Partial Evaluation of Machine Code *

Venkatesh Srinivasan
University of Wisconsin–Madison, USA
venk@cs.wisc.edu

Thomas Reps
University of Wisconsin–Madison
and GrammaTech, Inc., USA
reps@cs.wisc.edu

Abstract

This paper presents an algorithm for off-line partial evaluation of machine code. The algorithm follows the classical two-phase approach of binding-time analysis (BTA) followed by specialization. However, machine-code partial evaluation presents a number of new challenges, and it was necessary to devise new techniques for use in each phase.

- Our BTA algorithm makes use of an instruction-rewriting method that "decouples" multiple updates performed by a single instruction. This method counters the cascading imprecision that would otherwise occur with a more naïve approach to BTA.
- Our specializer specializes an explicit representation of the semantics of an instruction, and emits residual code via machine-code synthesis. Moreover, to create code that allows the stack and heap to be at different positions at run-time than at specialization-time, the specializer represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization.

Our experiments show that our algorithm can be used to specialize binaries with respect to commonly used inputs to produce faster binaries, as well as to extract an executable component from a bloated binary.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Partial evaluation

Keywords Partial evaluation, machine code, BTA, specialization, machine-code synthesis, IA-32 instruction set

* Supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by NSF under grant CCF-0904371; by AFRL under DARPA CRASH award FA8650-10-C-7088, DARPA MUSE award FA8750-14-2-0270, and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

1. Introduction

The analysis and rewriting of binaries has gotten an increasing amount of attention from the academic community in the last decade (e.g., see references in [44, §7], [9, §1], [13, §1], [18, §7]). One of the potential applications of machine-code[1] analysis and rewriting is to facilitate reuse of software binaries in the absence of source code. In particular, it would be desirable to have a binary-rewriting tool that can (i) optimize a binary for the common case (e.g., a file-write routine optimized for a certain file descriptor), or (ii) extract an executable component from a bloated binary (e.g., a small text encoder embedded in a web server).

Partial evaluation [25] is a program-specialization framework that can perform the aforementioned tasks. However, there are no tools that partially evaluate machine code. Existing binary-rewriting tools either de-obfuscate [16, 42, 48], superoptimize [11, 39], or harden [4, 20, 43] binaries, but do not perform partial evaluation. Moreover, machine-code partial evaluation presents a number of new challenges, and source-code partial-evaluation algorithms [6, 15, 26] would not produce satisfactory results if they were applied to machine code in a straightforward manner (see §3).

This paper presents an algorithm for off-line partial evaluation of machine code. The inputs to the algorithm are a binary B, and a division of B's inputs as S (*static inputs*) and D (*dynamic inputs*). Values for static inputs are known at specialization time; values for dynamic inputs are unknown at specialization time. Our partial-evaluation algorithm produces a binary B_S that satisfies the equation

$$[\![B]\!](S, D) = [\![B_S]\!](D),$$

where $[\![\cdot]\!]$ denotes the meaning function for the instruction set in which the binary is written. Executing B_S with input D produces the same results as executing B with inputs S and D. However, B_S is specialized with respect to S, and can be significantly faster than B.

Partially evaluating a binary with respect to commonly used inputs produces specialized versions of the binary that are optimized for the common inputs. (See §6.1.) Partially

[1] We use the term "machine code" to refer generically to low-level code, and do not distinguish between actual machine-code bits/bytes and assembly code to which it is disassembled.

evaluating with respect to a fixed value of an input flag can extract a smaller executable component from a bloated binary. (See §6.2.) The extracted component can be used in embedded systems where executable size matters, or be linked against other programs that require the component. (See the experiment with `lzfx` in §6.2.)

We have implemented our algorithm in a tool, called WIPER (an acronym for the "Wisconsin Partial Evaluator"), that partially evaluates Intel IA-32 binaries. WIPER works on the Intermediate Representations (IRs) recovered from machine code by an existing tool, CodeSurfer/x86 [10]. WIPER follows the classical two-phase approach of *binding-time analysis* (BTA) followed by *specialization*. WIPER's BTA uses *forward slicing* to determine a priori which instructions can be specialized away (static instructions), and which cannot (dynamic instructions). Given the values for static inputs in the form of a partial state, WIPER's specializer evaluates static instructions, residuates code for static values that might be used by dynamic instructions, and emits unmodified dynamic instructions.

As mentioned earlier, there are several new challenges that arise in the context of machine-code partial evaluation. These issues, along with the strategies that WIPER uses to overcome them, distinguish WIPER from other partial evaluators that have been described in the literature.

- Machine-code instructions are usually *multi-assignments*: they have several inputs, and several outputs (e.g., registers, flags, and memory locations). This aspect of the language introduces a *granularity* issue during slicing: in some cases, although we would like the slice to follow only a subset of an instruction's semantics, the slicing algorithm is forced to include the entire instruction. This effect can cascade, and cause BTA to be very imprecise. The BTA algorithm in WIPER makes use of an instruction-rewriting method that *decouples* multiple updates performed by a single instruction by splitting the assignments across multiple instructions. The decoupling transformation increases the precision of WIPER's BTA significantly.
- An instruction's abstract-syntax tree (AST) is often not parameterized by all of the operands of the instruction: some operands are "baked into" the semantics. In contrast with source-code partial evaluators that specialize the AST of a dynamic statement, WIPER specializes an explicit representation of the *semantics* of a dynamic instruction, and uses a machine-code synthesizer [46] to produce residual code.
- In machine code, values in memory are accessed by explicit address computations, followed by loading. The residual code produced by a naïve machine-code partial evaluator will contain specialization-time addresses, and consequently would not allow the stack and heap to be at different positions at run-time than at specialization-time. WIPER represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. The resulting code produced by WIPER is independent of the layout of the specialization-time address space, and can be run with, e.g., the stack base at a different address.[2]

The contributions of our work include the following:

- We present an algorithm for machine-code partial evaluation. To the best of our knowledge, WIPER is the first partial evaluator that works on machine code.[3]
- We have developed an instruction-decoupling transformation to counter cascading imprecision caused by the granularity issue in machine-code slicing.
- We have introduced an instruction-specialization technique that specializes an explicit representation of the semantics of an instruction with respect to a partial state, and residuates code via machine-code synthesis.
- We have developed a specialization algorithm that represents specialization-time addresses using symbolic constants, and uses a symbolic state for specialization. The residual code produced by our specializer is independent of the specialization-time memory layout, and allows the stack and heap to be at different positions at run-time than at specialization-time.

Our methods have been implemented in WIPER, a partial evaluator for Intel IA-32. We partially evaluated seven test applications using WIPER, and found that, on average (computed via geometric mean), the specialized binaries have a speedup of 1.3. We also used WIPER to extract executable components from bloated binaries such as `bzip2`.

2. Background

In this section, we briefly describe source-code partial evaluation, slicing, syntax and semantics of the IA-32 instruction-set architecture (ISA), and a logic to express the semantics of IA-32 instructions.

2.1 Partial Evaluation

Partial evaluation is a framework for specializing and optimizing programs [25]. Given a program that takes some inputs, and given values for some of the inputs, the goal of partial evaluation is to produce a program that is specialized (optimized) with respect to the input values provided. Fig. 1 shows a program power that computes $(a+b)^n$, where a, b, and n are the inputs to the program. Suppose that power is frequently called with the values $b = 1$, and $n = 4$. Given power, and the input values $b = 1$ and $n = 4$, a partial evaluator produces the residual program power_R shown in Fig. 2. As we can see in power_R, partial evaluation has compiled data ($n = 4$) into control. (See Eqn. (1).) In effect, partial evaluation performs optimizations such as loop unrolling,

[2] Note that we are referring here to the ability to reposition the stack at the beginning of run-time, not the relocatability of the residual code per se. The residual code produced WIPER is also relocatable.

[3] Confirmed by personal communication with Neil Jones, Robert Glück, and Saumya Debray.

```
int power(int a,
int b, int n) {
1: int prod = 1;
2: while (n--) {
3:   prod *= (a + b);
4: }
5: return prod;
}
```

Figure 1: Input power program. Computes $(a+b)^n$.

```
int power_r(int a) {
   int prod = 1;
   prod *= (a + 1);
   prod *= (a + 1);
   prod *= (a + 1);
   prod *= (a + 1);
   return prod;
}
```

Figure 2: Residual power_R program. Computes $(a+1)^4$.

```
int foo() {
1: int a = 1, b = 2;
2: int c, d, e;
3: scanf("%d", &c);
4: d = add(a, b);
5: e = add(c, c);
6: return d + e;
}
```

Figure 3: Example program foo to illustrate lifting.

```
int foo_r() {
1: int c, d, e;
2: scanf("%d", &c);
3: d = add(1, 2);
4: e = add(c, c);
5: return d + e;
}
```

Figure 4: Residual foo_r program.

constant propagation and folding, function in-lining, etc. to produce the residual program—although a partial evaluator does not have such optimizations built into it *per se*. In this section, and in the rest of this paper, when we say "source-code partial evaluator," we refer to a partial evaluator for an imperative language, such as C [6].

An off-line source-code partial evaluator works in two phases: *binding-time analysis* and *specialization*.

The input to BTA is a division (or classification) of the input variables as *static inputs* and *dynamic inputs*. Given a division of the input variables, the goal of BTA is to extend the division to all program variables—i.e., classify each occurrence of a variable as either static or dynamic. A division computed by BTA must be *congruent*, i.e., any variable that depends on a dynamic variable must be classified dynamic. Using the computed division, expressions and statements are classified as either static or dynamic [5]. For the power program, BTA classifies variables `a` and `prod`, and statements 1, 3, and 5 as dynamic.

Given values for static inputs, in the form of a partial static-state σ, the specializer specializes the input program with respect to σ, and produces the residual program. A source-code specializer uses the following approach:

Approach $\mathbf{PE}_S$

- Evaluate static expressions and statements, and
- Specialize the AST of dynamic expressions and statements with respect to σ by substituting values from σ for variables, simplifying, and emitting the resulting AST.

The second step is sometimes called *residuation*.

For the power program, σ is $[n \mapsto 4, b \mapsto 1]$. Because the loop condition (Line 2 in Fig. 1) is static, the specializer unrolls the loop four times. The specializer residuates the expression `a + 1` by specializing the dynamic expression `a + b` with respect to σ.

Sometimes a static expression could be used in a dynamic context. For example, consider the program foo given in Fig. 3. Because `c`'s value is established dynamically in line `3`, both of the formal parameters of `add` are classified dynamic by BTA. Although both actual parameters of the call to `add` in line `4` of Fig. 3 are static, they appear in a dynamic context—the static actuals will be bound to dynamic formals in `add`. To produce a residual program that compiles and/or does not access uninitialized locations, the partial evaluator must residuate code for the static actuals. Thus the partial evaluator "lifts" static expressions that appear in dynamic contexts, and the specializer residuates code for lifted expressions (e.g., line `3` in Fig. 4). The term "lifting" refers to the process of changing the binding time of an expression from static to dynamic.

Suppose that each statement in the original program is uniquely identified by a label *l*, and that each concrete state σ that arises during an execution is partitioned into σ_S and σ_D according to the division established by BTA. Then an execution trace of the original program is a sequence of elements of the form $l : (\sigma_S, \sigma_D)$. (Each element of the trace records the state (σ_S, σ_D) observed at an execution of a statement *l*.) The effect of partial evaluation on an execution trace of the specialized program can be expressed by the following reparenthesization:

$$l : (\sigma_S, \sigma_D) \longrightarrow (l, \sigma_S) : \sigma_D \tag{1}$$

Operationally, the partial evaluator creates a new residual statement for every unique (l, σ_S) pair it observes at specialization time. Eqn. (1) shows how each element $(l, \sigma_S) : \sigma_D$ in an execution trace of the specialized program can be mapped back to the corresponding element $l : (\sigma_S, \sigma_D)$ in the trace of the original program.

2.2 Slicing

Slicing [24, 47] computes the set of program points that affect (or are affected by) a given program point called the *slicing criterion*. (*Backward slicing* computes the set of program points that might affect the slicing criterion; *forward slicing* computes the set of program points that might be affected by the slicing criterion.) Slicing is typically performed using an IR called a *system dependence graph* (SDG) [21, 24]. An SDG consists of a set of *program dependence graphs* (PDGs), one for each procedure in the program. A node in a PDG corresponds to a construct in the program, such as a statement, a condition, a call to a procedure, a procedure entry/exit, an actual parameter of a call, or a formal parameter of a procedure. The edges correspond to data and control dependences between the nodes [21]. The PDGs are connected together with interprocedural control-dependence edges between call-site nodes and procedure-entry nodes, and interprocedural data-dependence edges between actual parameters and formal parameters/return values. Given an SDG representation of the program and a slicing criterion, an interprocedural-slicing algorithm includes in the slice the

$$\text{INT} \in \text{Integer},\ \text{Reg} \in \text{Register},\ \text{I} \in \text{Instruction},\ \text{O} \in \text{Operand},$$
$$\text{R} \in \text{RegisterOperand},\ \text{C} \in \text{ImmediateOperand},$$
$$\text{M} \in \text{IndirectOperand},\ \text{RC} \in \text{RegisterImmediateOperand}$$
$$\text{INT} ::= \{\ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots\} \quad \text{Reg} ::= \texttt{eax} \mid \texttt{ebx} \mid \texttt{esp} \mid \ldots$$
$$\text{R} ::= \text{Direct}(\text{Reg}) \quad \text{C} ::= \text{Imm}(\text{INT}) \quad \text{RC} ::= \text{R} \mid \text{C}$$
$$\text{M} ::= \text{Indirect}(\text{R}, \text{R}, \text{INT}, \text{INT}) \quad \text{O} ::= \text{RC} \mid \text{M}$$
$$\text{I} ::= \texttt{push}(\text{O}) \mid \texttt{mov}(\text{R}, \text{O}) \mid \texttt{mov}(\text{M}, \text{RC}) \mid \texttt{lea}(\text{R}, \text{M}) \mid$$
$$\texttt{add}(\text{R}, \text{O}) \mid \texttt{add}(\text{M}, \text{RC}) \mid \texttt{sub}(\text{R}, \text{O}) \mid \texttt{sub}(\text{M}, \text{RC}) \mid$$
$$\texttt{cmp}(\text{M}, \text{C}) \mid \texttt{jz}(\text{C}) \mid \texttt{jmp}(\text{C})$$

Figure 5: Abstract syntax for a subset of IA-32.

SDG nodes that reach (or can be reached from) the slicing criterion by following data- and control-dependence edges. A context-sensitive interprocedural-slicing algorithm uses context-free language (CFL) reachability [35] to reduce the number of nodes in the slice [24, 36].

2.3 Syntax and Semantics of IA-32

Syntax. The abstract syntax for a subset of IA-32 instructions that will be used in this paper is given in Fig. 5. Indirect operands are of the form $\langle$*base*, *index*, *scale*, *offset*$\rangle$, where *base* and *index* are registers, and *scale* and *offset* are integers. The effective address is computed as $base + index * scale + offset$. In instructions with two operands, the operand on the left is the destination (except the `cmp` instruction, in which both operands are source operands).

Semantics. The primitive domains of the IA-32 semantics include 32-bit integers, Booleans, registers, and flags. The primitive domains and their operators are given below.

$$i \in \mathit{INT} = \mathbb{Z}_{32} \qquad b \in \mathit{BOOL} = \{\mathit{True}, \mathit{False}\}$$
$$r \in \mathit{Register} = \{\mathit{EAX}, \mathit{ESP}, \ldots\} \quad f \in \mathit{Flag} = \{\mathit{CF}, \mathit{SF}, \ldots\}$$
$$op \in \mathit{ArithOp} = \{+, -, \ldots\} \quad bop \in \mathit{BoolOp} = \{\wedge, \vee, \ldots\}$$
$$rop \in \mathit{RelOp} = \{=, \neq, <, \ldots\} \quad cop \in \mathit{CondOp} = \{?\ :\}$$

Store is a compound domain denoting an IA-32 store. *Store* is a triple consisting of three maps: a register map, a flag map, and a memory map. *Store* has operators to access and update the maps. The *Store* domain is defined below.

$$\rho \in \mathit{Store} = \mathit{RegMap} \times \mathit{FlagMap} \times \mathit{MemMap}$$
$$\mathit{RegMap} : \mathit{Register} \to \mathit{INT}$$
$$\mathit{FlagMap} : \mathit{Flag} \to \mathit{BOOL} \qquad \mathit{MemMap} : \mathit{INT} \to \mathit{INT}$$

The valuation function $\mathcal{I}$ has the type $\mathcal{I} : \text{Instruction} \to \mathit{Store} \to \mathit{Store}$. $\mathcal{I}$ takes an instruction I and a pre-store, and returns a post-store that reflects the updates made by the execution of I. For example, the valuation function for the "`mov eax,[ebp]`" instruction is given below. The instruction copies a 32-bit value from the memory location pointed to by the frame-pointer register *EBP* to the *EAX* register. The overloaded function $[\![\cdot]\!]$ returns primitive semantic objects for their syntactic counterparts. The instruction also increments the program counter *EIP* by the length of the instruction. For brevity, we do not show this increment explicitly.

$$\boldsymbol{T} \in \textbf{\textit{Term}},\ \boldsymbol{\varphi} \in \textbf{\textit{Formula}},\ \boldsymbol{FE} \in \textbf{\textit{FuncExpr}}$$
$$\boldsymbol{c} \in \textbf{\textit{Int32}} = \{\ldots, \textbf{\textit{-1}}, \textbf{\textit{0}}, \textbf{\textit{1}}, \ldots\} \quad \boldsymbol{b} \in \textbf{\textit{Bool}} = \{\textbf{\textit{True}}, \textbf{\textit{False}}\}$$
$$\boldsymbol{I_{Int32}} \in \textbf{\textit{Int32Id}} = \{\textbf{\textit{EAX}}, \textbf{\textit{ESP}}, \textbf{\textit{EBP}}, \ldots\}$$
$$\boldsymbol{I_{Bool}} \in \textbf{\textit{BoolId}} = \{\textbf{\textit{CF}}, \textbf{\textit{SF}}, \ldots\} \quad \boldsymbol{F} \in \textbf{\textit{FuncId}} = \{\textbf{\textit{Mem}}\}$$
$$\boldsymbol{op} \in \textbf{\textit{ArithOp}} = \{+, -, \ldots\} \quad \boldsymbol{bop} \in \textbf{\textit{BoolOp}} = \{\wedge, \vee, \ldots\}$$
$$\boldsymbol{rop} \in \textbf{\textit{RelOp}} = \{=, \neq, <, \ldots\}$$
$$\boldsymbol{T} ::= \boldsymbol{c} \mid \boldsymbol{I_{Int32}} \mid \boldsymbol{T_1\ op\ T_2} \mid \textbf{\textit{ite}}(\boldsymbol{\varphi}, \boldsymbol{T_1}, \boldsymbol{T_2}) \mid \boldsymbol{F}(\boldsymbol{T_1})$$
$$\boldsymbol{\varphi} ::= \boldsymbol{b} \mid \boldsymbol{I_{Bool}} \mid \boldsymbol{T_1\ rop\ T_2} \mid \neg\boldsymbol{\varphi_1} \mid \boldsymbol{\varphi_1\ bop\ \varphi_2} \mid \boldsymbol{F = FE}$$
$$\boldsymbol{FE} ::= \boldsymbol{F} \mid \boldsymbol{FE_1}[\boldsymbol{T_1} \mapsto \boldsymbol{T_2}]$$

Figure 6: Syntax of L[IA-32].

$$\mathcal{I}[\![\texttt{mov eax,[ebp]}]\!]\rho = \mathit{update}_{\mathit{reg}}(\rho, [\![\texttt{eax}]\!], \mathit{access}_{\mathit{mem}}(\rho, \mathit{access}_{\mathit{reg}}(\rho, [\![\texttt{ebp}]\!])))$$

2.4 QFBV Formulas for IA-32

The semantics of IA-32 instructions can also be formally expressed by formulas in a logic. Consider a quantifier-free bit-vector logic L over finite vocabularies of constant symbols and function symbols. We will be dealing with a specific instantiation of L, denoted by L[IA-32]. (L can also be instantiated for other ISAs.) In L[IA-32], some constants represent IA-32's registers (***EAX***, ***ESP***, ***EBP***, etc.), and some represent flags (***CF***, ***SF***, etc.). L[IA-32] has only one function symbol "***Mem***," which denotes memory. Note that syntactic constructs of L[IA-32] are boldfaced to distinguish them from their counterparts in IA-32 semantics. The syntax of L[IA-32] is defined in Fig. 6.

A term of the form $\textbf{\textit{ite}}(\boldsymbol{\varphi}, \boldsymbol{T_1}, \boldsymbol{T_2})$ represents an if-then-else expression. A ***FuncExpr*** of the form $\boldsymbol{FE}[\boldsymbol{T_1} \mapsto \boldsymbol{T_2}]$ denotes a *function-update* expression.

The function $\langle\!\langle\cdot\rangle\!\rangle$ converts an IA-32 instruction sequence into a QFBV formula. While others have created such encodings by hand (e.g., [38]), we use a method that takes a specification of the concrete operational semantics of IA-32 instructions and creates a QFBV encoder automatically. The method reinterprets each semantic operator as a QFBV formula-constructor or term-constructor. (See [30].) To write formulas that express store transitions, all ***Int32Id***s, ***BoolId***s, and ***FuncId***s can be qualified by primes (e.g., ***Mem′***). The QFBV formula for an instruction sequence is a restricted 2-vocabulary formula that specifies a store transformation (also known as a "transition formula"). It has the form

$$\bigwedge\nolimits_m (\boldsymbol{I'_m} = \boldsymbol{T_m}) \wedge \bigwedge\nolimits_n (\boldsymbol{J'_n} = \boldsymbol{\varphi_n}) \wedge \textbf{\textit{Mem}}' = \boldsymbol{FE},$$

where I'_m and J'_n range over the constant symbols for registers and flags, respectively. The primed vocabulary is the post-store vocabulary, and the unprimed vocabulary is the pre-store vocabulary. We also refer to them as "vocabulary 1" and "vocabulary 0," respectively. The QFBV formulas for a few IA-32 instructions are given in Fig. 7. The second instruction pushes the 32-bit constant value 0 on the stack. If the zero flag *ZF* is set, the third instruction updates *EIP*

$$\langle\!\langle \texttt{mov eax,[ebp]} \rangle\!\rangle \equiv \boldsymbol{EAX' = Mem(EBP)}$$

$$\langle\!\langle \texttt{push 0} \rangle\!\rangle \equiv \boldsymbol{ESP' = ESP - 4 \wedge Mem' = Mem[ESP - 4 \mapsto 0]}$$

$$\langle\!\langle \texttt{jz 1000} \rangle\!\rangle \equiv \boldsymbol{ite(ZF = 0, EIP' = 1000, EIP' = EIP + 4)}$$

$$\langle\!\langle \texttt{lea eax,[ebp+4]} \rangle\!\rangle \equiv \boldsymbol{EAX' = EBP + 4}$$

Figure 7: QFBV formulas for example IA-32 instructions.

to the value 1000; otherwise, it increments *EIP* by four. The fourth instruction loads *EAX* with the value *EBP+4* (without modifying the value of any flag).

In this section, and in the rest of the paper, we show only the portions of QFBV formulas that express how the store is *modified*. QFBV formulas actually contain identity conjuncts of the form $\boldsymbol{I' = I}$, $\boldsymbol{J' = J}$, and $\boldsymbol{Mem' = Mem}$ for constants and functions that are *unmodified*.

3. Overview

At a very high level, WIPER is similar to the off-line source-code partial evaluator described in §2.1: WIPER's algorithm works in two phases, BTA and specialization. However, because WIPER is dealing with machine code, WIPER's algorithm differs significantly from that of a source-code partial evaluator. In this section, we present an example to illustrate WIPER's algorithm, while highlighting the following: (i) the issues that arise in machine code, (ii) why techniques used in a source-code partial evaluator are unsatisfactory to resolve the issues, and (iii) how WIPER resolves the issues.

Fig. 8 shows an IA-32 program sum that takes, a, v, and n as inputs, and computes $a + b[n]$. (Array b is statically allocated and has 5 elements. Lines 6–12 in Fig. 8 initialize the elements of b to v.) n is assumed to be between 0 and 4. The inputs a, v, and n are available in the `eax`, `ebx`, and `ecx` registers, respectively, at the beginning of the program. The output is available in the `eax` register at the end of the program. Let us use WIPER to partially evaluate sum with respect to the input value $v = 1$.

No source-code variables. Recall from §2.1 that the goal of BTA is to compute the division of all program variables as either static or dynamic. The abstraction of a source-code "variable" is absent at the machine-code level. (We assume that the input binary lacks symbol-table and debugging information.) Consequently, a division of source-code variables cannot be obtained at the machine-code level.

However, there are tools [10] that recover "variable-like" abstractions [9] from machine code, and use them to construct IRs such as an SDG, a control flow graph (CFG), etc. WIPER performs BTA via forward slicing over the SDG recovered from the binary [17].

The inputs to WIPER's BTA phase are: (i) the SDG of the binary, and (ii) the instructions that initialize the dynamic inputs (the slicing criterion). The output is an annotated program, whose instructions are annotated with binding times: static (S), or dynamic (D). In our example, the dynamic inputs a and n are available in the `eax` and `ecx` registers at instructions `1` and `3`, respectively. Consequently, instructions `1` and `3` constitute the slicing criterion. All instructions that are in the forward slice depend on a and n, and hence are classified dynamic; the remaining instructions are classified static.

```
1: push eax; a
2: push ebx; v
3: push ecx; n
4: push 4
5: sub esp,20; b
;Initialization loop
6: L2:cmp [esp+20],0
7: jl L1
8: mov ebx,[esp+28]
9: mov edx,[esp+20]
10: mov [esp+edx*4],ebx
11: sub [esp+20],1
12: jmp L2
;Computation of a+b[n]
13: L1:mov eax,[esp+32]
14: mov ecx,[esp+24]
15: mov ebx,[esp+ecx*4]
16: add eax,ebx
17: add esp,36
```

Figure 8: Input sum program, which computes $a + b[n]$.

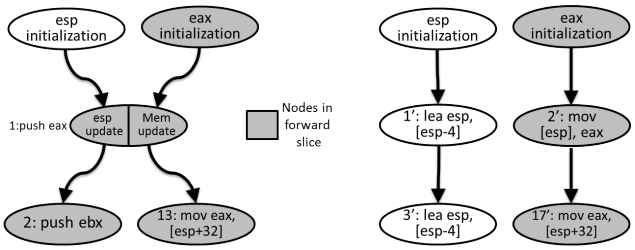

Figure 9: SDG snippet to illustrate decoupling.

Another feature of ISAs that makes them different from typical high-level languages is that most instructions are multi-assignments: they have several inputs, and several outputs. The next two issues discussed below arise in machine-code partial evaluation because instructions have multiple outputs, and multiple inputs, respectively.

The granularity issue in slicing. Consider instruction `1` in the sum program. `1` modifies the stack-pointer register *ESP* in addition to copying the value in the *EAX* register to a memory location. Because all of the remaining instructions in sum either directly or transitively use *ESP*, the slice with respect to instruction `1` consists of all the instructions in sum, and thus every instruction in sum would be classified dynamic. This imprecision in slicing is caused by a granularity problem: the `push` instruction *independently* updates *ESP* and the memory location. Because the dynamic input is in *EAX*, the slice requires only the memory update that `push` performs and not the update to the value of *ESP*. However, a slice cannot include only a part of an instruction; it has to include the entire instruction. Consequently, the entire `push` instruction, and the remaining instructions in sum are added to the slice. This issue is illustrated by the SDG snippet on the left in Fig. 9.

To resolve this issue, WIPER rewrites sum by decoupling each instruction that independently updates the stack pointer along with another register or memory location *l* (e.g., `push`, `pop`, `leave`, etc.). For such instructions, WIPER uses a machine-code synthesizer [46] (a tool that synthesizes an instruction sequence from a semantic specification) to synthe-

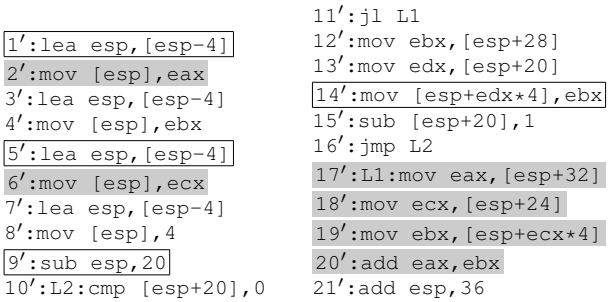

Figure 10: Rewritten program sum′.

size an equivalent instruction sequence that updates *ESP* in one instruction and *l* in a different instruction. The effect of decoupling on BTA is illustrated by the SDG snippet on the right in Fig. 9.

The rewritten binary sum′ is shown in Fig. 10. In sum′, the new slicing criterion includes instructions 2′ and 6′. The new forward slice includes only the instructions highlighted in Fig. 10 in light gray, which are classified dynamic. The remaining instructions are classified static.[4]

The inputs to the specialization phase of WIPER are: (i) the interprocedural CFG of the binary, (ii) the binding-time annotations, and (iii) an input partial static-store. The output is the residual program. Recall from §2.3 that an IA-32 store is a triple that consists of a register map, a flag map, and a memory map. The input partial static-store ρ_0 is

$$\rho_0 \equiv \langle [ESP \mapsto 1000][EBX \mapsto 1], [\,], [\,] \rangle.$$

In the rest of the paper, when we use the term "store," we are referring to a partial static-store. The input store has the value *1* for v, and *1000* for the stack pointer. Let us specialize sum′ with respect to ρ_0.

Non-parameterized operands in Instruction ASTs. Recall from §2.1 that $\mathbf{PE}_S$ substitutes values for operands while specializing ASTs of dynamic expressions and statements. However, WIPER cannot use this simple approach for specializing dynamic instructions. For an instruction that has multiple input operands, the instruction AST may not always be parameterized by all operands. For example, the instruction "`adc eax,ebx`" adds the values in registers *EAX* and *EBX*, and the value of the carry flag *CF*. The flag operand is "baked into" the semantics, and is not an explicit syntactic operand. Another example is the instruction "`push eax`," which has the stack-pointer operand (*ESP*) baked into the semantics.

To handle this issue, WIPER specializes a representation of the *semantics* of an instruction—instead of its syntax—with respect to a store. WIPER uses a QFBV formula to represent an instruction's semantics.

Now let us use the following approach to specialization:

[4] The division produced by WIPER's BTA is pointwise, and monovariant for procedures (i.e., an instruction in a procedure is classified as static or dynamic for all calling contexts).

1. Static instructions: Evaluate.
2. Dynamic instructions: Specialize the instruction's QFBV formula with respect to the pre-store, and residuate code. WIPER uses a machine-code synthesizer to synthesize an instruction-sequence that is equivalent to the specialized QFBV formula. The synthesized instruction-sequence constitutes the residual code.

Let us specialize sum′ with respect to ρ_0 using this simple approach. The specializer evaluates instruction 1′ in sum′ because it is static. The post-store is ρ_1.

$$\rho_1 \equiv \langle [ESP \mapsto 996][EBX \mapsto 1], [\,], [\,] \rangle. \tag{2}$$

The QFBV formula for instruction 2′ is

$$\langle\!\langle 2' \rangle\!\rangle \equiv \boldsymbol{Mem' = Mem[ESP \mapsto EAX]}.$$

Because 2′ is dynamic, the specializer specializes $\langle\!\langle 2' \rangle\!\rangle$ with respect to ρ_1 to obtain the specialized formula

$$\boldsymbol{Mem' = Mem[996 \mapsto EAX]},$$

and uses the synthesizer to synthesize the instruction "`mov [996],eax`." After evaluating static instructions 3′ and 4′, the resulting store is

$$\rho_4 \equiv \langle [ESP \mapsto 992][EBX \mapsto 1], [\,], [992 \mapsto 1] \rangle. \tag{3}$$

The residual program produced by this simple approach is given below.

```
mov [996],eax          mov ecx,[988]
mov [988],ecx          mov ebx,[964+ecx*4]
mov eax,[996]          add eax,ebx
```

Lifting. The approach described above produces code that can access uninitialized locations. For example, one can see that the instruction "`mov ebx,[964+ecx*4]`" in the residual program dereferences a dynamic pointer to access an element in the static array b. However, the static array is not initialized by the residual code. This issue arises because our simple specialization approach did not residuate code for static instructions that produce values that *might* be consumed by a downstream dynamic instruction (a.k.a., a dynamic context).

To resolve this issue, WIPER conservatively lifts static predecessors of a dynamic instruction. In Fig. 10, the boxed instructions are the lifted instructions. After lifted instructions are identified, one can use the following approach to specialization:

Approach $\mathbf{PE}_1$

- Static instructions: Evaluate.
- Lifted instructions: Specialize the instruction's QFBV formula with respect to the pre-store, and residuate code. Then update the store by evaluating the instruction.
- Dynamic instructions: Emit. Because all static locations that might be used by a dynamic instruction *d* have been initialized by upstream residual lifted instructions, there is no static data that needs to be infused into *d* via specialization, and the unmodified *d* can be emitted. ■

```
1₁: mov esp, 996          13₁³: mov [972], 1
2₁: mov [esp],eax         13₁⁴: mov [968], 1
5₁: mov esp,988           13₁⁵: mov [964], 1
6₁: mov [esp],ecx         16₁: mov eax,[esp+32]
9₁: mov esp, 964          17₁: mov ecx,[esp+24]
13₁¹: mov [980], 1        18₁: mov ebx,[esp+ecx*4]
13₁²: mov [976], 1        19₁: add eax,ebx
```

Figure 11: Residual program sum_1.

Let us specialize sum′ with respect to ρ_0 using $\mathbf{PE}_1$. Because 1′ is lifted, the specializer residuates code for it. The QFBV formula for 1′ is $\langle\!\langle 1' \rangle\!\rangle \equiv \boldsymbol{ESP' = ESP - 4}$. The specializer specializes $\langle\!\langle 1' \rangle\!\rangle$ with respect to ρ_0 to obtain the specialized formula $\boldsymbol{ESP' = 996}$, and uses the synthesizer to synthesize instruction 1_1 in sum_1 (shown in Fig. 11). The specializer also evaluates 1′ to obtain ρ_1 (Eqn. (2)). Because 2′ is dynamic, the specializer emits it as 2_1. Because 3′ is static, the specializer evaluates the instruction without residuating code. The new residual program sum_1 is shown in Fig. 11. In sum_1, one can see that all the static values required by emitted dynamic instructions (2_1, 6_1, etc.) have been set up by residual lifted instructions (1_1, 5_1, etc.).

Residual specialization-time addresses. In sum_1, one can see that the specialization-time values of the stack pointer have been residuated as constants. Although sum_1 is a correct residual program for sum′ partially evaluated with respect to ρ_0, the specialization-time stack layout is hardwired into the residual code. (The stack begins at address 1000, and grows toward lower addresses.) As a result, the residual code might crash if a different address is used as the base of the runtime stack.

One way to prevent the specializer from emitting stack-pointer values is to make the stack pointer dynamic by including instruction 1′ in the slicing criterion during BTA. However, that would make all instructions in sum′ dynamic.

To resolve this issue, WIPER uses a *symbolic* store during specialization, instead of a *concrete* store. In the initial symbolic store, specialization-time addresses (e.g., the initial value of *ESP*) are represented using symbolic constants. Static non-address values are represented using integer and Boolean constants. WIPER uses $\mathbf{PE}_2$ for specialization.

Approach $\mathbf{PE}_2$

- Static instructions: Symbolically evaluate. Although specialization-time addresses are symbolic, WIPER can still access and update static values in memory.
- Lifted instructions: Specialize the instruction's QFBV formula with respect to the symbolic pre-store, and residuate code. Then update the symbolic store by symbolically executing the instruction.
- Dynamic instructions: Emit. ■

Let us illustrate $\mathbf{PE}_2$ on sum′. The new initial store is $\rho_0^{sym} \equiv \langle [ESP \mapsto \boldsymbol{c}][EBX \mapsto \boldsymbol{1}], [\,], [\,]\rangle$. Note that the values in the store are boldface to indicate that they are logical terms, and not semantic values. The value of the stack pointer *ESP* is the symbolic constant $\boldsymbol{c}$ to indicate that

```
0₂: mov esi,esp           13₂²: mov [esi-24],1
1₂: lea esp,[esi-4]       13₂³: mov [esi-28],1
2₂: mov [esp],eax         13₂⁴: mov [esi-32],1
5₂: lea esp,[esi-12]      13₂⁵: mov [esi-36],1
6₂: mov [esp],ecx         16₂: mov eax,[esp+32]
9₂¹: lea esp,[esi-16]     17₂: mov ecx,[esp+24]
9₂²: sub esp,20           18₂: mov ebx,[esp+ecx*4]
13₂¹: mov [esi-20],1      19₂: add eax,ebx
```

Figure 12: Residual program sum_2.

we are not restricting *ESP* to a concrete value known at specialization-time.

Before starting specialization, WIPER residuates instruction 0_2 in sum_2 (shown in Fig. 12) to save the initial value of the stack pointer *ESP* in a dedicated location. (In this example, WIPER uses the *ESI* register because *ESI* is not used in sum′.) This location will be used whenever downstream residual instructions need the initial value of *ESP*. To residuate code for the lifted instruction 1′, WIPER specializes $\langle\!\langle 1' \rangle\!\rangle$ with respect to ρ_0^{sym}. The specialized formula is $\boldsymbol{ESP' = c - 4}$. Because at run-time the value of $\boldsymbol{c}$ will be available in *ESI*, WIPER replaces $\boldsymbol{c}$ with $\boldsymbol{ESI}$ to produce $\boldsymbol{ESP' = ESI - 4}$. WIPER uses the synthesizer to synthesize instruction 1_2 for the specialized formula. WIPER also symbolically evaluates 1′ to produce ρ_1^{sym}.

$$\rho_1^{sym} \equiv \langle [ESP \mapsto \boldsymbol{c - 4}][EBX \mapsto \boldsymbol{1}], [\,], [\,]\rangle.$$

2′ is dynamic, and WIPER emits it as 2_2. After symbolically evaluating static instructions 3′ and 4′, we obtain the store

$$\rho_4^{sym} \equiv \langle [ESP \mapsto \boldsymbol{c - 8}][EBX \mapsto \boldsymbol{1}], [\,], [\boldsymbol{c - 8} \mapsto \boldsymbol{1}]\rangle.$$

The final residual program sum_2 produced by WIPER is shown in Fig. 12. One can see that WIPER arranges for the initial stack-pointer value to be retrieved from *ESI* (1_2, 5_2, 9_2, 13_2^1, etc.) instead of residuating the stack-pointer value at residual lifted instructions (cf. 1_1, 5_1, 9_1, 13_1^1, etc. in sum_1); consequently, sum_2 will not crash when the stack base is initialized to different addresses. In contrast, static non-address quantities are residuated at lifted instructions (e.g., the value 1 in instructions 13_2^1 through 13_2^5).

Why do $\mathbf{PE}_1$ and $\mathbf{PE}_2$ work?

To formalize the effect of $\mathbf{PE}_1$ and $\mathbf{PE}_2$, we introduce the notion of an *environment*, which makes explicit the starting address of the stack and heap blocks. An environment maps a symbolic constant denoting the starting address of the stack or a heap block to a concrete address (or "location"). A machine-code state σ consists of an environment η, and a store ρ, which maps locations to values. For example, ρ_4 in Eqn. (3) is actually part of a state σ_4

$$\sigma_4 \equiv \langle \eta_4, \rho_4 \rangle \equiv \langle \boldsymbol{Stack} \mapsto 1000,$$
$$\langle [ESP \mapsto 992][EBX \mapsto 1], [\,], [992 \mapsto 1]\rangle\rangle.$$

η_4 maps the start of the stack, denoted by the symbolic constant $\boldsymbol{Stack}$, to the address *1000*.

Suppose that η and ρ can be partitioned into η_S and η_D, and ρ_S and ρ_D, respectively. The effect of $\mathbf{PE}_1$ on

Algorithm 1 Instruction Decoupling (Pre-processing step)

Input: Binary B
Output: Rewritten binary B′

```
1: for each i ∈ B do
2:   if UpdatesSPAndLoc(i) then
3:     I ← Synth(InstrToQFBV(i))
4:     B′ ← Replace(B, i, I)
5:   end if
6: end for
7: return B′
```

each execution trace of the binary can be expressed by the following reparenthesization:

$$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, \eta_S, \rho_S) : (\eta_D, \rho_D) \quad (4)$$

Operationally, the partial evaluator creates a new residual instruction for every unique (l, η_S, ρ_S) triple it observes at specialization time. Eqn. (4) shows how each element $(l, \eta_S, \rho_S) : (\eta_D, \rho_D)$ in an execution trace of a binary specialized using $\mathbf{PE}_1$ can be mapped back to the corresponding element $l : (\eta_S, \rho_S, \eta_D, \rho_D)$ in the trace of the original binary. Consequently, the specialization-time memory layout (η_S) is hard-wired into the residual code. If the run-time memory layout is different, the residual code produced by $\mathbf{PE}_1$ might crash.

To describe the symbolic techniques used in $\mathbf{PE}_2$, we introduce a static symbolic store ρ_S^{sym}, which maps symbolic expressions to symbolic expressions, where each such symbolic expression is parameterized on the symbolic constants found in η_S. (Note that the domain of ρ_S^{sym} also contains registers and flags.) ρ_S^{sym} mirrors the static concrete store ρ_S, i.e., $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, where the left-hand side denotes the simplification of each symbolic expression in ρ_S^{sym} with respect to the values obtained from η_S. The principal effect of $\mathbf{PE}_2$ on each execution trace of the binary can be expressed by the following reparenthesization:

$$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, \rho_S^{sym}) : (\eta_S, \eta_D, \rho_D) \quad (5)$$

Eqn. (5) shows how each element $(l, \rho_S^{sym}) : (\eta_S, \eta_D, \rho_D)$ in an execution trace of a binary specialized using $\mathbf{PE}_2$ can be mapped back to the corresponding element $l : (\eta_S, \rho_S, \eta_D, \rho_D)$ in the trace of the original binary. In effect, the residual code is independent of η_S. (See also §4.4.)

4. Algorithm

In this section, we describe the algorithms used by WIPER. First, we present the algorithm for a pre-processing step used prior to partial evaluation. Second, we present WIPER's intraprocedural partial-evaluation algorithm. Third, we present extensions to handle multiple procedures. Finally, we present the threats to the validity of our algorithms.

4.1 Pre-processing: Decoupling Instructions

Prior to performing BTA, WIPER rewrites the input binary by decoupling each instruction *i* that updates the stack-pointer register *ESP* along with another location *l*. WIPER uses the machine-code synthesizer MCSYNTH [46] for decoupling. MCSYNTH uses a divide-and-conquer strategy that splits *i*'s QFBV formula φ into two independent subformulas φ_1 and φ_2, such that the updates to *ESP* and *l* are split between φ_1 and φ_2. Then, MCSYNTH synthesizes code for φ_1 and φ_2, concatenates the results, and returns the resulting instruction sequence *I*. *ESP* and *l* are updated in different instructions in *I*. The rewriting of the input binary via instruction decoupling is shown as Alg. 1. In the algorithm, `UpdatesSPAndLoc` returns true if an instruction updates the stack pointer along with another register or memory location; `InstrToQFBV` converts an instruction into a QFBV formula; `Synth` invokes MCSYNTH to synthesize an instruction sequence; `Replace` replaces an instruction in a binary with an instruction sequence.

Algorithm 2 BTA algorithm in WIPER

Input: SDG, I_D
Output: BTAMap

```
1: slice ← ForwardSlice(SDG, I_D)
2: for each I ∈ slice do
3:   BTAMap ← BTAMap[I ↦ Dynamic]
4:   for each static predecessor S of I do
5:     BTAMap ← BTAMap[S ↦ Lifted]
6:   end for
7: end for
8: for each I ∉ Dom(BTAMap) do
9:   if I is a call to malloc or an actual argument of a call to
     malloc then
10:    BTAMap ← BTAMap[I ↦ Lifted]
11:  else
12:    BTAMap ← BTAMap[I ↦ Static]
13:  end if
14: end for
15: return BTAMap
```

4.2 Intraprocedural Partial Evaluation

In this section, we present WIPER's intraprocedural partial-evaluation algorithm. First, we present the BTA algorithm; then, we present the specialization algorithm.

4.2.1 BTA

The slicing-based BTA algorithm used in WIPER is shown as Alg. 2. In the algorithm, I_D is the set of instructions that initialize dynamic inputs; `ForwardSlice` computes a forward slice with respect to a slicing criterion; BTAMap is a data structure that maps each instruction to its binding time. If a static instruction is control dependent on a dynamic branch, the specialization algorithm might not terminate [25, Chap. 14]. To avoid the possibility of non-termination, `ForwardSlice` follows both data and control dependences during slicing. Lines 4–6 in Alg. 2 show the computation of lifted instructions during BTA. WIPER lifts all static calls to `malloc` (lines 9–11). This step is performed so that heap blocks that are allocated at specialization-time also get allocated at run-time. (See §4.2.2.3.)

Algorithm 3 WIPER's specialization loop - IntraPE

Input: CFG, ρ, BTAMap
Output: ρ'

```
 1: CFG' ← NewCFG()
 2: bb ← CFG.FirstBB()
 3: worklist ← ⟨bb, ρ⟩
 4: processed ← ∅
 5: ρ' ← ρ
 6: while worklist ≠ ∅ do
 7:    ⟨bb, ρ⟩ ← RemoveItem(worklist)
 8:    if ⟨bb, ρ⟩ ∈ processed then
 9:       continue
10:    end if
11:    processed ← processed ∪ {⟨bb, ρ⟩}
12:    for each instruction i ∈ bb do
13:       ⟨i', ρ⟩ ← specialize(i, ρ, BTAMap)
14:       if i' ≠ ϵ then
15:          CFG'.Emit(i')
16:       end if
17:    end for
18:    if CFG.FinalBB(bb) then
19:       ρ' ← ρ
20:    end if
21:    for each successor s of bb do
22:       worklist ← worklist ∪ {⟨s, ρ⟩}
23:    end for
24: end while
25: return ρ'
```

4.2.2 Specialization

This section presents the specialization algorithm used in WIPER. We first present the skeleton of WIPER's specialization loop. We then present two versions of the core specialization function specialize, which gets called in the specialization loop. While the first version produces correct residual code, the residual code cannot be executed with the stack and heap blocks at different positions at run-time than at specialization-time. The second version does not have this limitation, and is used in WIPER.

4.2.2.1 Specialization loop. The skeleton of WIPER's specialization loop is similar to that of a standard partial evaluator [25, p. 87], and is given as Alg. 3. specialize is the core specialization function (line 13 of Alg. 3), which specializes an instruction with respect to a pre-store to obtain the residual instruction and a post-store. In the algorithm, ϵ denotes the empty sequence of instructions; `NewCFG` creates a new CFG; `FirstBB` returns the first basic-block in a CFG; `FinalBB` returns true if the argument is the final basic-block in its parent CFG; `RemoveItem` removes an item from the worklist; `Emit` appends an instruction to a CFG. (We assume that `Emit` creates and connects nodes in the CFG.)

Recall that the register *EIP* is the IA-32 program counter. Alg. 3 follows CFG edges, rather than the value of *EIP*, because we have no way to give consistent *EIP* values to new instructions created via decoupling. Alg. 3 does not even track *EIP* in stores, and consequently, it can propagate the same store to the two successors of a branch instruction (lines 21–23).

Algorithm 4 specialize$_1$

Input: I, ρ, BTAMap
Output: $\langle \mathrm{I}', \rho' \rangle$

```
1: if BTAMap[I] = Dynamic then
2:    return ⟨I, ρ⟩
3: else if BTAMap[I] = Lifted then
4:    return ⟨reduce1(I, ρ), eval1(I, ρ)⟩
5: else
6:    return ⟨ϵ, eval1(I, ρ)⟩
7: end if
```

4.2.2.2 PE_1 Specializer. specialize$_1$ is a specialization function that implements $\mathbf{PE}_1$. specialize$_1$ has two constituent functions, eval$_1$ and reduce$_1$. eval$_1$ evaluates instructions, and reduce$_1$ residuates code. The algorithm for specialize$_1$ is given as Alg. 4.

***eval*$_1$.** eval$_1$ can be implemented by writing an interpreter for IA-32 instructions using the concrete operational-semantics of IA-32. eval$_1$ evaluates an instruction with respect to a pre-store, and returns the post-store.

$$\mathsf{eval}_1 : \text{Instruction} \rightarrow \mathit{Store} \rightarrow \mathit{Store} \quad \mathsf{eval}_1(\mathrm{I}, \rho) = \mathcal{I}\,[\![\mathrm{I}]\!]\,\rho$$

***reduce*$_1$.** reduce$_1$ can be defined as follows:

$$\mathsf{reduce}_1 : \text{Instruction} \rightarrow \mathit{Store} \rightarrow \text{Instruction-sequence}$$
$$\mathsf{reduce}_1(\mathrm{I}, \rho) = \mathtt{Synth}(\mathtt{Subst}_0(\mathtt{InstrToQFBV}(\mathrm{I}), \rho))$$

reduce$_1$ converts an instruction into a QFBV formula, specializes the formula with respect to the pre-store (using `Subst`$_0$), and uses a machine-code synthesizer to synthesize an instruction sequence that is equivalent to the specialized formula. The synthesized instruction-sequence constitutes the residual code. Recall from §2.4 that vocabulary-0 terms and formulas in a QFBV formula represent pre-store locations and predicates, respectively. `Subst`$_0$ replaces vocabulary-0 terms and formulas in I's QFBV formula with static values from ρ. Because lifted instructions are originally static, they access only static locations. Consequently, `Subst`$_0$ replaces *all* vocabulary-0 terms and formulas in I's QFBV formula with static values from the store.

For example, suppose that I is the instruction "`add [esp],eax`." I's QFBV formula φ is given below. (For brevity, we only show one of the flags updated by `add`.)

$$\varphi \equiv \mathit{Mem}' = \mathit{Mem}[\mathit{ESP} \mapsto \mathit{Mem}(\mathit{ESP}) + \mathit{EAX}] \wedge$$
$$\mathit{SF}' = (\mathit{Mem}(\mathit{ESP}) + \mathit{EAX} < 0)$$

Suppose that ρ is $\rho \equiv \langle [\mathit{ESP} \mapsto \mathit{1000}][\mathit{EAX} \mapsto \mathit{2}], [\,], [\mathit{1000} \mapsto \mathit{1}] \rangle$. `Subst`$_0$ produces the following formula:

$$\mathit{Mem}' = \mathit{Mem}[\mathit{1000} \mapsto \mathit{3}] \wedge \mathit{SF}' = \mathit{False},$$

and reduce$_1$(I, ρ) produces the residual instruction-sequence "`mov [1000],3; add [1000],0`." (The second instruction in the sequence is necessary to set the *SF* flag.)

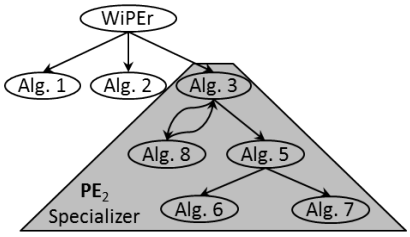

Figure 13: Call graph depicting the organization of WIPER.

```
1: mov [esp],16 ;L          1_R: mov [esp],16
2: call malloc ;L           2^1_R: call malloc
3: mov [eax],1 ;S           2^2_R: mov edi,eax
4: mov [eax+4],2 ;L         4_R: mov [edi+4],2
5: add ebx,[eax+4] ;D       5_R: add ebx,[eax+4]
```

Figure 14: Code snippet, and residual program to illustrate specialize$_2$ on heap blocks.

4.2.2.3 PE_2 specializer. As discussed in §3, Alg. 4 specializes the binary with respect to static addresses as well as static values (e.g., sum$_1$ in Fig. 11). In this section, we present the specialization algorithm that WIPER uses to residuate code that is independent of the specialization-time memory layout. In particular, we illustrate how the algorithm residuates code that allows the stack and heap blocks to be at different positions at run-time than at specialization-time.

WIPER's call-graph is represented in Fig. 13. WIPER's specializer is highlighted in Fig. 13 in light gray. WIPER uses specialize$_2$ (Alg. 5) as the specialization function in Alg. 3. specialize$_2$ implements **PE**$_2$. The remaining components of the specializer (Algs. 6, 7, and 8) will be described later in upcoming paragraphs and sections.

specialize$_2$ uses symbolic addresses and a symbolic store to access and update memory at specialization time. To represent symbolic addresses, we enhance ***Int32Id*** in L[IA-32] by adding symbolic constants (***m***, ***n***, etc.). A different constant is used to represent the starting address of the stack and each heap block allocated at specialization-time.

$$\textbf{\textit{Int32Id}} = \{\textbf{\textit{EAX}}, \textbf{\textit{ESP}}, \textbf{\textit{EBP}}, ..., \boldsymbol{m}, \boldsymbol{n}, ...\}$$

specialize$_2$ uses a symbolic store $Store^{sym}$ instead of *Store*.

$$\rho^{sym} \in Store^{sym} = RegMap^{sym} \times FlagMap^{sym} \times MemMap^{sym}$$

$$RegMap^{sym} : \textbf{\textit{Int32Id}} \rightarrow \textbf{\textit{Term}} \quad MemMap^{sym} : \textbf{\textit{Term}} \rightarrow \textbf{\textit{Term}}$$

$$FlagMap^{sym} : \textbf{\textit{BoolId}} \rightarrow \textbf{\textit{Formula}}$$

The residual code generated by specialize$_2$ will use specific run-time addresses to access and update memory using an extra level of indirection. To achieve this effect, specialize$_2$ uses a memory-layout map $\mu \in MemLayout$. A memory-layout map μ maps the symbolic starting-address ***m*** of the stack or a heap block to a location *l* (register or global address) that holds the run-time counterpart of ***m***.

$$\mu \in MemLayout : \textbf{\textit{Int32Id}} \rightarrow (Register \cup INT)$$

We will illustrate specialize$_2$, and its constituent functions eval$_2$ and reduce$_2$, using the code snippet shown in Fig. 14, which allocates and uses a heap block. (We describe how specialize$_2$ handles the stack and stack locations at the end of this sub-section.) The snippet has a static call to `malloc` at instruction `2`, and static values are written to the allocated heap-block at instructions `3` and `4`. The binding-time annotations for the instructions are shown as comments on the left-hand column of Fig. 14. Because WIPER lifts all static calls to `malloc`, instructions `1` and `2` are lifted.

Algorithm 5 specialize$_2$

Input: I, ρ, BTAMap, μ
Output: $\langle I', \rho', \mu' \rangle$
1: **if** BTAMap[I] = Dynamic **then**
2: **return** $\langle I, \rho, \mu \rangle$
3: **else if** BTAMap[I] = Lifted **then**
4: $\boldsymbol{m} \leftarrow \epsilon$
5: $l \leftarrow \epsilon$
6: **if** `Alloc(I)` **then**
7: $\boldsymbol{m} \leftarrow$ `NewConst()`
8: $l \leftarrow$ `NewLoc()`
9: $\mu' \leftarrow \mu[\boldsymbol{m} \mapsto l]$
10: **end if**
11: **return** $\langle$reduce$_2$(I, ρ, μ, l), eval$_2$(I, ρ, ***m***), $\mu' \rangle$
12: **else**
13: **return** $\langle \epsilon$, eval$_2$(I, ρ, ϵ), $\mu \rangle$
14: **end if**

Algorithm 6 eval$_2$

Input: I, ρ, ***m***
Output: ρ'
1: $\rho' \leftarrow \mathcal{I}^{sym}\ [\![\text{I}]\!]\ (\rho[NextBlock \mapsto \boldsymbol{m}])$
2: **return** ρ'

Algorithm 7 reduce$_2$

Input: I, ρ, μ, l
Output: I$'$
1: I$' \leftarrow$ `Synth(Subst(Subst`$_0$`(InstrToQFBV(I), `ρ`), `μ`))`
2: I$'' \leftarrow \epsilon$
3: **if** `Alloc(I)` **then**
4: I$'' \leftarrow$ `EmitMove(`l, *EAX*`)`
5: **end if**
6: **return** I$'$; I$''$

The algorithms for specialize$_2$, eval$_2$, and reduce$_2$ are given as Algs. 5, 6, and 7, respectively. In Alg. 5, we overload ϵ to denote "no symbolic constant" (lines 4 and 13) and "no location" (line 5), respectively. In our example, because instruction `2` allocates memory (checked by `Alloc`), specialize$_2$ adds $[\boldsymbol{m} \mapsto EDI]$ to μ, where ***m*** is a fresh symbolic constant (obtained using `NewConst`) that is used to represent the specialization-time starting address of the heap block, and *EDI* is a location that is not used in the residual binary (obtained using `NewLoc`). This location can also be a global memory location that is otherwise unused in the residual binary. specialize$_2$ passes ***m*** to eval$_2$, which stores ***m*** in ρ under the key *NextBlock*. eval$_2$ symbolically executes the instruction using $\mathcal{I}^{sym}$, which reinterprets the semantics of an instruction I to symbolically execute I. If I allocates memory, $\mathcal{I}^{sym}$ uses the symbolic constant bound to *NextBlock* in ρ as

the next address in the free list. Consequently, in ρ' returned by eval_2 for the call to `malloc`, *EAX* holds the symbolic constant $\boldsymbol{m}$—i.e., $\boldsymbol{m}$ is the starting address of the heap block allocated at instruction `2`.

$\mathsf{specialize}_2$ then passes I, ρ, μ, and *EDI* to reduce_2. Because instruction `2` allocates memory, reduce_2 residuates two instructions. The first instruction (2^1_R) is the call to `malloc`; the second instruction (2^2_R) saves the *EAX* register in *EDI* (emitted using `EmitMove`). After the residual code executes instruction 2^2_R, the run-time starting address of the heap block will be available in *EDI*.

$\mathsf{specialize}_2$ symbolically executes the static instruction `3`, producing the post-store $\langle[\boldsymbol{EAX} \mapsto \boldsymbol{m}], [\,], [\boldsymbol{m} \mapsto \boldsymbol{1}]\rangle$. While residuating code for lifted instruction `4`, reduce_2 uses $\mathtt{Subst}_0$ to replace all vocabulary-0 terms and formulas in `4`'s QFBV formula with terms and formulas, respectively, from the symbolic store. This step produces the formula

$$\boldsymbol{Mem'} = \boldsymbol{Mem}[\boldsymbol{m} + \boldsymbol{4} \mapsto \boldsymbol{2}]. \tag{6}$$

reduce_2 then replaces each symbolic constant $\boldsymbol{m}$ in the resulting formula with $[\![\mu(\boldsymbol{m})]\!]_L$ using `Subst`. For our example, for Eqn. (6), `Subst` produces $\boldsymbol{Mem'} = \boldsymbol{Mem}[\boldsymbol{EDI} + \boldsymbol{4} \mapsto \boldsymbol{2}]$. Finally, reduce_2 uses the synthesizer to residuate instruction 4_R for the specialized formula. One can see that when the residual code executes, instructions 4_R and 5_R use the correct starting address of the heap block.

Handling the stack. Suppose that $\boldsymbol{n}$ is the symbolic constant used to denote the specialization-time base of the stack, and l is a dedicated location that will hold the address of the base of the stack at run-time. Because the stack is typically allocated at the beginning of the program, WIPER adds $[\boldsymbol{n} \mapsto l]$ to μ before specialization begins, and emits "`mov l,esp`" as the first instruction in the residual program. Additionally, in the initial symbolic store, *ESP* is bound to $\boldsymbol{n}$. In our running example from §3, WIPER adds $[\boldsymbol{n} \mapsto \boldsymbol{ESI}]$ to μ. Consequently, reduce_2 uses $\boldsymbol{ESI}$ instead of $\boldsymbol{n}$ in the QFBV formulas of all downstream lifted instructions.

In summary, to residuate code that is independent of the specialization-time memory layout, $\mathsf{specialize}_2$ uses symbolic addresses, and residuates code that obtains the corresponding run-time addresses from designated locations, using one level of indirection. In effect, the binary is partially evaluated with respect to static values, but not specialization-time addresses.

$\mathbf{PE}_2$-Safety. For $\mathbf{PE}_2$ to produce correct residual code for a binary *B*, *B* should be ***$\mathbf{PE}_2$-safe***. A binary *B*, along with a specific division of instructions in *B* as static and dynamic, are $\mathbf{PE}_2$-safe if they satisfy the following properties:

- P1 *B* does not contain a static branch condition that depends on the starting address of the stack or heap blocks (e.g., `cmp esp,1000`).
- P2 Every static instruction that accesses (updates) memory in block M only accesses (updates) memory at a static offset from the base of M, and within the bounds of M.

```
main:
    :
A1: mov [esp],eax; S
C1: call foo
R1: mov [ebp-4],eax        foo:
    :                         push ebp
A2: mov [esp],eax; D          mov ebp, esp
C2: call foo               R: mov eax,[ebp-8]
R2: mov [ebp-8],eax           pop ebp
    :                         ret
```

Figure 15: Code snippet to illustrate issues related to interprocedural BTA.

P1 implies that the partial evaluator does not encounter a symbolic branch condition during specialization of *B*. P2 implies that every term in every store ρ^{sym} that arises during specialization of *B* can be simplified to have the form $\boldsymbol{c}$, or $\boldsymbol{m} + \boldsymbol{c}$, where $\boldsymbol{c}$ is an integer constant, and $\boldsymbol{m}$ is a symbolic constant.

If a binary *B* violates P1 or P2, *B* might have different semantics for different layouts of the stack and heap blocks. For example, consider the code snippet

```
1: mov [esp+1000],10 ;S
2: mov [esp*2],20 ;S
3: mov eax,[esp+1000] ;L
```

This snippet violates P2: after symbolically executing instruction `2`, we obtain the store ρ_2^{sym}

$$\rho_2^{sym} \equiv \langle[\mathit{ESP} \mapsto \boldsymbol{n}], [\,], [\boldsymbol{n} + \boldsymbol{1000} \mapsto \boldsymbol{10}][\boldsymbol{n} * \boldsymbol{2} \mapsto \boldsymbol{20}]\rangle$$

In ρ_2^{sym}, the term $\boldsymbol{n} * \boldsymbol{2}$ cannot be rewritten in the form $\boldsymbol{m} + \boldsymbol{c}$. Note that the value in the *EAX* register after (concretely) executing instruction `3` depends on the initial value of *ESP*. If we execute the snippet with the initial concrete store $\rho_0 \equiv \langle[\mathit{ESP} \mapsto \mathit{1000}], [\,], [\,]\rangle$, the value in *EAX* after executing instruction `3` will be *20*. If we execute the code snippet with a different value for *ESP* in ρ_0, the value in *EAX* will be *10*. Consequently, the snippet loads different values into *EAX* for different stack layouts.

Binaries of memory-safe programs produced by a standard compiler are usually $\mathbf{PE}_2$-safe. Apart from P1 and P2, we will also require that allocated memory chunks do not overlap in the address space used at run-time. Allocated memory chunks also include the chunk in which the residual code resides. (See §4.4.)

4.3 Interprocedural Partial Evaluation

In this section, we present the extensions to the algorithm to handle binaries with multiple procedures.

4.3.1 BTA

For interprocedural partial-evaluation, WIPER uses a monovariant division, i.e., the classification of each instruction as static/dynamic holds for *all* calling contexts. This approach to BTA introduces the two issues discussed below, which will be illustrated using the code snippet shown in Fig. 15. Procedure `main` contains two calls to `foo`. The actual parameter of the first call is static, and that of the second call is dynamic. `foo` merely returns the formal parameter. (For possible future work on polyvariant BTA, see §8.)

Algorithm 8 specializeFn

Input: CFG, ρ, BTAMap, specializedCFGs
Output: ρ'
1: **if** $\langle$CFG, $\rho\rangle \in$ specializedCFGs **then**
2: **return** $\langle$specializedCFGs[$\langle$CFG, $\rho\rangle$], specializedCFGs$\rangle$
3: **end if**
4: **if** `Recursive`(CFG) **then**
5: specializedCFGs[$\langle$CFG, $\rho\rangle$] $= \rho$ // See explanation in the text
6: **end if**
7: $\langle\rho'$, specializedCFGs$\rangle \leftarrow$ IntraPE(CFG, ρ, BTAMap, specializedCFGs)
8: specializedCFGs[$\langle$CFG, $\rho\rangle$] $= \rho'$
9: **return** $\langle\rho'$, specializedCFGs$\rangle$

Parameter mismatches. A forward slice can include multiple calls to the same procedure, with different subsets of actual parameters at different call-sites. However, the slice contains the union of the corresponding formal-parameter sets, which causes a mismatch between the actual parameters at a call-site and the procedure's formal parameters [12, 24].

For the moment, assume that the forward slice used by BTA is context-sensitive. In the code snippet shown in Fig. 15, suppose that the slice includes instructions `A2`, `C2`, `R2`, and the entire body of `foo`. Note that there is a mismatch between the actual parameter `A1`, and `foo`'s formal parameter. (The latter is in the slice, but the former is not.) WIPER lifts instructions A1 and C1 because they are interprocedural predecessors of `foo`'s instructions; this step ensures that there will be no parameter mismatches in the residual code.

Return mismatches. If the forward slice used during BTA is context-sensitive, there is also a return mismatch between the return value of `foo`, and the use of the return value at instruction `R1`. The former is in the slice, but the latter is not, which violates congruence. (At instruction `R1`, the specializer expects to find a return value for `foo` in the static store, but there is no value.) To prevent return mismatches, WIPER uses context-insensitive slicing, which causes instruction `R1` to be included in the slice.

In summary, for interprocedural BTA, WIPER (i) lifts interprocedural static-predecessors of dynamic instructions, and (ii) uses context-insensitive forward slicing. In addition, to ensure that partial evaluation always terminates, WIPER conservatively classifies all instructions in a recursive procedure as dynamic. (This point is discussed further in §4.3.2.)

4.3.2 Specialization

To perform specialization for interprocedural partial-evaluation, the following lines are added after line [16] of Alg. 3. `IsCall` returns true if the argument is a call instruction; `Callee` returns the callee CFG for a call instruction.

if `IsCall`(i) **then**
$\langle\rho$, specializedCFGs$\rangle \leftarrow$SpecializeFn(`Callee`(i), ρ, BTAMap, specializedCFGs)
endif

In addition, Alg. 3 takes an additional argument, specializedCFGs, which is a map whose entries are of the form $\langle$CFG, $\rho\rangle \mapsto \rho'$, where ρ is a partial static pre-store and ρ' is a partial static post-store. Alg. 3 also returns specializedCFGs as an additional return value. Procedure SpecializeFn is given as Alg. 8. Alg. 8 implements function caching—if CFG has already been specialized with respect to ρ, Alg. 8 returns ρ' (lines 1–3).

To ensure that partial evaluation always terminates, WIPER does not attempt to specialize recursive functions. Recall from §4.3.1 that all instructions in a recursive procedure are classified as dynamic (and the binding-time classification computed by BTA is congruent with respect to that classification). Thus, for a recursive procedure P, the correct values for all variables that are classified static at the return from P are found in the input partial static pre-store ρ. Consequently, Alg. 8 can use ρ as the partial static post-store in the entry added to specializedCFGs before IntraPE is called (lines 4–6). This trick ensures that Alg. 8 cannot get into an infinite loop while specializing a recursive function.

If a function has not already been specialized, Alg. 8 calls Alg. 3 to specialize the CFG with respect to ρ, and adds the partial static post-store to the map (lines 7–8).

4.4 Correctness

In this section, we present two theorems concerning: (i) termination of WIPER, and (ii) correctness of the residual code produced by WIPER. The first theorem applies to binaries with instructions from the entire IA-32 instruction set. Because it would be difficult to prove the correctness theorem for the entire IA-32 ISA, we prove correctness only for binaries containing instructions from a small instruction subset. (See App. A.1.)

Theorem 1. *Suppose that B is a* ***PE$_2$-safe*** *binary, and ρ_S^{sym} is a partial static symbolic-store. Then* WIPER(B, ρ_S^{sym}) *terminates.*

Proof. WIPER treats recursive functions conservatively, so WIPER cannot enter into an infinite loop while attempting to specialize a recursive function. WIPER unrolls static loops finitely many times depending upon the static loop bound. Moreover, the forward slice performed during BTA follows both data and control dependences, which rules out the possibility of there being a static instruction that is control dependent on a dynamic branch. Consequently, WIPER cannot create infinitely many specialized versions of the same basic block. Thus WIPER(B, ρ_S^{sym}) is guaranteed to terminate. □

In a standard semantics that formalizes the behavior of an IA-32 processor, an evaluation function would take an IA-32 *Store* as an input, and produce an IA-32 *Store* as an output. In §3, we introduced the notion of a state σ consisting of an environment η and an IA-32 store ρ. In this section, we use a non-standard semantics that uses an environment along with an IA-32 store. The environment is really a ghost variable

that makes explicit the starting address of the stack and heap blocks, while the concrete part of the state is the store.

Based on the congruent division established by BTA, a state can be partitioned into η_S, ρ_S, η_D, and ρ_D. Thus the type for the evaluation function $[\![\cdot]\!]$ (the function that evaluates an instruction or a sequence of instructions with respect to a state, and returns the post-state) can be written as

$$[\![\cdot]\!] : \text{Instruction} \times \mathit{Env}_S \times \mathit{Store}_S \times \mathit{Env}_D \times \mathit{Store}_D \rightarrow \mathit{Env}_S \times \mathit{Store}_S \times \mathit{Env}_D \times \mathit{Store}_D.$$

If we execute an instruction in a partially evaluated binary, ρ_S is never accessed or updated. Thus we will overload $[\![\cdot]\!]$ to also denote the meaning function for instructions in a partially evaluated binary:

$$[\![\cdot]\!] : \text{Instruction} \times \mathit{Env}_S \times \mathit{Env}_D \times \mathit{Store}_D \rightarrow \mathit{Env}_S \times \mathit{Env}_D \times \mathit{Store}_D$$

It will always be clear from context which meaning of $[\![\cdot]\!]$ is intended.

We assume that the memory map for a program that has been partially evaluated is equipped with a special area in which the starting addresses of the stack and heap blocks allocated by lifted instructions are stored. Thus, strictly speaking, the memory maps for a binary B and some residual binary B' of B will be different. However, the only differences are in the special area, and we denote equality of two stores ρ and $\overline{\rho}$ *modulo the special area* by $\rho \approx \overline{\rho}$.

For an input binary B that is $\mathbf{PE}_2$-safe, and an IA-32 state σ that can be partitioned into η_S, ρ_S, η_D, and ρ_D based on a division of inputs to B, WIPER produces a specialized binary that, when executed, produces an answer that matches the answer that would be produced by B (modulo the special area). The precise statement of the property is given in the following theorem:

Theorem 2. *Suppose that B is $\mathbf{PE}_2$-safe. Also suppose that η_S is an environment such that there are no overlaps among (a) any of the chunks in Dom(η_S), (b) the chunk in which the residual code produced by* WIPER *is loaded, and (c) any of the chunks allocated during the execution of the residual code. For all $\sigma = (\eta_S, \rho_S, \eta_D, \rho_D)$ such that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D)$ terminates, suppose that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta'_S, \rho'_S, \eta'_D, \rho'_D)$. If ρ_S^{sym} is a symbolic store such that $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, then $[\![\text{WIPER}(B, \rho_S^{sym})]\!](\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \overline{\rho}'_D)$, where $\overline{\rho}'_D \approx \rho'_D$.*

Proof. The proof for Thm. 2 is given in App. A.1. □

Thm. 2 does not include ρ_S while comparing states because the residual code produced by $\text{WIPER}(B, \rho_S^{sym})$ does not contain static instructions, and consequently never accesses or updates the static store ρ_S.

4.5 Threats to Validity

There are two threats to the validity of our algorithms.

1. If the input binary is not $\mathbf{PE}_2$-safe, the residual code might violate Thm. 2.
2. The accuracy of BTA depends on the accuracy of the SDG for the binary. If the methods used to construct the SDG are overly conservative, BTA can classify instructions as "dynamic" that do not depend on dynamic inputs. If the methods used to construct the SDG are under-approximative, BTA can classify instructions as "static" that actually do depend on dynamic inputs. In our experiments, we observed the former behavior. The SDG that WIPER uses is built using "variable-like abstractions" recovered by machine-code variable-recovery analyses [9]. Each recovered variable-like abstraction could be imprecise—it is often an agglomeration of some source-code variables. Consequently, the SDG recovered from the program binary could be more imprecise than an SDG that is obtained from the program's source code. As a result, WIPER's BTA classifies more instructions dynamic than an analogous BTA that starts from source code.

5. Implementation

WIPER uses CodeSurfer/x86 [10] to obtain the SDG, CFG, and call graph for the binary. WIPER's BTA uses CodeSurfer/x86's forward-slicing operation. WIPER's specializer uses CodeSurfer/x86's rewriting API to create the sections, CFGs, and instructions in the residual binary. WIPER uses Transformer Specification Language (TSL) [29] to build its concrete and symbolic interpreters. The concrete operational semantics of the integer subset of IA-32 is written in TSL, and the semantics is interpreted for concrete evaluation, and reinterpreted for symbolic evaluation. WIPER also uses TSL [30] to convert an instruction into a QFBV formula. WIPER uses MCSYNTH [46] to synthesize an IA-32 instruction sequence from a QFBV formula. MCSYNTH sometimes requires a set of scratch registers to hold results of intermediate calculations in the residual code. WIPER supplies dead registers to MCSYNTH to be used as scratch registers. If the number of dead registers at a point is not sufficient, WIPER supplies global addresses that are unused in the residual program as scratch locations to MCSYNTH.

In the examples presented in this paper, we have treated memory as if each memory location holds a 32-bit integer. However, our implementation supports the actual IA-32 memory model, in which each memory location holds an 8-bit integer.

The compiler might add instructions at the beginning of each procedure so that stack variables are aligned on 4-byte, 8-byte, or 16-byte boundaries. For example, consider the code snippet shown in Fig. 16. In procedures `foo` and `main`, the compiler aligns the stack variables on a 16-byte boundary. To accommodate such alignment, we relax P2 of

Table 1: Characteristics of applications for optimization via specialization.

Application	Domain	LOC	Description	Static Input
Power	Mathematical library functions	19	Computes x^n	$n = 100$
Dotproduct	Linear algebra operations	29	Computes the dot product of two vectors with n dimensions	$n = 100$, and co-efficients of the first vector
Interpreter	Language interpreters	71	Interpreter for the minimalist language "Brainf*ck"	Input program
Filter	Image processing	107	Applies a convolution filter of size $m \times m$ on an image of size $n \times n$	$m = 3$, $n = 3$, and elements of the filter
uuencode	Text utilities	129	Base-64 encodes a string of size n	$n = 256$, and the string
SHA1	Cryptographic operations	140	Computes the sha1 digest of a message of size n bits	$n = 1024$, and contents of the first 512 bits
uudecode	Text utilities	167	Decodes a base-64-encoded string of size n	$n = 256$, and the encoded string

```
foo:                          main:
  push ebp                      push ebp
  mov ebp,esp                   mov ebp,esp
  and esp,0xFFFFFF00            and esp,0xFFFFFF00
  sub esp,170                   sub esp,150
        :                             :
```

Figure 16: Code snippet to illustrate variable alignment.

PE$_2$-safety properties to allow terms of the form $\boldsymbol{m} + \boldsymbol{p}_0 + \boldsymbol{p}_1 + \ldots + \boldsymbol{c}$, where $\boldsymbol{m}$ is a symbolic constant representing the specialization-time starting address of the stack or a heap block M, $\boldsymbol{c}$ is an integer constant, and $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, etc. are symbolic constants that represent the alignment adjustments performed by the procedures Proc0, Proc1, ... on the current call stack.

If the binary is statically linked, WIPER does not need any additional input from the user for partial evaluation. If the binary is not statically linked, the user needs to provide a model for each library function as additional inputs to WIPER. A model consists of the following information: (i) an input-output dependence relation, and (ii) a procedure to compute return values of library-function calls that are classified as static. WIPER specializes library-function calls in a bimodal manner: if any argument to a library-function call is dynamic, all of the static arguments are lifted along with the call; if all arguments are static, WIPER uses the user-supplied model to compute a return value for the call (i.e., a fully static call is specialized away.)

We are generally unable to use WIPER's fully automated end-to-end partial-evaluation algorithm to extract components from a large binary. Binaries of large applications do not have clearly demarcated inputs. They often have only one input—a buffer that stores the entire command line. The binary interprets the contents of the buffer to assign values to variables. If the buffer is classified dynamic, Alg. 2 classifies all instructions in the binary dynamic.

For component extraction, we run WIPER in an alternative experimental mode, called Component Extraction (CE) mode. In CE-mode, WIPER performs a restricted BTA. (The specialization algorithm is the same in CE and non-CE modes.) In CE-mode, the user specifies a subset I_S of instructions in the original binary to be treated as static instructions. The remaining instructions in the binary are treated as dynamic instructions. WIPER identifies lifted instructions, and then specializes the binary using WIPER's original specialization algorithm.

CE-mode is related to *generalized partial computation* [22], in which a program is specialized with respect to constraints. As will become clear below, in CE-mode we are specializing a program P with respect to a condition, such as $\varphi \equiv (\boldsymbol{EAX} = \boldsymbol{1})$, which must hold at a given point p in the middle of the program. The goal is to produce a residual program that is (i) a specialization of P, in which (ii) φ holds each time the points residuated from p are reached.

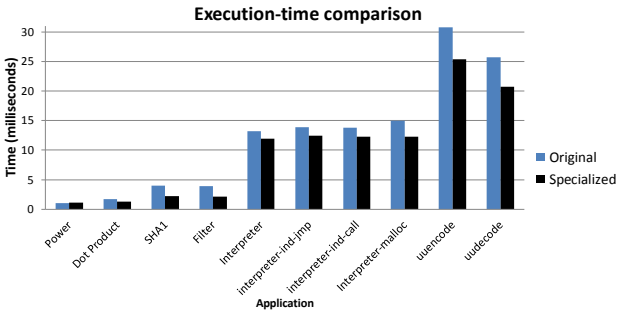

Figure 17: Comparison of the execution times of the original and specialized binaries.

For simple examples of the kind discussed below, CE-mode successfully extracts components, even though congruence is violated. It remains for future work to devise a theory of program specialization that covers CE-mode.

6. Experiments

The goal of our experiments was to test and evaluate the two primary use cases of WIPER: (i) optimization via specialization, and (ii) component extraction. Our experiments were run on a system with a 64-core, 2.3 GHz AMD Opteron 6376 processor; however, WIPER's algorithm is single-threaded. The system has 264 GB of memory and runs RHEL 6.6.

6.1 Optimization via Specialization

For this use case, we wanted to answer the following question:

- What is the speedup produced by partial evaluation for different applications?

For the first set of experiments, we used a suite of applications from application domains that were suitable for partial evaluation. Table 1 presents the characteristics of the applications. The first two applications listed in the table have been used in prior work [34, 41]. There are four versions of the interpreter application in our test suite: (i) a standard version, (ii) a version that uses indirect jumps (via switch statements), (ii) a version that uses indirect calls (via function pointers), and (iv) a version that uses heap-allocated storage. uuencode and uudecode are FreeBSD utilities [3]. We used CE-mode for partially evaluating the FreeBSD utilities. The

Table 2: Characteristics of applications for component extraction.

Application	KLOC	Description	Component
b64	0.544	Base64 encoder-decoder	decoder
lzfx	0.914	LZF compression-decompression utility	compress
bzip2	7	Compression-decompression utility	compress

Table 3: Comparison of sizes and number of procedures in original binary and extracted component.

Application	Size of binary (KB)	No. of procs. in binary	Size of component (KB)	No. of procs. in component
b64	5.4	16	5.9	4
lzfx	7.6	24	6	8
bzip2	90.2	117	64.6	78

results are shown in Fig. 17. For these applications, the average speedup produced by partial evaluation, computed via the geometric mean, is 1.3.

For Power, partial evaluation results in a slight slowdown. Power has a tight loop, and we believe that the aggressive loop unrolling caused by partial evaluation disrupts instruction caching.

At the other end of the spectrum, we measured a speedup in Filter of 1.9. Filter has four nested for-loops. The inner two loops iterate over the filter elements. Partial evaluation unrolls all four loops, and—because the filter elements are static—replaces almost all instructions in the bodies of the inner two loops with just a few residual instructions.

6.2 Component Extraction

For this use case, we wanted to answer the following questions:

- How can we extract an executable component from a binary via partial evaluation?
- How does the size of the extracted component compare to the size of the original binary?

For this set of experiments, we used three applications that bundle several components into a bloated executable. Table 2 presents the characteristics of the applications. The first and second applications were obtained from Google Code [1] and SourceForge [2], respectively.

CE-mode of WIPER was used for this set of experiments. The sizes of the original binary and the extracted components are shown in Table 3.

b64. The decoder extracted from b64 is bigger than the b64 binary because our current implementation does not optimize the layout of basic blocks in the residual code to reduce the number of jump instructions. For example, b64 has 30 `jmp` instructions, and the extracted decoder has 164 `jmp` instructions. The issues in partially evaluating `b64` are similar to those of `bzip2`, which is discussed below.

lzfx. `lzfx` is a text-compression utility. In this case study, we present how WIPER extracts the compression component of `lzfx`. The decision about whether to call `fx_create` (which compresses) or `fx_read` (which decompresses) is made in the following code snippet from the binary:

```
_text:080499D4   test  eax,eax ;S
_text:080499D6   jnz  loc_80499E2 ;S
_text:080499D8   mov  dword [esp+44],0 ;S
                          :
_text:08049A3A   mov  eax,dword [esp+44] ;L
_text:08049A3E   mov  dword [esp+12],eax ;D
                          :
_text:08049A5E   call  fx_create ;D
                          :
loc_80499E2:
                          :
_text:08049A84   call fx_read; D
```

If the value in the `eax` register is 0 at instruction `80499D4`, `lzfx` calls `fx_create`. The user specifies that instructions `80499D4` through `8049A3A` should be treated as static instructions. WIPER lifts instruction `8049A3A` because it supplies a static value to the dynamic instruction `8049A3E`. WIPER performs specialization and evaluates the static and lifted instructions with respect to the partial store

$$\rho \equiv \langle [EAX \mapsto 0][ESP \mapsto \boldsymbol{n}], [\,], [\,] \rangle,$$

and residuates the instruction `mov eax,0` for the lifted instruction `8049A3A`, and emits the remaining dynamic instructions. Because `eax` is `0` at instruction `80499D4`, the fall-through branch is taken at `80499D6`, which forces WIPER to enter `fx_create` and to bypass `fx_read`. Consequently, WIPER residuates an executable that only performs compression.

With `lzfx`, we also went further by embedding the extracted compression component in a different application. We created a small header file with the signature of `fx_create`, and linked the new application with the compression component extracted by WIPER.

bzip2. `bzip2` is a large application that takes several auxiliary command-line inputs in addition to one that specifies whether to perform compression or decompression. Ultimately, the decision about whether to call `compress` or `uncompress` is made in the following code snippet of the binary:

```
_text:0805B539   mov eax,[esp+40] ;S
_text:0805B53E   cmp  eax,1 ;S
_text:0805B541   jnz  loc_805B5DB ;S
                          :
_text:0805B558   call  compress ;D
                          :
loc_805B5DB:
                          :
_text:0805B5E4   call  uncompress ;D
```

The memory location whose address is $ESP + 40$ holds a value computed from the compression/decompression flag on the command line. If the value is 1, `bzip2` calls `compress`; otherwise, it calls `uncompress`.

The user specifies that instructions `805B539` through `805B541` should be treated as static instructions. WIPER performs specialization and evaluates the aforementioned instructions with respect to the partial store

$$\rho \equiv \langle [ESP \mapsto \boldsymbol{n}], [\,], [\boldsymbol{n + 40} \mapsto \boldsymbol{1}] \rangle.$$

This partial store forces WIPER to enter `compress` and bypass `uncompress`, and consequently WIPER residuates an executable that only performs compression.

7. Related Work

Partial evaluation. Partial evaluation has been used as a general framework for the specialization of programs belonging to different languages, and different domains. In addition to several functional languages [25, 32], partial evaluation has been used to specialize a flow-chart language [25], C [6, 15], FORTRAN [26], and Java [40, 41]. Partial evaluators can either be *on-line* (performed in a single phase) [23], *off-line* (performed in two phases), or hybrid [41]. WIPER's high-level architecture resembles that of an off-line partial evaluator for an imperative language. However, the need to handle several machine-code-specific issues makes WIPER's algorithm significantly different from those of source-code partial evaluators.

Run-time specialization techniques. Run-time specialization (or dynamic compilation) generates optimized code during program execution by partially evaluating user-annotated regions of the program with respect to invariant data computed at run time [8, 14, 19, 28].

Like WIPER, these tools perform specialization on machine code. However, the partial-evaluator-like component in such tools is part of the compiler. BTA is performed with respect to source-code variables, and carried out on an IR that is constructed from source code. Run-time-specialization tools have to address an issue that does not arise in WIPER, namely, the need to create code templates that can be reused at run-time. Some tools accomplish this task by jury-rigging an existing compiler's code generator to inhibit optimizations, such as code motion, by using the `volatile` type qualifier of C.

In contrast, WIPER faced other issues, such as the need to decouple multiple updates performed by a single instruction, and to create code that allows the stack and heap to be at different positions at run-time than at specialization-time.

Specialization of Java bytecode. WIPER is not the first partial evaluator that works on low-level code. Lancet [37] performs partial evaluation of Java bytecode snippets that need to be compiled by a just-in-time (JIT) compiler. JScp [27] performs supercompilation on Java bytecode. Like WIPER, the specializers in Lancet and JScp make use of a static partial store. For Lancet and JScp, the static partial store is a Java virtual-machine (JVM) store, while WIPER's is an IA-32 store. However, while those specializers work on low-level code, WIPER works at an "even lower level." The following issues that arise in IA-32 and not in bytecode justify the novel design choices made in WIPER:

1. Because the JVM is a stack-based machine, specialization of bytecode instructions often involves loading constant values instead of locals/fields. In contrast, the operands of IA-32 instructions vary widely, and the specialization of an instruction with respect to a partial store need not be a variant of the instruction. Moreover, IA-32 has around 43,000 unique opcodes, and WIPER's specializer must be able to specialize *any* instruction with respect to *any* partial store. WIPER addressed this issue by using machine-code synthesis.
2. Address computation is not explicit in bytecode (variables are referenced using offsets). Consequently, bytecode specialization does not face the problem of residuating specialization-time addresses. However in machine code, instructions compute addresses, and a naïve specializer, such as $\mathbf{PE}_1$, is unsatisfactory. For this reason, WIPER uses symbolic techniques for specialization.

Superoptimization and link-time optimization. Superoptimization aims at finding an optimal instruction-sequence for a target instruction-sequence [11, 31, 39]. Peephole superoptimization [11] uses "peepholes" to harvest target instruction-sequences, and replace them with equivalent instruction-sequences that have a lower cost. Link-time optimizers carry out whole-program optimizations at link time [33]. While these optimization techniques optimize a binary for all possible inputs, WIPER optimizes a binary for a specific input.

8. Conclusion and Future Work

In this paper, we presented an algorithm for machine-code partial evaluation. We presented WIPER, a partial evaluator for IA-32. We used WIPER to partially evaluate the binaries of seven test applications with respect to static inputs, and obtained specialized binaries that had an average speedup of 1.3 (computed via the geometric mean). We also presented two case studies that describe how WIPER can be used to extract an executable component from a bloated binary.

One possible direction for future work is to investigate polyvariant BTA via *specialization slicing* [7]. For every call to a procedure P with a different combination of static and dynamic actual parameters, specialization slicing can be used to produce a different binding-time annotation for P's instructions. This technique could increase the number of static instructions during specialization, leading to residual code that is more specialized than the code currently residuated by WIPER. A second direction is to use liveness information to reduce the number of specialized copies of a basic block produced by WIPER.

Acknowledgments

We thank Suan Yong, Junghee Lim, Tushar Sharma, and Brian Alliet for answering several questions during the implementation of WIPER. We thank Suan Yong for implementing a skeleton binary rewriter using CodeSurfer/x86. We thank the anonymous reviewers for their valuable feedback.

References

[1] `http://www.code.google.com`.
[2] `http://www.sourceforge.net`.
[3] `http://www.opensource.apple.com/source`.
[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.
[5] L. O. Andersen. Binding-time analysis and the taming of C pointers. In *PEPM*, 1993.

[6] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Univ. of Copenhagen, 1994.
[7] M. Aung, S. Horwitz, R. Joiner, and T. Reps. Specialization slicing. *TOPLAS*, 36(2), 2014.
[8] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *PLDI*, 1996.
[9] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.
[10] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *CC*, 2005.
[11] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.
[12] D. Binkley. Precise executable interprocedural slices. *LOPLAS*, 2: 31–45, 1993.
[13] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, 2011.
[14] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL*, 1996.
[15] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. *Dagstuhl Seminar on Partial Evaluation*, pages 54–72, 1996.
[16] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *CCS*, 2011.
[17] M. Das, T. Reps, and P. van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *PEPM*, 1995.
[18] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.
[19] D. Engler, W. Hsieh, and F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *POPL*, 1996.
[20] U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Workshop on New Security Paradigms*, 1999.
[21] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3), 1987.
[22] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theor. Comp. Sci.*, 90(1), 1991.
[23] J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark*, 1998.
[24] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1), 1990.
[25] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.
[26] P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4), 1995.
[27] A. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. *Perspectives of Systems Informatics*, 5947:185–192, 2010.
[28] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI*, 1996.
[29] J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *TOPLAS*, 35(4), 2013.
[30] J. Lim, A. Lal, and T. Reps. Symbolic analysis via semantic reinterpretation. *Softw. Tools for Tech. Transfer*, 13(1), 2011.
[31] H. Massalin. Superoptimizer: A look at the smallest program. In *ASPLOS*, 1987.
[32] T. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *PEPM*, 1995.
[33] R. Muth, S. Debray, S. Watterson, and K. D. Bosschere. Alto: A link-time optimizer for the compaq alpha. *Softw. Pract. Exper.*, 31(1), 2001.
[34] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Computer Languages*, 1998.
[35] T. Reps. Program analysis via graph reachability. *Inf. and Softw. Tech.*, 40(11–12), 1998.
[36] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *FSE*, 1994.
[37] T. Rompf, A. Sujeeth, K. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision JIT compilers. In *PLDI*, 2014.
[38] H. Saïdi. Logical foundation for static analysis: Application to binary static analysis for security. *ACM SIGAda Ada Letters*, 28(1), 2008.
[39] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.
[40] U. Schultz, J. Lawall, and C. Consel. Automatic program specialization for Java. *TOPLAS*, 25(4), 2003.
[41] A. Shali and W. Cook. Hybrid partial evaluation. In *OOPSLA*, 2011.
[42] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *S&P*, 2009.
[43] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *ATC*, 2012.
[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Int. Conf. on Information Systems Security*, 2008.
[45] V. Srinivasan and T. Reps. Partial evaluation of machince code. TR-1821, University of Wisconsin–Madison Tech Report, Aug. 2015. URL `http://www.cs.wisc.edu/wpis/papers/tr1821.pdf`.
[46] V. Srinivasan and T. Reps. Synthesis of machince code from semantics. In *PLDI*, 2015.
[47] M. Weiser. Program slicing. *TSE*, SE-10(4), 1984.
[48] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *S&P*, 2015.

A. Appendix

Theorem 2. *Suppose that B is **PE_2-safe**. Also suppose that η_S is an environment such that there are no overlaps among (a) any of the chunks in Dom(η_S), (b) the chunk in which the residual code produced by* WIPER *is loaded, and (c) any of the chunks allocated during the execution of the residual code. For all $\sigma = (\eta_S, \rho_S, \eta_D, \rho_D)$ such that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D)$ terminates, suppose that $[\![B]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta'_S, \rho'_S, \eta'_D, \rho'_D)$.*
If ρ_S^{sym} is a symbolic store such that $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$, then $[\![\mathsf{WIPER}(B, \rho_S^{sym}))]\!](\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \overline{\rho}'_D)$, where $\overline{\rho}'_D \approx \rho'_D$. □

A.1 Proof of Thm. 2

We prove Thm. 2 for binaries whose instructions come from a small instruction set, TinyIA32. TinyIA32 consists of a small subset of the IA-32 instruction set, augmented with (i) an extra register, and (ii) two new instructions. We decided to exclude instructions related to branches and function calls from TinyIA32 because if we were to prove Thm. 2 for the entire IA-32 ISA, the part of the proof concerning branches and function calls would be similar to that of an analogous proof for a source-code partial evaluator. We prove Thm. 2 using induction on the length of the trace of a run of the binary, and it is straightforward to extend our trace-based proof to an ISA that includes branches and function calls.

TinyIA32 has three registers, no flags, and seven instructions. The abstract syntax of TinyIA32 is defined in Fig. 18. In an indirect operand, the register holds the address of the memory location. For instructions with two operands, the operand on the left is the destination operand. For the

$$\text{INT} ::= \{..., \texttt{-1}, 0, 1, ...\} \quad \text{Reg} ::= \texttt{r1} \mid \texttt{r2} \mid \texttt{rw1}$$
$$\text{R} ::= \text{Direct}(\text{Reg}) \quad \text{C} ::= \text{Imm}(\text{INT}) \quad \text{M} ::= \text{Indirect}(\text{R})$$
$$\text{I} ::= \texttt{mov}(\text{R}, \text{C}) \mid \texttt{mov}(\text{R1}, \text{R2}) \mid \texttt{mov}(\text{R}, \text{M}) \mid \texttt{mov}(\text{M}, \text{R}) \mid \texttt{add}(\text{R1}, \text{R2}) \mid \texttt{stackalloc}(\text{R}) \mid \texttt{heapalloc}(\text{R})$$

Figure 18: Abstract syntax of TinyIA32.

`stackalloc` instruction, the single operand is the destination operand. For the `heapalloc` instruction, the single operand is both a source and destination operand.

The register `rw1` is a *work register* that is used for holding the results of intermediate calculations in residual code. A work register is used to simulate the fact that the real WIPER uses dead registers as scratch registers for such calculations. In the rest of this section, we do not show the work register in TinyIA32 stores. Also, if a register is not used in a TinyIA32 binary, it has the default value 0.

The first `mov` instruction loads a 32-bit constant into a register. The second `mov` instruction performs a register-to-register move. The final two `mov` instructions move a 32-bit value from a memory location to a register and from a register to a memory location, respectively. The `add` instruction adds the 32-bit values in the two register operands, leaving the result in the destination-operand register. The `stackalloc`(R) instruction allocates a stack of some fixed, but arbitrarily large, size, whose starting address is available as an output in R. The `heapalloc`(R) instruction allocates a heap block whose size is provided as an input in R, and whose starting address is returned in R as the output. Because TinyIA32 has no flags, a TinyIA32 store consists of only a register map and a memory map.

We prove Thm. 2 by induction on the length n of a partial trace T of a run of binary B.

A.1.1 Base Case $n = 0$

Thm. 2 trivially holds on a partial trace with no instructions.

A.1.2 Induction Hypothesis

Let us assume that Thm. 2 holds on a partial trace $T \equiv I_1, I_2, \ldots, I_n$ with n instructions. Suppose that the corresponding residual trace produced by WIPER for T is $T' \equiv J_1, J_2, \ldots, J_n$, where J_i is the residual instruction sequence produced by the Specialize phase of WIPER for I_i. (Note that J_i can be an empty sequence of instructions or can consist of several instructions.) Suppose that the initial state is $(\eta_S, \rho_S, \eta_D, \rho_D)$. (Recall from §3 that η maps a symbolic constant denoting the starting address of the stack or a heap block to a concrete address.) As the induction hypothesis, we assume the following:

$$\begin{aligned} &[\![I_1, I_2, \ldots, I_n]\!](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) \\ \wedge\ &[\![J_1, J_2, \ldots, J_n]\!](\eta_S, \eta_D, \rho_D) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n) \\ \wedge\ &\overline{\rho}_D^n \approx \rho_D^n \end{aligned} \tag{7}$$

To account for the actions of WIPER on static and lifted instructions, we need to introduce yet one more ghost variable, *ghostStaticStore* (*gSS*). At each point k in the trace T', this variable holds the static symbolic store that was used by WIPER to create the next residual instruction-sequence in T', i.e., $J_{k+1} = \text{Specialize}(I_{k+1},\ gSS^k)$.

We strengthen the induction hypothesis with the following conjunct:

$$[\![gSS^n]\!]\eta_S^n = \rho_S^n, \tag{8}$$

where the left-hand side denotes the simplification of each symbolic expression in *gSS* with respect to the values obtained from η_S^n. Because gSS^0 is ρ_S^{sym}, by the hypothesis $[\![\rho_S^{sym}]\!](\eta_S) = \rho_S$ in Thm. 2, we have $[\![gSS^0]\!](\eta_S) = \rho_S$, and thus Eqn. (8) holds for the base case.

A.1.3 Induction Step

We now prove Thm. 2 on a partial trace $I_1, I_2, \ldots, I_{n+1}$ with $n+1$ instructions. That is, the goal is to show

$$\begin{aligned} &[\![I_1, I_2, \ldots, I_{n+1}]\!](\eta_S, \rho_S, \eta_D, \rho_D) \\ &\qquad = (\eta_S^{n+1}, \rho_S^{n+1}, \eta_D^{n+1}, \rho_D^{n+1}) \\ \wedge\ &[\![J_1, J_2, \ldots, J_{n+1}]\!](\eta_S, \eta_D, \rho_D) \\ &\qquad = (\eta_S^{n+1}, \eta_D^{n+1}, \overline{\rho}_D^{n+1}) \end{aligned} \tag{9}$$

$$\wedge\ \overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$$

$$\wedge\ [\![gSS^{n+1}]\!]\eta_S^{n+1} = \rho_S^{n+1}. \tag{10}$$

Some additional intuition about the induction hypothesis can be obtained from the following more precise restatement of reparenthesization (5):

$l : (\eta_S, \rho_S, \eta_D, \rho_D) \longrightarrow (l, gSS) : (\eta_S, \eta_D, \rho_D)$
where $\rho_S = [\![gSS]\!]\eta_S$.

There are three subcases, for I_{n+1} being static, lifted, or dynamic.

A.1.3.1 Static instructions. Because static-allocation sites are lifted—see line [10] of Alg. 2—a static instruction cannot have the form `stackalloc`(R) or `heapalloc`(R). The argument for each of the static-instruction cases for I_{n+1} has the following form: Suppose that instruction I_{n+1} is static. Then

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^{n+1}, \eta_D^n, \rho_D^n), \tag{11}$$

where some update is made to ρ_S^n to create ρ_S^{n+1}.

$$\text{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \epsilon,$$

where ϵ denotes the empty sequence of instructions. Because J_{n+1} is empty, we have

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n). \tag{12}$$

Eqns. (11) and (12), together with Eqn. (7), show that Eqn. (9) holds.

The congruence property of BTA ensures that WIPER can completely evaluate I_{n+1} at specialization time, and gSS^n is updated by WIPER to produce gSS^{n+1} in the same way as ρ_S^n is updated by $[\![I_{n+1}]\!]$ to produce ρ_S^{n+1}. Moreover, because η_S^n is not updated by $[\![I_{n+1}]\!]$, by Eqn. (8) we have $[\![gSS^{n+1}]\!]\eta_S^n = \rho_S^{n+1}$, and Eqn. (10) holds. Consequently, the induction hypothesis holds on a trace with $n+1$ instructions, where the $(n+1)^{st}$ instruction is static.

A.1.3.2 Lifted and dynamic instructions. We now make an observation that is relied on in many of the cases—namely, because static-allocation sites are lifted, if instruction I_i in T is an allocation instruction, then J_i in T' also contains an allocation instruction, i.e., the allocations in the two traces are in lock-step. For purposes of the proof, we can assume that B and WIPER(B, ρ_S^{sym}) are run in environments that will perform storage allocation in the same way. Consequently, if I_i allocates the stack or a heap block M at address m, an instruction in J_i also allocates M at m. In the respective semantics, such allocations will be recorded by updating η to $\eta[\boldsymbol{m} \mapsto m]$, where $\boldsymbol{m}$ is a symbolic constant that denotes the starting address of M. (Lifted allocation instructions update η_S and η_D; dynamic allocation instructions update η_D.)

Dynamic instructions. The argument for each of the dynamic-instruction cases for I_{n+1}, except `stackalloc` and `heapalloc`, has the following form: Suppose that instruction I_{n+1} is dynamic. Then

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^{n+1}), \qquad (13)$$

where some update is made to ρ_D^n to create ρ_D^{n+1}.

$$\mathsf{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = I_{n+1}$$

Because J_{n+1} is the same as I_{n+1}, we have

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^{n+1}), \qquad (14)$$

where, $\overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$. Eqns. (13) and (14), together with Eqn. (7), show that Eqn. (9) holds.

The static components of the state, η_S^n, gSS^n, and ρ_S^n, are left unchanged, so Eqn. (10) holds.

The cases for `stackalloc` and `heapalloc` are similar, except that those instructions update η_D^n in addition to ρ_D^n. Moreover, because of our assumption about the storage allocator, the updates to η_D^n by $[\![I_{n+1}]\!]$ and $[\![J_{n+1}]\!]$ are identical, and the updates to ρ_D^n by $[\![I_{n+1}]\!]$ and $[\![J_{n+1}]\!]$ are also identical. Consequently, the induction hypothesis holds on a trace with $n+1$ instructions, where the $(n+1)^{st}$ instruction is dynamic.

Lifted instructions. The semantics we use for evaluating static and dynamic instructions is fairly standard—-static instructions access and update ρ_S, and dynamic instructions access and update ρ_D (in addition to updating η_D). However, the evaluation of lifted instructions is non-standard: lifted instructions were originally classified static, so they must be evaluated just like static instructions; however, they also take "snapshots" of the current static state and dump them into η_D and ρ_D for use by downstream dynamic instructions.

Lifted instructions access ρ_S, but produce values that might be consumed by dynamic successors, so they must update ρ_D. In addition, lifted instructions can also have static successors, so they must also update ρ_S. Thus, in the semantics used for the original program, a lifted instruction performs the same update on both ρ_S and ρ_D. In the semantics used for the residual program, it updates only ρ_D because ρ_S is not present in states. However, for the purposes of this proof, an appropriate update is performed on *gSS*.

Similarly, if a lifted instruction allocates memory, both static and dynamic instructions downstream might access or update locations in the allocated memory. Thus in both semantics, if a lifted instruction allocates memory, it performs identical updates on η_S and η_D.

The non-overlap hypothesis of Thm. 2, together with property P2 of $\mathbf{PE}_2$-safety, imply that each specialization-time address can be represented canonically by a term of the form $\boldsymbol{m} + c$, where $\boldsymbol{m}$ is the symbolic constant that denotes the starting address m of the stack or a heap block M, and c is a constant offset.

The actual symbolic constants used are essentially arbitrary (as long as each new symbolic constant is fresh). However, for the purposes of the proof, we need to relate the symbolic constants used by the original program to those in the residual program. Thus, for the purposes of the proof, we assume that when WIPER partially evaluates a lifted instruction I_i in T that allocates memory, the symbolic constant $\boldsymbol{m}$ that should be used in *gSS* to denote the starting address of the allocated stack or heap block is supplied by an oracle. This oracle ensures that the *same* symbolic constant $\boldsymbol{m}$ gets added to the ghost variables η_S and η_D when I_i and the allocation instruction in J_i are evaluated. (If the oracle did not ensure this concordance, we cannot establish the equality in Eqn. (8).) WIPER binds $\boldsymbol{m}$ to a dedicated global address `M-addr`, and uses `M-addr` in the residual code to save the concrete starting address m of the allocated memory. (`M-addr` is located in the special area in the memory map of the residual program discussed in §4.4. Recall from §4.2.2.3 that WIPER uses a memory-layout map μ to bind a symbolic constant to a dedicated location.)

We handle the lifted-instruction cases according to the form of the instruction. In the cases below, we abbreviate an update to the register map as $\rho[R1 \mapsto v]$ rather than ***let*** $\langle rm, mm\rangle = \rho$ ***in*** $\langle rm[R1 \mapsto v], mm\rangle$, and similarly use $\rho[a \mapsto v]$, to denote an update to the memory map at address a. Which kind of map-update operation is intended should be clear from context. We present the most important cases below. The remaining cases can be found in an extended version of the paper [45].

1. I_{n+1} = `mov` (Direct(R1), Imm(v)):

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto v], \eta_D^n, \rho_D^n[R1 \mapsto v]) \qquad (15)$$

$$\mathsf{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \qquad (16)$$

Eqns. (15), (16), and (7) show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \qquad (17)$$

$$\eta_S^{n+1} = \eta_S^n \text{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto v] \text{ (From (15))} \qquad (18)$$

Eqns. (17), (18), and (8) show that Eqn. (10) holds.

2. I_{n+1} = **`mov` *(Direct(R1), Direct(R2))*:** There are two cases that can arise.

(a) Suppose that $\rho_S^n(R2) = u$ and $gSS^n(R2) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[R1 \mapsto u], \eta_D^n, \rho_D^n[R1 \mapsto u]) \quad (19)$$

$$\text{specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{mov r1,v}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto v]) \quad (20)$$

$$u = \rho_S^n(R2) = ([\![gSS^n]\!]\eta_S^n)(R2)\ (By\ Eqn.\ (8))$$
$$= ([\![gSS^n(R2)]\!]\eta_S^n) = [\![v]\!]\eta_S^n = v \quad (21)$$

Eqns. (19)–(21), and (7) show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto v] \quad (22)$$

$$\eta_S^{n+1} = \eta_S^n \text{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u]\ (From\ (19)) \quad (23)$$

Eqns. (21)–(23), and (8) show that Eqn. (10) holds.

(b) Suppose that $\rho_S^n(R2) = u$ and $gSS^n(R2) = \boldsymbol{m} + c$.

$$\text{specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c; add r1,rw1; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[R1 \mapsto m+c]) \quad (24)$$

$$u = \rho_S^n(R2) = ([\![gSS^n]\!]\eta_S^n)(R2)\ (By\ Eqn.\ (8))$$
$$= ([\![gSS^n(R2)]\!]\eta_S^n) = [\![\boldsymbol{m} + c]\!]\eta_S^n = m + c \quad (25)$$

Eqns. (19), (24), (25), and (7) show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto \boldsymbol{m} + c] \quad (26)$$

$$\eta_S^{n+1} = \eta_S^n \text{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto u]\ (From\ (19)) \quad (27)$$

Eqns. (25)–(27), and (8) show that Eqn. (10) holds. Note that J_{n+1} has the `mov rw1,0` instruction at the end of the residual code to restore the default value 0 in work register `rw1`.

3. I_{n+1} = **`mov` *(Indirect(R1), Direct(R2))*:** Because *R1* must hold an address, there are two cases that can arise.

(a) Suppose that $\rho_S^n(R1) = a$ and $\rho_S^n(R2) = u$, and $gSS^n(R1) = \boldsymbol{m} + c$ and $gSS^n(R2) = v$.

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n, \rho_S^n[a \mapsto u], \eta_D^n, \rho_D^n[a \mapsto u]) \quad (28)$$

$$\text{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M-addr; mov r1,[rw1]; mov rw1,c; add r1,rw1; mov [r1],v; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[m+c \mapsto v]) \quad (29)$$

Similar to the proof of Eqn. (21), we can prove that $u = v$. Similar to the proof of Eqn. (25), we can prove that $a = m + c$. These equalities, together with Eqns. (28), (29), and (7), show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[\boldsymbol{m} + c \mapsto v] \quad (30)$$

$$\eta_S^{n+1} = \eta_S^n \text{ and } \rho_S^{n+1} = \rho_S^n[a \mapsto u]\ (From\ (28)) \quad (31)$$

The equalities $u = v$ and $a = m + c$, along with Eqns. (30), (31), and Eqn. (8), show that Eqn. (10) holds.

(b) Suppose that $\rho_S^n(R1) = a$ and $\rho_S^n(R2) = u$, and $gSS^n(R1) = \boldsymbol{m}_1 + c_1$ and $gSS^n(R2) = \boldsymbol{m}_2 + c_2$.

$$\text{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{mov rw1,M1-addr; mov r1,[rw1]; mov rw1,c1; add r1,rw1; mov rw1,M2-addr; mov r2,[rw1]; mov rw1,c2; add r2,rw1; mov [r1],r2; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n, \eta_D^n, \overline{\rho}_D^n[m_1+c_1 \mapsto m_2+c_2]) \quad (32)$$

Similar to the proof of Eqn. (25), we can prove that $a = m_1 + c_1$ and $u = m_2 + c_2$. These equalities, together with Eqns. (28), (32), and (7), show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[\boldsymbol{m}_1 + c_1 \mapsto \boldsymbol{m}_2 + c_2] \quad (33)$$

$$\eta_S^{n+1} = \eta_S^n \text{ and } \rho_S^{n+1} = \rho_S^n[a \mapsto u]\ (By\ Eqn.\ (28)) \quad (34)$$

The equalities $a = m_1 + c_1$ and $u = m_2 + c_2$, along with Eqns. (33), (34), and Eqn. (8), show that Eqn. (10) holds.

4. I_{n+1} = **`stackalloc` *(Direct(R1))***

$$[\![I_{n+1}]\!](\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n) = (\eta_S^n[\boldsymbol{m} \mapsto m], \rho_S^n[R1 \mapsto m], \eta_D^n[\boldsymbol{m} \mapsto m], \rho_D^n[R1 \mapsto m]) \quad (35)$$

$$\text{Specialize}(I_{n+1},\ gSS^n) \equiv J_{n+1} = \texttt{stackalloc r1; mov rw1,M-addr; mov [rw1],r1; mov rw1,0}$$

$$[\![J_{n+1}]\!](\eta_S^n, \eta_D^n, \overline{\rho}_D^n) = (\eta_S^n[\boldsymbol{m} \mapsto m], \eta_D^n[\boldsymbol{m} \mapsto m], \overline{\rho}_D^n[R1 \mapsto m][\texttt{M-addr} \mapsto m]) \quad (36)$$

$\overline{\rho}_D^{n+1} \approx \rho_D^{n+1}$ because `M-addr` is in the special area of the memory map of the residual binary. This observation, along with Eqns. (35), (36), and (7), show that Eqn. (9) holds.

$$gSS^{n+1} = gSS^n[R1 \mapsto \boldsymbol{m}] \quad (37)$$

$$\eta_S^{n+1} = \eta_S^n[\boldsymbol{m} \mapsto m] \text{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto m]\ (By\ Eqn.\ (35)) \quad (38)$$

Note that the oracle supplies the correct $\boldsymbol{m}$ while updating *gSS*, and WIPER uses $\boldsymbol{m}$ to obtain the correct `M-addr` to use in the residual code. Eqns. (37), (38), and (8) show that Eqn. (10) holds.