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

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 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, 47], 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
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 [45] 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.\footnote{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.}

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.\footnote{Confirmed by personal communication with Neil Jones, Robert Glück, and Saumya Debray.}

• 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 \texttt{power} that computes \((a + b)^n\), where \(a\), \(b\), and \(n\) are the inputs to the program. Suppose that \texttt{power} is frequently called with the values \(b = 1\), and \(n = 4\). Given \texttt{power}, and the input values \(b = 1\) and \(n = 4\), a partial evaluator produces the residual program \texttt{power}_R shown in Fig. 2. As we can see in \texttt{power}_R, partial evaluation has compiled data \((n = 4)\) into control. (See Eqn. (1).) In effect, partial evaluation performs optimizations such as loop unrolling,
int power(int a, int b, int n) {
    int prod = 1;
    while (n--) {
        prod *= (a + 1);
    }
    return prod;
}

int power_r(int a) {
    int foo_r() {
        int foo() {
            int prod = 1;
            while (n--) {
                prod *= (a + b);
            }
            return prod;
        }
    }
    int power_r(int a) {
        int prod = 1;
        while (n--) {
            prod *= (a + 1);
        }
        return prod;
    }
}

Figure 1: Input program. Computes \((a + b)^n\).
Figure 2: Residual program. Computes \((a + 1)^n\).

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 \(b\) and statements 1, 3, and 5 as dynamic.

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

Approach PE

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

The second step is sometimes called residuation.

For the power program, \(\sigma\) is \(\{ n \rightarrow 4, b \rightarrow 1 \}\). Because the loop condition (Line 2 in Fig. 1) is static, the specializer unrolls the loop four times. The specializer residuates the expression \(a + b\) with respect to \(\sigma\).

Sometimes a static expression could be used in a dynamic context. For example, consider the program \(\text{foo}\) given in Fig. 3. Because \(c\)’s value is established dynamically in line 3, both of the formal parameters of \(\text{add}\) are classified dynamic by BTA. Although both actual parameters of the call to \(\text{add}\) in line 4 of Fig. 3 are static, they appear in a dynamic context—the static actuals will be bound to dynamic formals in \(\text{add}\). 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 \(\sigma\) that arises during an execution is partitioned into \(\sigma_S\) and \(\sigma_D\) 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 \((\sigma_S, \sigma_D)\) 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:

\[
l : (\sigma_S, \sigma_D) \rightarrow (l, \sigma_S) : \sigma_D
\]

Operationally, the partial evaluator creates a new residual statement for every unique \((l, \sigma_S)\) 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, 46] 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
INT ∈ Integer, Reg ∈ Register, I ∈ Instruction, O ∈ Operand,
     R ∈ RegisterOperand, C ∈ ImmediateOperand,
     M ∈ IndirectOperand, RC ∈ RegisterImmediateOperand
INT ::= {..., -1, 0, 1, ...}  Reg ::= eax | ebx | esp | ... 
     R ::= Direct(Reg)  C ::= Imm(INT)  RC ::= R | C 
     M ::= Indirect(R, R, INT)  O ::= RC | M 
I ::= push(O) | mov(R, O) | mov(M, RC) | lea(R, M) |
     add(R, O) | add(M, RC) | sub(R, O) | sub(M, RC) |
     cmp(M, C) | jz(C) | jmp(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 ⟨base, index, scale, offset⟩, 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.

\[ i \in \text{INT} = \mathbb{Z}_{32} \quad b \in \text{BOOL} = \{\text{True}, \text{False}\} \]

\[ r \in \text{Register} = \{\text{EAX}, \text{ESP}, \ldots\} \quad f \in \text{Flag} = \{\text{CF}, \text{SF}, \ldots\} \]

\[ \text{op} \in \text{ArithOp} = \{+, -, \ldots\} \quad \text{bop} \in \text{BoolOp} = \{\land, \lor, \ldots\} \]

\[ \text{rop} \in \text{RelOp} = \{\text{eq}, \neq, <, \ldots\} \quad \text{cop} \in \text{CondOp} = \{? :\} \]

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.

\[ \rho \in \text{Store} = \text{RegMap} \times \text{FlagMap} \times \text{MemMap} \]

\[ \text{RegMap} : \text{Reg} \rightarrow \text{INT} \quad \text{FlagMap} : \text{Flag} \rightarrow \text{BOOL} \quad \text{MemMap} : \text{INT} \rightarrow \text{INT} \]

The valuation function \( \mathcal{I} \) has the type \( \mathcal{I} : \text{Instruction} \rightarrow \text{Store} \rightarrow \text{Store} \). \( \mathcal{I} \) 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.

\[ T \in \text{Term}, \varphi \in \text{Formula}, \text{FE} \in \text{FuncExpr} \]

\[ c \in \text{Int32} = \{\ldots, -1, 0, 1, \ldots\} \quad b \in \text{Bool} = \{\text{True}, \text{False}\} \]

\[ I_{\text{Inst32}} \in \text{Int32Id} = \{\text{EAX}, \text{ESP}, \text{EBP}, \ldots\} \]

\[ I_{\text{Bool}} \in \text{BoolId} = \{\text{CF}, \text{SF}, \ldots\} \quad F \in \text{FuncId} = \{\text{Mem}\} \]

\[ \text{op} \in \text{ArithOp} = \{+, -, \ldots\} \quad \text{bop} \in \text{BoolOp} = \{\land, \lor, \ldots\} \]

\[ \text{rop} \in \text{RelOp} = \{=, \neq, <, \ldots\} \]

\[ T := \text{c \mid I_{\text{Inst32}} \mid T_1 \mid op \mid \text{ite(\varphi, T_1, T_2)} \mid F(T_1) \mid \varphi := \text{b \mid I_{\text{Bool}} \mid T_1 \mid rop \mid \text{⊤(\varphi, bop \varphi_2)} \mid F = \text{FE} \mid FE := F \mid FE_1 \mid T_1 \rightarrow T_2} \]

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

\[ \mathcal{I} \left[ \text{mov eax, [ebp]} \right] \left[ \varphi \right] \rho = \text{update}_\text{reg}(\rho, \left[\text{eax}\right], \text{access}_\text{mem}(\rho, \text{access}_\text{reg}(\rho, \left[\text{ebp}\right]))) \]

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 \( \text{ite}(\varphi, T_1, T_2) \) represents an if-then-else expression. A \( \text{FuncExpr} \) of the form \( \text{FE}[T_1 \rightarrow T_2] \) denotes a function-update expression.

The function \( \langle\rangle \) 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 \( \text{Int32Ids}, \text{BooIds}, \) and \( \text{FuncIds} \) can be qualified by primes (e.g., \( \text{Mem}' \)). The QFBV formula for an instruction sequence is a restricted 2-variables formula that specifies a store transformation (also known as “transition formula”). It has the form

\[ \bigwedge_m (T_m' = T_m) \land \bigwedge_n (J_n' = \varphi_n) \land \text{Mem}' = \text{FE}, \]

where \( T_m' \) 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 flagZFis set, the third instruction updates EIP.
\[\langle \text{mov eax, [ebp]} \rangle \equiv \text{EAX} = \text{Mem}(\text{EBP})\]
\[\langle \text{push 0} \rangle \equiv \text{ESP}' = \text{ESP} - 4 \land \text{Mem}' = \text{Mem}[\text{ESP} - 4 \rightarrow 0]\]
\[\langle \text{jz 1000} \rangle \equiv \text{ite}(ZF = 0, \text{EIP}' = 1000, \text{EIP}' = \text{EIP} + 4)\]
\[\langle \text{lea eax, [ebp+4]} \rangle \equiv \text{EAX}' = \text{EAX} + 4\]

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 \(I' = I, J' = J\), and \(\text{Mem}' = \text{Mem}\) 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 \(\text{sum}\) 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 \(\text{sum}\) 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.

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 decouples the 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 [45] (a tool that synthesizes an instruction sequence from a semantic specification) to synthe-
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 \( z' \) and \( \phi' \). 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 \( \rho_0 \) is

\[
\rho_0 \equiv \langle [\text{ESP} \rightarrow 1000], [\text{EBX} \rightarrow 1], [], [] \rangle.
\]

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 \( I \) for \( v \), and 1000 for the stack pointer. Let us specialize \( \sum' \) with respect to \( \rho_0 \).

Non-parameterized operands in Instruction ASTs.\(^5\) Recall from §2.1 that \( \text{PE}_S \) 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:

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 \( \rho_0 \) using this simple approach. The specializer evaluates instruction \( 1' \) in \( \sum' \) because it is static. The post-store is \( \rho_1 \).

\[
\rho_1 \equiv \langle [\text{ESP} \rightarrow 996], [\text{EBX} \rightarrow 1], [], [] \rangle.
\]

The QFBV formula for instruction \( 2' \) is

\[
\langle \langle z' \rangle \rangle = \langle \langle \text{Mem}' = \text{Mem}[\text{ESP} \rightarrow \text{EAX}] \rangle \rangle.
\]

Because \( z' \) is dynamic, the specializer specializes \( \langle \langle z' \rangle \rangle \) with respect to \( \rho_1 \) to obtain the specialized formula

\[
\text{Mem}' = \text{Mem}[996 \rightarrow \text{EAX}],
\]

and uses the synthesizer to synthesize the instruction “mov [996],eax.” After evaluating static instructions \( 3' \) and \( 4' \), the resulting store is

\[
\rho_4 \equiv \langle [\text{ESP} \rightarrow 992], [\text{EBX} \rightarrow 1], [], [992 \rightarrow I] \rangle.
\]

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

\[
\text{mov} \ [996], \text{eax} \quad \text{mov} \text{ecx}, \ [988] \\
\text{mov} \ [988], \text{ecx} \quad \text{mov} \text{ebx}, \ [964+\text{ecx}+4] \\
\text{mov} \text{eax}, \ [996] \quad \text{add} \text{eax}, \text{ebx}
\]

Lifting. The approach described above produces code that can access uninitialized locations. For example, one can see that the instruction “mov [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 PE**

- **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.

---

\(^4\)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).

\(^5\)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).
Let us illustrate $\sum'_1$ with respect to $\rho_0$ using $\text{PE}_1$. Because $1'$ is lifted, the specializer residades code for it. The QFBV formula for $1'$ is $\{1\} = \text{ESP} = \text{ESP} - 4$. The specializer specializes $\{1\}$ with respect to $\rho_0$ to obtain the specialized formula $\text{ESP'} = 996$, and uses the synthesizer to synthesize instruction $1$ in $\sum_1$ (shown in Fig. 11). The specializer also evaluates $1'$ to obtain $\rho_1$ (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 type values required by emitted dynamic instructions ($2_1, 6_1, \text{etc.}$) have been set up by residual lifted instructions ($1_1, 5_1, \text{etc.}$).

**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 $\sum_1$ partially evaluated with respect to $\rho_0$, the specialization-time stack layout is hard-wired 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 $\sum'_1$ dynamic.

To resolve this issue, $\text{WiPER}$ 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 $\text{ESP}$) are represented using symbolic constants.

Static non-address values are represented using integer and Boolean constants. $\text{WiPER}$ uses $\text{PE}_2$ for specialization.

Approach $\text{PE}_2$

- Static instructions: Symbolically evaluate. Although specialization-time addresses are symbolic, $\text{WiPER}$ 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 $\text{PE}_2$ on $\sum'_1$. The new initial store is $\rho_0^{\text{sym}} \equiv \langle \text{ESP} \mapsto c[\text{EBX} \mapsto 1], [], [] \rangle$. 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 $\text{ESP}$ is the symbolic constant $c$ to indicate that we are not restricting $\text{ESP}$ to a concrete value known at specialization-time.

Before starting specialization, $\text{WiPER}$ residuates instruction $0_2$ in $\sum_2$ (shown in Fig. 12) to save the initial value of the stack pointer $\text{ESP}$ in a dedicated location. (In this example, $\text{WiPER}$ uses the $\text{ESI}$ register because $\text{ESI}$ is not used in $\sum'_1$.) This location will be used whenever downstream residual instructions need the initial value of $\text{ESP}$. To residuate code for the lifted instruction $1'$, $\text{WiPER}$ specializes $\{1\}$ with respect to $\rho_{0}^{\text{sym}}$. The specialized formula is $\text{ESP'} = c - 4$. Because at run-time the value of $c$ will be available in $\text{ESI}$, $\text{WiPER}$ replaces $c$ with $\text{ESI}$ to produce $\text{ESP'} = \text{ESI} - 4$. $\text{WiPER}$ uses the synthesizer to synthesize instruction $1_2$ for the specialized formula. $\text{WiPER}$ also symbolically evaluates $1'$ to produce $\rho_1^{\text{sym}}$.

$$\rho_0^{\text{sym}} \equiv \langle \text{ESP} \mapsto c - 4 \rangle \langle \text{EBX} \mapsto 1 \rangle, [], [] \rangle.$$ $z'$ is dynamic, and $\text{WiPER}$ emits it as $2_2$. After symbolically evaluating static instructions $3'$ and $4'$, we obtain the store $\rho_4^{\text{sym}} \equiv \langle \text{ESP} \mapsto c - 8 \rangle \langle \text{EBX} \mapsto 1 \rangle, [], [c - 8 \mapsto 1] \rangle$.

The final residual program $\sum_2$ produced by $\text{WiPER}$ is shown in Fig. 12. One can see that $\text{WiPER}$ arranges for the initial stack-pointer value to be retrieved from $\text{ESI}$ ($1_2, 5_2, 9_2, 13_2$, etc.) instead of residuating the stack-pointer value at residual lifted instructions (cf. $1_1, 5_1, 9_1, 13_1$, 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 $13_1$ through $13_2$).

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

To formalize the effect of $\text{PE}_1$ and $\text{PE}_2$, 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 $\sigma$ consists of an environment $\eta$, and a store $\rho$, which maps locations to values. For example, $\rho_4$ in Eqn. (3) is actually part of a state $\sigma_4$

$$\sigma_4 \equiv \langle \eta_4, \rho_4 \rangle \equiv \langle \text{Stack} \mapsto 1000, \\
\langle \text{ESP} \mapsto 992 \rangle \langle \text{EBX} \mapsto 1 \rangle, [], [992 \mapsto 1] \rangle.$$ $\eta_4$ maps the start of the stack, denoted by the symbolic constant $\text{Stack}$, to the address 1000.

Suppose that $\eta$ and $\rho$ can be partitioned into $\eta_S$ and $\eta_D$, and $\rho_S$ and $\rho_D$, respectively. The effect of $\text{PE}_1$ 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: \( I \leftarrow \text{Synth}(\text{InstrToQFBV}(i)) \)
4: \( B' \leftarrow \text{Replace}(B, i, I) \)
5: end if
6: end for
7: return B’

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

\[ I : (\eta_S, \rho_S, \eta_D, \rho_D) \rightarrow (l, \eta_S, \rho_S) : (\eta_D, \rho_D) \]  

(4)

Operationally, the partial evaluator creates a new residual instruction for every unique \((l, \eta_S, \rho_S)\) 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 \(\text{PE}_1\) 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 \((\eta_S)\) is hard-wired into the residual code. If the run-time memory layout is different, the residual code produced by \(\text{PE}_1\) might crash.

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

\[ I : (\eta_S, \rho_S, \eta_D, \rho_D) \rightarrow (l, \rho_S^{sym}) : (\eta_S, \eta_D, \rho_D) \]  

(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 \(\text{PE}_2\) 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 \(\eta_S\). (See also §4.4.)

4. Algorithm

In this section, we describe the algorithms used by \(\text{WiPER}\). First, we present the algorithm for a pre-processing step used prior to partial evaluation. Second, we present \(\text{WiPER}\)’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, \(\text{WiPER}\) rewrites the input binary by decoupling each instruction \(i\) that updates the stack-pointer register \(\text{ESP}\) along with another location \(l\). \(\text{WiPER}\) uses the machine-code synthesizer \(\text{MCSYNTH}\) [45] for decoupling. \(\text{MCSYNTH}\) uses a divide-and-conquer strategy that splits \(i\)’s QFBV formula \(\varphi\) into two independent subformulas \(\varphi_1\) and \(\varphi_2\), such that the updates to \(\text{ESP}\) and \(l\) are split between \(\varphi_1\) and \(\varphi_2\). Then, \(\text{MCSYNTH}\) synthesizes code for \(\varphi_1\) and \(\varphi_2\), concatenates the results, and returns the resulting instruction sequence \(I\). \(\text{ESP}\) 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, \(\text{UpdatesSPAndLoc}\) returns true if an instruction updates the stack pointer along with another register or memory location; \(\text{InstrToQFBV}\) converts an instruction into a QFBV formula; \(\text{Synth}\) invokes \(\text{MCSYNTH}\) to synthesize an instruction sequence; \(\text{Replace}\) replaces an instruction in a binary with an instruction sequence.

Algorithm 2 BTA algorithm in \(\text{WiPER}\)

Input: SDG, \(I_D\)
Output: BTAMap

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

4.2 Intraprocedural Partial Evaluation

In this section, we present \(\text{WiPER}\)’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 \(\text{WiPER}\) is shown as Alg. 2. In the algorithm, \(I_D\) is the set of instructions that initialize dynamic inputs; \(\text{ForwardSlice}\) computes a forward slice with respect to a slicing criterion; \(\text{BTAMap}\) 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, \(\text{ForwardSlice}\) follows both data and control dependences during slicing. Lines 4–6 in Alg. 2 show the computation of lifted instructions during BTA. \(\text{WiPER}\) lifts all static calls to \text{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).

4.2.2.2 PE1 Specializer. specialize1 is a specialization function that implements PE1. specialize1 has two constituent functions, eval1 and reduce1. eval1 evaluates instructions, and reduce1 residuates code. The algorithm for specialize1 is given as Alg. 4.

eval1. eval1 can be implemented by writing an interpreter for IA-32 instructions using the concrete operational-semantics of IA-32. eval1 evaluates an instruction with respect to a pre-store, and returns the post-store.

reduce1. reduce1 can be defined as follows:

reduce1 : Instruction → Store → Instruction-sequence
reduce1(I, ρ) = Synth(Subst0(InstrToQFBV(I), ρ))

reduce1 converts an instruction into a QFBV formula, specializes the formula with respect to the pre-store (using Subst0), 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. Subst0 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, Subst0 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.)


Suppose that ρ is ρ ≡ ([ESP → 1000] EAX → 2], [], [1000 → I]). Subst0 produces the following formula:

Mem’ = Mem[1000 → 3] ∧ SF’ = False,

and reduce1(1, 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.)
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., \( \text{sum} \) in Fig. 11). In this section, we present the specialization algorithm that \( \text{WiPER} \) uses to residuate code that is independent of the specialization-time memory layout. In particular, we illustrate how the algorithm residuates code that is independent of the specialization-time annotations for the instructions are shown as comments later in upcoming paragraphs and sections.

\( \text{specialize}_2 \) uses symbolic addresses and a symbolic store to access and update memory at specialization time. To represent symbolic addresses, we enhance \( \text{Int32Id} \) in \( \text{LIA-32} \) by adding symbolic constants (\( m, n \), etc.). A different constant is used to represent the starting address of the stack and each heap block at run-time and at specialization-time.

\[ \text{specialize}_2 \text{ uses a symbolic store } \text{Store}^{\text{sym}} \text{ instead of } \text{Store}. \]

\[ \mu^{\text{sym}} \in \text{Store}^{\text{sym}} = \text{RegMap}^{\text{sym}} \times \text{FlagMap}^{\text{sym}} \times \text{MemMap}^{\text{sym}} \]

\[ \text{RegMap}^{\text{sym}} : \text{Int32Id} \to \text{Term} \quad \text{MemMap}^{\text{sym}} : \text{Term} \to \text{Term} \]

\[ \text{FlagMap}^{\text{sym}} : \text{BoolId} \to \text{Formula} \]

The residual code generated by \( \text{specialize}_2 \) will use specific run-time addresses to access and update memory using an extra level of indirection. To achieve this effect, \( \text{specialize}_2 \) uses a memory-layout map \( \mu \in \text{MemLayout} \). A memory-layout map \( \mu \) 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 \( m \).

\[ \mu \in \text{MemLayout} : \text{Int32Id} \to (\text{Register} \cup \text{INT}) \]

We will illustrate \( \text{specialize}_2 \), and its constituent functions \( \text{eval}_2 \) and \( \text{reduce}_2 \), using the code snippet shown in Fig. 14, which allocates and uses a heap block. (We describe how \( \text{specialize}_2 \) handles the stack and stack locations at the end of this sub-section.) The snippet has a static call to \( \text{malloc} \) 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 \( \text{WiPER} \) lifts all static calls to \( \text{malloc} \), instructions 1 and 2 are lifted.

The algorithms for \( \text{specialize}_2 \), \( \text{eval}_2 \), and \( \text{reduce}_2 \) are given as Algs. 5, 6, and 7, respectively. In Alg. 5, we overload \( \epsilon \) 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 \( \text{Alloc} \)), \( \text{specialize}_2 \) adds \( [m \mapsto \text{EDI}] \) to \( \mu \), where \( m \) is a fresh symbolic constant (obtained using \( \text{NewConst} \)) that is used to represent the specialization-time starting address of the heap block, and \( \text{EDI} \) is a location that is not used in the residual binary (obtained using \( \text{NewLoc} \)). This location can also be a global memory location that is otherwise unused in the residual binary. \( \text{specialize}_2 \) passes \( m \) to \( \text{eval}_2 \), which stores \( m \) in the key \( \text{NextBlock} \). \( \text{eval}_2 \) symbolically executes the instruction using \( I^{\text{sym}} \), which reinterprets the semantics of an instruction \( I \) to symbolically execute \( I \). If \( I \) allocates memory, \( I^{\text{sym}} \) uses the symbolic constant bound to \( \text{NextBlock} \) in \( \rho \) as

---

**Algorithm 5** \( \text{specialize}_2 \)

**Input:** \( I, \rho, \text{BTAMap}, \mu \)

**Output:** \( (I', \rho', \mu') \)

1: if \( \text{BTAMap}[I] = \text{Dynamic} \) then
   2: \( \text{return} \) \( (I, \rho, \mu) \)
3: else if \( \text{BTAMap}[I] = \text{Lifted} \) then
   4: \( m \leftarrow \epsilon \)
   5: \( l \leftarrow \epsilon \)
   6: if \( \text{Alloc}(I) \) then
      7: \( m \leftarrow \text{NewConst}() \)
      8: \( l \leftarrow \text{NewLoc()} \)
      9: \( \mu' \leftarrow \mu|m \mapsto l|' \)
   10: end if
   11: \( \text{return} \) \( (\text{reduce}_2(I, \rho, \mu, l), \text{eval}_2(I, \rho, m), \mu') \)
12: else
13: \( \text{return} \) \( (\epsilon, \text{eval}_2(I, \rho, \epsilon), \mu) \)
14: end if

**Algorithm 6** \( \text{eval}_2 \)

**Input:** \( I, \rho, m \)

**Output:** \( \rho' \)

1: \( \rho' \leftarrow I^{\text{sym}}[I](\rho|_{\text{NextBlock} \mapsto m}) \)
2: \( \text{return} \) \( \rho' \)

**Algorithm 7** \( \text{reduce}_2 \)

**Input:** \( I, \rho, m, l \)

**Output:** \( I' \)

1: \( I' \leftarrow \text{Synth}(\text{Subst}(\text{Subst}_0(\text{InstrToQFBV}(I, \rho), \rho))) \)
2: \( l' \leftarrow \epsilon \)
3: if \( \text{Alloc}(I) \) then
   4: \( l'' \leftarrow \text{EmitMove}(l, \text{EAX}) \)
5: end if
6: \( \text{return} \) \( I'; l'' \)
Every static instruction that accesses (updates) memory are specific division of instructions in our running example from §. Additionally, in the initial symbolic store, the next address in the free list. Consequently, in $\rho'$ returned by $\text{eval}_2$ for the call to malloc, $\text{EAX}$ holds the symbolic constant $m$—i.e., $m$ is the starting address of the heap block allocated at instruction 2.

$\text{specialize}_2$ then passes $l$, $\rho$, $\mu$, and $\text{EDI}$ to $\text{reduce}_2$. Because instruction 2 allocates memory, $\text{reduce}_2$ residuates two instructions. The first instruction ($2_R$) is the call to malloc; the second instruction ($2_R$) saves the $\text{EAX}$ register in $\text{EDI}$ (emitted using $\text{EmitMove}$). After the residual code executes instruction $2_R$, the run-time starting address of the heap block will be available in $\text{EDI}$.

$\text{specialize}_2$ symbolically executes the static instruction 3, producing the post-store ($\text{EAX} \mapsto m$, $\mu$, $[m \mapsto 1]$). While residuating code for lifted instruction 4, $\text{reduce}_2$ uses $\text{Subst}_R$ 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

$$\text{Mem}' = \text{Mem} \cdot m + 4 \mapsto 2.$$

(6)

$\text{reduce}_2$ then replaces each symbolic constant $m$ in the resulting formula with $[\mu(m)]_L$ using $\text{Subst}_R$. For our example, for Eqn. (6), $\text{Subst}_R$ produces $\text{Mem}' = \text{Mem}[	ext{EDI} + 4 \mapsto 2]$. Finally, $\text{reduce}_2$ 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 $n$ 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 $[n \mapsto l]$ to $\mu$ before specialization begins, and emits “$\textit{mov l, esp}$” as the first instruction in the residual program. Additionally, in the initial symbolic store, $\text{ESP}$ is bound to $n$. In our running example from §3, WiPER adds $[n \mapsto \text{ESI}]$ to $\mu$. Consequently, $\text{reduce}_2$ uses $\text{ESI}$ instead of $n$ in the QFBV formulas of all downstream lifted instructions.

In summary, to residuate code that is independent of the specialization-time memory layout, $\text{specialize}_2$ 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.

PE2-Safety. For PE2 to produce correct residual code for a binary B, B should be PE2-safe. A binary B, along with a specific division of instructions in B as static and dynamic, are PE2-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., $\text{cmp esp,1000}$).

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 $\rho^\text{sym}$ that arises during specialization of B can be simplified to have the form $c$, or $m + c$, where $c$ is an integer constant, and $m$ 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

```assembly
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 $\rho^\text{sym}_2$

$$\rho^\text{sym}_2 = \langle [\text{ESP} \mapsto n], \mu, [n + 1000 \mapsto 10][n + 2 \mapsto 20] \rangle$$

In $\rho^\text{sym}_2$, the term $n + 2$ cannot be rewritten in the form $m + c$. Note that the value in the EAX register after (concretely) executing instruction 3 depends on the initial value of EAX.

If we execute the snippet with the initial concrete store $\rho_0 \equiv \langle [\text{ESP} \mapsto 1000], \mu, \rangle$, the value in EAX after executing instruction 3 will be 20. If we execute the code snippet with a different value for EAX in $\rho_0$, 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 PE2-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.)
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 (ρ, specializedCFGs) ← 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 (CFG, ρ) → ρ’, 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 PE2-safe binary, and ρsym is a partial static symbolic-store. Then WiPER(B, ρsym) 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, ρsym) 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 $\eta_S$, $\rho_S$, $\eta_D$, and $\rho_D$. Thus the type for the evaluation function $\llbracket \cdot \rrbracket$ (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

$$
\llbracket \cdot \rrbracket : \text{Instruction} \times \text{Env}_S \times \text{Store}_S \times \text{Env}_D \times \text{Store}_D \rightarrow \text{Env}_S \times \text{Store}_S \times \text{Env}_D \times \text{Store}_D.
$$

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

$$
\llbracket \cdot \rrbracket : \text{Instruction} \times \text{Env}_S \times \text{Env}_D \times \text{Store}_D \rightarrow \text{Env}_S \times \text{Env}_D \times \text{Store}_D
$$

It will always be clear from context which meaning of $\llbracket \cdot \rrbracket$ 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 $\rho$ and $\bar{\rho}$ modulo the special area by $\rho \equiv \bar{\rho}$.

For an input binary $B$ that is PE$_2$-safe, and an IA-32 state $\sigma$ that can be partitioned into $\eta_S$, $\rho_S$, $\eta_D$, and $\rho_D$ 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 PE$_2$-safe. Also suppose that $\eta_S$ is an environment such that there are no overlaps among (a) any of the chunks in $\text{Dom}(\eta_S)$, (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 $\rho'_S$ is a symbolic store such that $\llbracket \rho'_S \rrbracket(\eta_S) = \rho_S$, then $\llbracket \text{WiPER}(B, \rho'_S) \rrbracket(\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \rho'_D)$, where $\rho'_D \equiv \rho'_D$.

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

Thm. 2 does not include $\rho_S$ while comparing states because the residual code produced by WiPER($B$, $\rho'_S$) does not contain static instructions, and consequently never accesses or updates the static store $\rho_S$.

### 4.5 Threats to Validity

There are two threats to the validity of our algorithms.

1. If the input binary is not 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 underapproximative, 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 MC SYNTH [45] to synthesize an IA-32 instruction sequence from a QFBV formula. MC SYNTH sometimes requires a set of scratch registers to hold results of intermediate calculations in the residual code. WiPER supplies dead registers to MC SYNTH 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 MC SYNTH.

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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Domain</th>
<th>LOC</th>
<th>Description</th>
<th>Static Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dotproduct</td>
<td>Linear algebra operations</td>
<td>29</td>
<td>Computes the dot product of two vectors with $n$ dimensions</td>
<td>$n = 100$</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Language interpreters</td>
<td>71</td>
<td>Interpreter for the minimalist language “Brainf*ck”</td>
<td>Input program</td>
</tr>
<tr>
<td>Filter</td>
<td>Image processing</td>
<td>107</td>
<td>Applies a convolution filter of size $n \times n$ on an image of size $n \times n$</td>
<td>$n = 3$, $n = 3$, and elements of the filter</td>
</tr>
<tr>
<td>uuencode</td>
<td>Text utilities</td>
<td>129</td>
<td>Base-64 encodes a string of size $n$</td>
<td>$n = 256$, and the string</td>
</tr>
<tr>
<td>SHA1</td>
<td>Cryptographic operations</td>
<td>140</td>
<td>Computes the shal digest of a message of size $n$ bits</td>
<td>$n = 1024$, and contents of the first 512 bits</td>
</tr>
<tr>
<td>uudecode</td>
<td>Text utilities</td>
<td>167</td>
<td>Decodes a base-64-encoded string of size $n$</td>
<td>$n = 206$, and the encoded string</td>
</tr>
</tbody>
</table>

```
foo:
main:
    push ebp
    mov ebp,esp
    and esp,0xFFFFF000
    sub esp,170
    
Figure 16: Code snippet to illustrate variable alignment.
```

PE$_2$-safety properties to allow terms of the form $m + p_0 + p_1 + \ldots + c$, where $m$ is a symbolic constant representing the specialization-time starting address of the stack or a heap block $M$, $c$ is an integer constant, and $p_0$, $p_1$, 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 $l_S$ 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 con-
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:

```assembly
loc_8049A3A: mov eax,dword [esp+44] ;L
loc_8049A3E: mov dword [esp+12],eax ;D
loc_8049A5E: call fx_create ;D
loc_80499E2:
loc_8049A84: 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 8049A3E should be treated as static instructions. WiPER lifts instruction 8049A3E 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

\[ \rho \equiv \langle [EAX \mapsto 0][ESP \mapsto n],[],[] \rangle, \]

and residues the instruction mov eax,0 for the lifted instruction 8049A3E, and emits the remaining dynamic instructions. Because eax is 0 at instruction 80499D4, the fallthrough 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:

```assembly
loc_805B539: mov eax,[esp+40] ;S
loc_805B53E: cmp eax,1 ;S
loc_805B541: jnz loc_805B5DB ;S
loc_805B558: call compress ;D
loc_805B5DB:
loc_805B5E4: 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

\[ \rho \equiv \langle [ESP \mapsto n],[],[n + 40 \mapsto 1]\rangle. \]

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 PE1, 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, Junghie 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

A. Appendix

Theorem 2. Suppose that $B$ is PE$_{\mathcal{E}}$-safe. Also suppose that $\eta_S$ is an environment such that there are no overlaps among (a) any of the chunks in Dom($\eta_S$), (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 $\rho'_S$ is a symbolic store such that $[\rho'_S](\eta_S) = \rho_S$, then $[\text{WiPEr}(B, \rho'_S)](\eta_S, \eta_D, \rho_D) = (\eta'_S, \eta'_D, \rho'_D)$, where $\rho'_D \approx \rho'_D$. □

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 stackalloc instruction, the single operand is the destination operand. For the heapalloc instruction, the single operand is both a source and destination operand.
we assume the following:

\[ \text{INT} ::= \{ \ldots, -1, 0, 1, \ldots \} \quad \text{Reg} ::= r_1 \mid r_2 \mid rw_1 \]
\[ \text{R} ::= \text{Direct}(\text{Reg}) \quad \text{C} ::= \text{Imm}((\text{INT})) \quad \text{M} ::= \text{Indirect}(\text{R}) \]
\[ I ::= \text{mov}(\text{R}, \text{C}) \mid \text{mov}(\text{R}_1, \text{R}_2) \mid \text{mov}(\text{R}, \text{M}) \mid \text{mov}(\text{M}, \text{R}) \mid \text{add}(\text{R}_1, \text{R}_2) \mid \text{stackalloc}(\text{R}) \mid \text{heapalloc}(\text{R}) \]

Figure 18: Abstract syntax of TinyIA32.

The register \( rw_1 \) 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 \( \text{WIPER} \) 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 \( \text{mov} \) instruction loads a 32-bit constant into a register. The second \( \text{mov} \) instruction performs a register-to-register move. The final two \( \text{mov} \) instructions move a 32-bit value from a memory location to a register and from a register to a memory location, respectively. The \( \text{add} \) instruction adds the 32-bit values in the two register operands, leaving the result in the destination operand register. The \( \text{stackalloc}(\text{R}) \) instruction allocates a stack of some fixed, but arbitrarily large, size, whose starting address is available as an output in R. The \( \text{heapalloc}(\text{R}) \) 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 \( \text{WIPER} \) for \( T \) is \( T' \equiv J_1, J_2, \ldots, J_n \), where \( J_i \) is the residual instruction sequence produced by the \( \text{Specialize} \) phase of \( \text{WIPER} \) 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 \( \eta \) 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:

\[
\begin{align*}
[I_1, I_2, \ldots, I_n](\eta_S, \rho_S, \eta_D, \rho_D) &= (\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D) \\
\wedge [J_1, J_2, \ldots, J_n](\eta_S, \eta_D, \rho_D) &= (\eta^n_S, \eta^n_D, \rho^n_D) \\
\wedge \rho^n_D &= \rho^n_D
\end{align*}
\]

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

We strengthen the induction hypothesis with the following conjunct:

\[ [gSS^k] \eta^n_S = \rho^n_S, \quad (8) \]

where the left-hand side denotes the simplification of each symbolic expression in \( \text{gSS} \) with respect to the values obtained from \( \eta^n_S \). Because \( \text{gSS}^0 = \rho^n_S \), by the hypothesis \([\rho^n_S](\eta^n_S) = \rho_S \) in Thm. 2, we have \([\text{gSS}^0](\eta^n_S) = \rho_S \), 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

\[
[I_1, I_2, \ldots, I_{n+1}](\eta_S, \rho_S, \eta_D, \rho_D) = (\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D) \wedge [J_1, J_2, \ldots, J_{n+1}](\eta_S, \eta_D, \rho_D) = (\eta^n_S, \eta^n_D, \rho^n_D) \\
\wedge \rho^n_D = \rho^n_D \\
\wedge [gSS^{n+1}](\eta^n_S) = \rho^n_S + 1.
\]

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

\[ l : (\eta_S, \rho_S, \eta_D, \rho_D) \mapsto (l, gSS) : (\eta_S, \eta_D, \rho_D) \]

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 \( \text{stackalloc}(\text{R}) \) or \( \text{heapalloc}(\text{R}) \). 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

\[
[I_{n+1}](\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D) = (\eta^n_S, \rho^n_S + 1, \eta^n_D, \rho^n_D),
\]

where some update is made to \( \rho^n_S \) to create \( \rho^n_{S+1} \). Specialize\((I_{n+1}, gSS^n) \equiv J_{n+1} = \epsilon\),

where \( \epsilon \) denotes the empty sequence of instructions. Because \( J_{n+1} \) is empty, we have

\[
[I_{n+1}](\eta^n_S, \eta^n_D, \rho^n_D) = (\eta^n_S, \eta^n_D, \rho^n_D).
\]

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

The congruence property of BTA ensures that \( \text{WIPER} \) can completely evaluate \( I_{n+1} \) at specialization time, and \( gSS^n \) is updated by \( \text{WIPER} \) to produce \( gSS^{n+1} \) in the same way as \( \rho^n_S \) is updated by \( I_{n+1} \) to produce \( \rho^n_{S+1} \). Moreover, because \( \eta^n_S \) is not updated by \( I_{n+1} \), by Eqn. (8) we have \([gSS^{n+1}](\eta^n_S) = \rho^n_{S+1} \), and Eqn. (10) holds. Consequently, the induction hypothesis holds on a trace with \( n + 1 \) instructions, where the \((n + 1)^\text{th}\) 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 \( \text{PiPER}(B, \rho_S^\text{sym}) \) 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 \( \eta \) to \( \eta[I \mapsto m] \), where \( m \) is a symbolic constant that denotes the starting address of \( M \). (Lifted allocation instructions update \( \eta_S \) and \( \eta_D \); dynamic allocation instructions update \( \eta_D \).)

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

\[
[I_{n+1}](\eta^n_D, \rho^n_S, \eta^n_D, \rho^n_D) = (\eta^n_D, \rho^n_S, \eta^n_D, \rho^n_D + 1), \tag{13}
\]

where some update is made to \( \rho^n_D \) to create \( \rho^n_{D+1} \).

Specialize \( I_{n+1} \), \( gSS^n \) \( \equiv I_{n+1} \). Because \( J_{n+1} \) is the same as \( I_{n+1} \), we have

\[
[J_{n+1}](\eta^n_D, \rho^n_D, \eta^n_D, \rho^n_D) = (\eta^n_D, \rho^n_D, \eta^n_D, \rho^n_{D+1}), \tag{14}
\]

where \( \rho^n_{D+1} \approx \rho^n_{D+1} \). Eqns. (13) and (14), together with Eqn. (7), show that Eqn. (9) holds.

The static components of the state, \( \eta^n_S \), \( gSS^n \), and \( \rho^n_S \), are left unchanged, so Eqn. (10) holds.

The cases for \text{stackalloc} and \text{heapalloc} are similar, except that those instructions update \( \eta^n_D \) in addition to \( \rho^n_D \). Moreover, because of our assumption about the storage allocator, the updates to \( \eta^n_D \) by \( I_{n+1} \) and \( J_{n+1} \) are identical, and the updates to \( \rho^n_D \) 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 \((n+1)\text{th} \) instruction is dynamic.

Lifted instructions. The semantics we use for evaluating static and dynamic instructions is fairly standard—static instructions access and update \( \rho_S \), and dynamic instructions access and update \( \rho_D \) (in addition to updating \( \eta_D \)). 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 \( \eta_D \) and \( \rho_D \) for use by downstream dynamic instructions.

Lifted instructions access \( \rho_S \), but produce values that might be consumed by dynamic successors, so they must update \( \rho_D \). In addition, lifted instructions can also have static successors, so they must also update \( \rho_S \). Thus, in the semantics used for the original program, a lifted instruction performs the same update on both \( \rho_S \) and \( \rho_D \). In the semantics used for the residual program, it updates only \( \rho_D \) because \( \rho_S \) 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 \( \eta_S \) and \( \eta_D \).

The non-overlap hypothesis of Thm. 2, together with property P2 of \( \text{PE}_2 \)-safety, imply that each specialization-time address can be represented canonically by a term of the form \( m + c \), where \( m \) is the symbolic constant that denotes the starting address of the allocated stack or 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 \( \text{PiPER} \) partially evaluates a lifted instruction \( I_i \) in \( T \) that allocates memory, the symbolic constant \( m \) 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 \textit{same} symbolic constant \( m \) gets added to the ghost variables \( \eta_S \) and \( \eta_D \) 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).) \( \text{PiPER} \) binds \( m \) to a dedicated global address \( \text{M-addr} \) and uses \( \text{M-addr} \) in the residual code to save the concrete starting address \( m \) of the allocated memory. (\( \text{M-addr} \) 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 \( \text{PiPER} \) uses a memory-layout map \( \mu \) 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 \textit{let} \( (m, mm) = \rho \text{ in } (rm[R1 \mapsto v], mm) \), 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.

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

\[
[I_{n+1}](\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D) = (\eta^n_S, \rho^n_S, [R1 \mapsto v], \rho^n_D, [R1 \mapsto v]) \tag{15}
\]

Specialize \( I_{n+1}, gSS^n \equiv J_{n+1} = \text{mov} \; r1, v \)

\[
[J_{n+1}](\eta^n_S, \eta^n_D, \rho^n_D, [R1 \mapsto v]) = (\eta^n_S, \eta^n_D, \rho^n_D, [R1 \mapsto v]) \tag{16}
\]

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

\[
gSS^{n+1} = gSS^n \; [R1 \mapsto v] \tag{17}
\]

\[
\eta^n_S + 1 = \eta^n_S \; \text{and} \; \rho_S^{n+1} = \rho_S^n \; [R1 \mapsto v] \; \text{(From (15))} \tag{18}
\]

Eqns. (17), (18), and (8) show that Eqn. (10) holds.
2. \( I_{n+1} = \text{mov (Direct(R1), Direct(R2))} \): There are two cases that can arise.

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

\[
\begin{align*}
[I_{n+1}][\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n] &= \\
&= (\eta_S^n, \rho_S^n[RI \rightarrow u], \eta_D^n, \rho_D^n[RI \rightarrow u]) \quad (19)
\end{align*}
\]

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

\[
\begin{align*}
[J_{n+1}][\eta_S^n, \eta_D^n, \rho_D^n] &= (\eta_S^n, \eta_D^n, \rho_D^n[R1 \rightarrow v]) \quad (20)
\end{align*}
\]

\( u = \rho_S^g(R2) = ([gSS^n]\eta_S^n)(R2) \) (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.

\[
\begin{align*}
gSS^{n+1} &= gSS^n[R1 \rightarrow v] \\
\eta_S^{n+1} &= \eta_S^n \quad \text{and } \rho_S^{n+1} = \rho_S^n[R1 \rightarrow u] \quad \text{(From 19)}
\end{align*}
\]

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

(b) Suppose that \( \rho_S^g(R2) = u \) and \( gSS^n(R2) = m + c \).

\[
\begin{align*}
\text{specialize}(I_{n+1}, gSS^n) = J_{n+1} &= \text{mov } r1, m; \\
&\text{mov } r1, [rwl]; \text{mov } rwl, c; \text{add } r1, rwl;
\end{align*}
\]

\[
\begin{align*}
[J_{n+1}][\eta_S^n, \eta_D^n, \rho_D^n] &= (\eta_S^n, \eta_D^n, \rho_D^n[R1 \rightarrow m+c]) \quad (24)
\end{align*}
\]

\( u = \rho_S^g(R2) = ([gSS^n]\eta_S^n)(R2) \) (By Eqn. (8))

\[
= ([gSS^n(R2)]\eta_S^n) = [m+c][\eta_S^n] = m+c \quad (25)
\]

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

\[
\begin{align*}
gSS^{n+1} &= gSS^n[R1 \rightarrow m+c] \\
\eta_S^{n+1} &= \eta_S^n \quad \text{and } \rho_S^{n+1} = \rho_S^n[R1 \rightarrow u] \quad \text{(From 19)}
\end{align*}
\]

Eqns. (25)–(27), and (8) show that Eqn. (10) holds. Note that \( J_{n+1} \) has the mov \( r1, 0 \) instruction at the end of the residual code to restore the default value 0 in work register \( rwl \).

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

(a) Suppose that \( \rho_S^g(R2) = a \) and \( gSS^n(R2) = m + c \) and \( gSS^n(m+c) \).

\[
\begin{align*}
[I_{n+1}][\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n] &= \\
&= (\eta_S^n, \rho_S^n[RI \rightarrow u], \eta_D^n, \rho_D^n[RI \rightarrow u]) \quad (28)
\end{align*}
\]

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

\[
\begin{align*}
[J_{n+1}][\eta_S^n, \eta_D^n, \rho_D^n] &= (\eta_S^n, \eta_D^n, \rho_D^n[R1 \rightarrow v]) \quad (29)
\end{align*}
\]

\( u = \rho_S^g(a) = \rho_S^g(\rho_S^g(R2)) \)

\[
= \rho_S^g([gSS^n]\eta_S^n)(R2) \) (By Eqn. (8))

\[
= \rho_S^g([gSS^n(R2)]\eta_S^n) = \rho_S^g([m+c][\eta_S^n])
\]

\[
= ([gSS^n(m+c)]\eta_S^n) = [v][\eta_S^n] = v \quad (30)
\]

Eqns. (28)–(30), and (7) show that Eqn. (9) holds.

\[
\begin{align*}
gSS^{n+1} &= gSS^n[m+c \rightarrow v] \\
\eta_S^{n+1} &= \eta_S^n \quad \text{and } \rho_S^{n+1} = \rho_S^g[a \rightarrow u] \quad \text{(From 28)}
\end{align*}
\]

Eqns. (30)–(32), and (8) show that Eqn. (10) holds.

(b) Suppose that \( \rho_S^g(R2) = a \) and \( gSS^n(R2) = m_1 + c_1 \) and \( gSS^n(m_1 + c_1) = m + c \).

\[
\begin{align*}
\text{specialize}(I_{n+1}, gSS^n) = J_{n+1} &= \text{mov } rwl, m; \\
&\text{mov } r1, [rwl]; \text{mov } rwl, c; \text{add } r1, rwl;
\end{align*}
\]

\[
\begin{align*}
[J_{n+1}][\eta_S^n, \eta_D^n, \rho_D^n] &= (\eta_S^n, \eta_D^n, \rho_D^n[R1 \rightarrow m+c]) \quad (33)
\end{align*}
\]

\( u = \rho_S^g(a) = \rho_S^g(\rho_S^g(R2)) \)

\[
= \rho_S^g([gSS^n]\eta_S^n)(R2) \) (By Eqn. (8))

\[
= \rho_S^g([gSS^n(R2)]\eta_S^n) = \rho_S^g([m_1+c_1][\eta_S^n])
\]

\[
= ([gSS^n(m_1+c_1)]\eta_S^n) = [m+c][\eta_S^n] = m+c \quad (34)
\]

Eqns. (28), (33), (34), and (7) show that Eqn. (9) holds.

\[
\begin{align*}
gSS^{n+1} &= gSS^n[R1 \rightarrow m+c] \\
\eta_S^{n+1} &= \eta_S^n \quad \text{and } \rho_S^{n+1} = \rho_S^n[R1 \rightarrow u] \quad \text{(From 28)}
\end{align*}
\]

Eqns. (34)–(36), and (8) show that Eqn. (10) holds.

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

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

\[
\begin{align*}
[I_{n+1}][\eta_S^n, \rho_S^n, \eta_D^n, \rho_D^n] &= \\
&= (\eta_S^n, \rho_S^n[RI \rightarrow u], \eta_D^n, \rho_D^n[RI \rightarrow u]) \quad (37)
\end{align*}
\]

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

\[
\begin{align*}
[J_{n+1}][\eta_S^n, \eta_D^n, \rho_D^n] &= (\eta_S^n, \eta_D^n, \rho_D^n[R1 \rightarrow v]) \quad (38)
\end{align*}
\]

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. (37), (38), and (7), show that Eqn. (9) holds.

\[
\begin{align*}
gSS^{n+1} &= gSS^n[m+c \rightarrow v] \\
\eta_S^{n+1} &= \eta_S^n \quad \text{and } \rho_S^{n+1} = \rho_S^g[a \rightarrow u] \quad \text{(From 37)}
\end{align*}
\]

The equalities \( u = v \) and \( a = m + c \), along with Eqns. (39), (40), and Eqn. (8), show that Eqn. (10) holds.
(b) Suppose that $\rho_S^n(R1) = a$ and $\rho_S^n(R2) = u$, and $gSS^n(R1) = m_1 + c_1$ and $gSS^n(R2) = m_2 + c_2$.

5. I

Specialize($I_{n+1}$, $gSS^n$) $\equiv J_{n+1} = \text{mov } rwl, M-addr;$

\[ \text{mov } r1, [rwl]; \text{mov } rwl, c; \text{add } r1, rwl; \]
\[ \text{mov } rwl, M-addr; \text{mov } r2, [rwl]; \]
\[ \text{mov } rwl, c_2; \text{add } r2, rwl; \text{mov } [r1], r2; \]
\[ \text{mov } rwl, 0 \]
\[ [J_{n+1}][\eta^n_S, \eta^n_D, \eta_D^n] = (\eta^n_S, \eta^n_D, \eta_D^n[m_1+c_1 \mapsto m_2+c_2]) \]
(41)

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. (37), (41), and (7), show that Eqn. (9) holds.

\[ gSS^n+1 = gSS^n[m_1 + c_1 \mapsto m_2 + c_2] \]
(42)
\[ \eta^n_S+1 = \eta^n_S \text{ and } \rho^n_S+1 = \rho^n_S[a \mapsto u] \text{ (By Eqn. (37))} \]
(43)

The equalities $a = m_1 + c_1$ and $u = m_2 + c_2$, along with Eqns. (42), (43), and Eqn. (8), show that Eqn. (10) holds.

5. $I_{n+1} = \text{add (Direct(R1), Direct(R2))}$ There are two cases that can arise.

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

\[ [I_{n+1}][\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D] = \]
\[ (\eta^n_S, \rho^n_S[R1 \mapsto u], \eta^n_D, \rho^n_D[R1 \mapsto u]) \]
(44)

specialize($I_{n+1}$, $gSS^n$) $\equiv J_{n+1} = \text{mov } r1, v$
\[ [I_{n+1}][\eta^n_S, \eta^n_D, \eta_D^n] = (\eta^n_S, \eta^n_D, \eta_D^n[R1 \mapsto v]) \]
(45)
\[ u = \rho^n_S(R1) + \rho^n_S(R2) = \]
\[ = (gSS^n[R1] + (gSS^n[R2]) \text{ (By Eqn. (8))} \]
\[ = (gSS^n[R1]) \eta^n_S + (gSS^n[R2]) \eta^n_S \]
\[ = (gSS^n[R1] + gSS^n[R2]) \eta^n_S = \eta^n_S + v \]
(46)

Eqns. (44)–(46), and (7) show that Eqn. (9) holds.

\[ gSS^n+1 = gSS^n[R1 \mapsto v] \]
(47)
\[ \eta^n_S+1 = \eta^n_S \text{ and } \rho^n_S+1 = \rho^n_S[R1 \mapsto u] \text{ (From (44))} \]
(48)

Eqns. (46)–(48), and (8) show that Eqn. (10) holds.

(b) Suppose that $\rho_S^n(R1) + \rho_S^n(R2) = u$, and $gSS^n(R1) + gSS^n(R2) = m + c$.

specialize($I_{n+1}$, $gSS^n$) $\equiv J_{n+1} = \text{mov } rwl, M-addr;$

\[ \text{mov } r1, [rwl]; \text{mov } rwl, c; \text{add } r1, rwl; \]
\[ \text{mov } rwl, 0 \]
\[ [J_{n+1}][\eta^n_S, \eta^n_D, \eta_D^n] = (\eta^n_S, \eta^n_D, \eta_D^n[R1 \mapsto m + c]) \]
(49)
\[ u = \rho^n_S(R1) + \rho^n_S(R2) = \]
\[ = (gSS^n[R1] + (gSS^n[R2]) \text{ (By Eqn. (8))} \]
\[ = (gSS^n[R1] + gSS^n[R2]) \eta^n_S \]
\[ = \eta^n_S \]
(50)

Eqns. (44), (49), (50), and (7) show that Eqn. (9) holds.

\[ gSS^n+1 = gSS^n[R1 \mapsto m + c] \]
(51)
\[ \eta^n_S+1 = \eta^n_S \text{ and } \rho^n_S+1 = \rho^n_S[R1 \mapsto u] \text{ (From (44))} \]
(52)

Eqns. (50)–(52), and (8) show that Eqn. (10) holds.

6. $I_{n+1} = \text{stackalloc(Direct(R1))}$

\[ [I_{n+1}][\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D] = (\eta^n_S[m \mapsto m], \rho^n_S[R1 \mapsto m]) \]
(53)

Specialize($I_{n+1}$, $gSS^n$) $\equiv J_{n+1} = \text{stackalloc r1;}

\[ \text{mov } rwl, M-addr; \text{mov } [rwl], r1; \text{mov } rwl, 0 \]
\[ [J_{n+1}][\eta^n_S, \eta^n_D, \eta_D^n] = (\eta^n_S[m \mapsto m], \eta^n_D[m \mapsto m], \eta_D^n[R1 \mapsto m]) \]
(54)
\[ \eta^n_D+1 \approx \rho^n_D \] because $M$-addr is in the special area of the memory map of the residual binary. This observation, along with Eqns. (53), (54), and (7), show that Eqn. (9) holds.

\[ gSS^n+1 = gSS^n[R1 \mapsto m] \]
(55)
\[ \eta^n_S+1 = \eta^n_S[m \mapsto m] \text{ and } \rho^n_S+1 = \rho^n_S[R1 \mapsto m] \text{ (By Eqn. (53))} \]
(56)

Note that the oracle supplies the correct $m$ while updating $gSS$, and $\text{WiPER}$ uses $m$ to obtain the correct $M$-addr to use in the residual code. Eqns. (55), (56), and (8) show that Eqn. (10) holds.

7. $I_{n+1} = \text{heapalloc(Direct(R1))}$

\[ [I_{n+1}][\eta^n_S, \rho^n_S, \eta^n_D, \rho^n_D] = (\eta^n_S[m \mapsto m], \rho^n_S[R1 \mapsto m]) \]
(57)

Specialize($I_{n+1}$, $gSS^n$) $\equiv J_{n+1} = \text{heapalloc r1;}

\[ \text{mov } rwl, M-addr; \text{mov } [rwl], r1; \text{mov } rwl, 0 \]
\[ [J_{n+1}][\eta^n_S, \eta^n_D, \eta_D^n] = (\eta^n_S[m \mapsto m], \eta_D^n[m \mapsto m], \eta_D^n[R1 \mapsto m]) \]
(58)
\( p_D^{n+1} \approx \rho_D^{n+1} \) because \( M\text{-addr} \) is in the special area of the memory map of the residual binary. This observation, along with Eqns. (57), (58), and (7), show that Eqn. (9) holds.

\[
gSS^{n+1} = gSS^n[R1 \mapsto m] \tag{59}
\]
\[
\eta_S^{n+1} = \eta_S^n[m \mapsto m] \text{ and } \rho_S^{n+1} = \rho_S^n[R1 \mapsto m] \tag{60}
\]
(By Eqn. (57))

Note that the oracle supplies the correct \( m \) while updating \( gSS \), and WiPER uses \( m \) to obtain the correct \( M\text{-addr} \) to use in the residual code. Eqns. (59), (60), and (8) show that Eqn. (10) holds. \( \square \)