives and negatives. The "Cvrg" column shows the coverage in percent, and the "FP" column shows the percentage of false positives in the total instructions reported.

We can see that using alternation improves coverage over purely under-approximate reconstruction in all cases, and that it drastically reduces the number of false positives compared to over-approximation. In fact, the number of false positives in over-approximation is prohibitive for realistic programs, therefore alternation is essential for applying control flow reconstruction in practice. We expect that these results can be significantly improved by either using more precise static analyses or directed test generation to generate additional traces.

6.3 Current Limitations and Future Work

Automated Trace Generation. Alternating control flow reconstruction could be effectively interleaved with automated test case generation directed at the locations of statically unresolved indirect jumps that have not yet been covered by a trace. The method for obtaining suitable test cases and thus traces is orthogonal to our approach. Indeed, the framework presented here could be combined with random testing, whitebox fuzz testing [10,17], or binary symbolic execution [6].

Divergences. If the over-approximate semantics is a sound abstraction of the concrete semantics, under-approximate states will always concretize to a subset of the concretization of the corresponding over-approximate states. In practice, however, unsoundness can be introduced by abstracting library calls or not handling parts of the instruction set. This can cause the over-approximation to diverge and not include the under-approximation. In our implementation, we give precedence to the concretization of the over-approximation in such cases to keep it consistent.

Concurrency. Our approach in its current form is limited to sequential programs. A simple way to apply it to multithreaded programs is to treat each thread separately, which can lead to unsoundness and possibly divergences from ignoring the actions of the other threads. A better solution would be to allow context switches in the under-approximation, e.g., by replaying one or more traces containing multiple threads arranged in a total order. To avoid state explosion, the over-approximation then only considers the schedules present in the under-approximation. We leave the implementation of such a strategy for future work.

7 Related Work

Several approaches use runtime control flow information to improve the results of analyzing binaries. Nanda et al. introduced *hybrid disassembly* in their tool BIRD [16]. They first use a heuristic disassembly algorithm to identify likely code regions in an executable. The code is then instrumented to check whether control transfers to a previously unexplored region, in which case the disassembler is invoked again. They rely on several heuristics for avoiding excessive instrumentation, and do not give a formal justification for their approach. In general, the exploration of code areas not covered at runtime suffers from the same limitations as regular heuristic disassembly strategies.

In recent work, Babic et al. [1] construct a CFG by folding a set of concrete traces and exploring the unexecuted branches of conditional jumps. In our own framework, this corresponds to using trace replay with a trivial static analysis that only knows a single $\top$ state and thus always enables both branches of conditional jumps. They do not provide a formal treatment of the reconstruction, only stating that the CFG "is, necessarily, an under-approximation of the complete interprocedural CFG". In fact, this statement is incorrect with respect to our definition of the concrete CFG, since there can be conditional jumps with only a single feasible branch.

Directed proof generation [4] combines over- and under-approximation in the form of predicate abstraction and directed test generation. Thakur et al. [18] lift this approach to binaries in their McVeto tool. As part of the analysis, the CFG is built through folding traces and connecting not yet fully explored branching points to the target surrogate, a special unknown node. The over-approximate data flow analysis is only performed on the folded trace. Therefore, the CFG can only be expanded by generating new test cases, which requires exhaustive exploration to achieve a complete CFG. In contrast, our method is able to perform over-approximation reconstruction of CFG parts not covered by traces. Their method would lend itself to generating traces towards statically unresolvable jumps in our framework, however.

Most work addressing disassembly relies on the use of heuristics to identify likely code regions [12,2,19]. Vigna [19] describes how to defeat anti-disassembly obfuscations by starting disassembly from every possible program location. In our framework, this amounts to performing purely over-approximate control flow reconstruction with again a trivial analysis that knows only one single $\top$ state.

Systematic, purely static approaches to control flow reconstruction from binaries have been scarce. De Sutter et al [8] described how to use data flow analysis in building a CFG, using unknown nodes for indirect jumps. Chang et al. [5] connect abstract interpreters at different language levels to achieve a form of decompilation. Bardin et al. [3] build on our own framework [15] to introduce iterative abstraction refinement.

8 Conclusion

This paper addresses a major weakness in static control flow reconstruction that arises when the possible targets of an indirect jump cannot be resolved precisely. The new reconstruction framework alternates between over- and under-approximation and skips over unresolvable jumps by substituting targets detected by an under-approximation.

Our framework is formalized as an abstract interpretation of a parameterized adapter semantics. It establishes a formal common ground for several approaches in control flow reconstruction from binaries, an area which has traditionally been dominated by ad-hoc solutions. By proving termination and relative soundness for the framework, we hope that this paper contributes to better understanding of static and dynamic approaches to the analysis of machine code.

Acknowledgments

We would like to thank Volodymyr Kuznetsov, Péter Bokor, and the anonymous reviewers for their insightful comments that greatly helped to improve the paper. The second author is supported by CASED (www.cased.de).

References

1. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proc. Int. Conf. Soft. Testing and Analysis (ISSTA 2011). ACM (2011)
2. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Proc. 13th Int. Conf. Compiler Construction (CC 2004). LNCS, vol. 2985, pp. 5–23. Springer (2004)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from unstructured programs. In: Proc. 12th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2011). LNCS, vol. 6538, pp. 54–69. Springer (2011)
4. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. ACM/SIGSOFT Int. Symp. Soft. Testing and Analysis (ISSTA 2008). pp. 3–14. ACM (2008)
5. Chang, B., Harren, M., Necula, G.: Analysis of low-level code using cooperating decompilers. In: Yi, K. (ed.) 13th Int. Static Analysis Symp. (SAS 2006). LNCS, vol. 4134, pp. 318–335. Springer (2006)

6. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proc. 16th. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011). pp. 265–278. ACM (2011)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Rec. 4th ACM Symp. Principles of Programming Languages (POPL 1977). pp. 238–252 (Jan 1977)
8. 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)
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proc. ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation (PLDI 2005). pp. 213–223. ACM (2005)
10. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. Network and Distributed System Security Symp. (NDSS 2008). The Internet Society (2008)
11. Hex-Rays SA.: IDA Pro, `http://www.hex-rays.com/idapro/`
12. 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'02-SCOPES'02). pp. 46–55. ACM (2002)
13. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Proc. 20th Int. Conf. Computer Aided Verification (CAV 2008). LNCS, vol. 5123, pp. 423–427. Springer (2008)
14. Kinder, J., Veith, H.: Precise static analysis of untrusted driver binaries. In: Proc. 10th Int. Conf. Formal Methods in Computer-Aided Design (FMCAD 2010). pp. 43–50. FMCAD, Inc. (2010)
15. Kinder, J., Veith, H., Zuleger, F.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Proc. 10th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI 2009). LNCS, vol. 5403, pp. 214–228. Springer (2009)
16. Nanda, S., Li, W., Lam, L., Chiueh, T.: BIRD: Binary interpretation using runtime disassembly. In: 4th IEEE/ACM Int. Symp. Code Generation and Optimization (CGO 2006). pp. 358–370. IEEE Computer Society (2006)
17. Song, D.X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proc. 4th Int. Conf. Information Systems Security (ICISS 2008). LNCS, vol. 5352, pp. 1–25. Springer (2008)
18. Thakur, A.V., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.W.: Directed proof generation for machine code. In: Proc. 22nd Int. Conf. Computer Aided Verification (CAV 2010). LNCS, vol. 6174, pp. 288–305. Springer (2010)
19. Vigna, G.: Static disassembly and code analysis. In: Christodorescu, M., Jha, S., Maughan, D., Song, D.X., Wang, C. (eds.) Malware Detection, Advances in Information Security, vol. 27, chap. 2, pp. 19–41. Springer (2007)