Modeling Dynamic (De)Allocations of Local Memory for Translation Validation

ABHISHEK ROSE, Indian Institute of Technology Delhi, India
SORAV BANSAL, Indian Institute of Technology Delhi, India

End-to-End Translation Validation is the problem of verifying the executable code generated by a compiler against the corresponding input source code for a single compilation. This becomes particularly hard in the presence of dynamically-allocated local memory where addresses of local memory may be observed by the program. In the context of validating the translation of a C procedure to executable code, a validator needs to tackle constant-length local arrays, address-taken local variables, address-taken formal parameters, variable-length local arrays, procedure-call arguments (including variadic arguments), and the `alloca()` operator. We provide an execution model, a definition of refinement, and an algorithm to soundly convert a refinement check into first-order logic queries that an off-the-shelf SMT solver can handle efficiently. In our experiments, we perform blackbox translation validation of C procedures (with up to 100+ SLOC), involving these local memory allocation constructs, against their corresponding assembly implementations (with up to 200+ instructions) generated by an optimizing compiler with complex loop and vectorizing transformations.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Compilers.*

Additional Key Words and Phrases: Translation validation, Equivalence checking, Certified compilation

ACM Reference Format:
Abhishek Rose and Sorav Bansal. 2024. Modeling Dynamic (De)Allocations of Local Memory for Translation Validation. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 146 (April 2024), 30 pages. https://doi.org/10.1145/3649863

1 INTRODUCTION

Compiler bugs can be catastrophic, especially for safety-critical applications. End-to-End Translation Validation (TV for short) checks a single compilation to ascertain if the machine executable code generated by a compiler agrees with the input source program. In our work, we validate translations from *unoptimized* IR of a C program to *optimized* executable (or assembly) code, which forms an overwhelming majority of the complexity in an end-to-end compilation pipeline. In this setting, the presence of dynamic allocations and deallocations due to local variables and procedure-call arguments in the IR program presents a special challenge — in these cases, the identification and modeling of relations between a local variable (or a procedure-call argument) in IR and its stack address in assembly is often required to complete the validation proof.

Unlike IR-to-assembly, modeling dynamic local memory allocations is significantly simpler for IR-to-IR TV [Kasampalis et al. 2021; Lopes et al. 2021; Menendez et al. 2016; Namjoshi and Zuck 2013; Necula 2000; Stepp et al. 2011; Tristan et al. 2011; Zhao et al. 2012, 2013]. For example, (pseudo)register-allocation of local variables can be tackled by identifying relational invariants that equate the value contained in a local variable's memory region (in the original program) with the value in the corresponding pseudo-register (in the transformed program) [Kang et al. 2018]. If the

Authors' addresses: Abhishek Rose, Indian Institute of Technology Delhi, New Delhi, India, abhishek.rose@cse.iitd.ac.in; Sorav Bansal, Indian Institute of Technology Delhi, New Delhi, India, sbansal@iitd.ac.in.

ACM 2475-1421/2024/4-ART146
https://doi.org/10.1145/3649863

address of a local variable is observable by the C program (e.g., for an address-taken local variable), we need to additionally relate the variable addresses across both programs. These address correlations can be achieved by first correlating the corresponding allocation statements in both programs (e.g., through their names) and then assuming that their return values are equal. Provenance-based syntactic pointer analyses, that show separation between distinct variables [Andersen 1994; Steensgaard 1996], thus suffice for translation validation across IR-to-IR transformations.

An IR-to-assembly transformation involves the lowering of a memory allocation (deallocation) IR instruction to a stackpointer decrement (increment) instruction in assembly. Further, the stack space in assembly is shared by multiple local variables, procedure-call arguments, and by the potential intermediate values generated by the compiler, e.g., pseudo-register spills. Provenance-based pointer analyses are thus inadequate for showing separation in assembly.

Prior work on IR-to-assembly and assembly-to-assembly TV [Churchill et al. 2019; Gupta et al. 2020; Sewell et al. 2013; Sharma et al. 2013] assumes that local variables are either absent or their addresses are not observed in the program and so they are removed through (pseudo)register-allocation. Similarly, these prior works assume that variadic parameters (and other cases of address-taken parameters) are absent in the program.

Prior work on certified compilation, embodied in CompCert [Leroy 2006], validates its own transformation passes from IR to assembly, and supports both address-taken local variables and variadic parameters. However, CompCert sidesteps the task of having to model dynamic allocations by ensuring that the generated assembly code *preallocates* the space for all local variables and procedure-call arguments at the beginning of a procedure's body. Because preallocation is not possible if the size of an allocation is not known at compile time, CompCert does not support variable-sized local variables or `alloca()`. Moreover, preallocation is prone to stack space wastage. In contrast to a certified compiler, TV needs to validate the compilation of a third-party compiler, and thus needs to support an arbitrary (potentially dynamic) allocation strategy.

Example: Consider a C and a 32-bit x86 assembly program in fig. 1. The `fib` procedure in fig. 1a accepts two integers `n` and `m`, allocates a variable-length array (VLA) `v` of `n+2` elements, computes the first `m+1` fibonacci numbers in `v`, calls `printf()`, and returns the m^{th} fibonacci number. Notice that for an execution free of *Undefined Behaviour* (UB), both `n` and `m` must be non-negative and `m` must be less than `(n+2)`. Note that the memory for local variables (`v` and `i`) and procedure-call arguments (for the call to `printf`) is allocated dynamically through the `alloc` instruction in the IR program (fig. 1b). In the assembly program (fig. 1c), memory is allocated through instructions that manipulate the stackpointer register `esp`.

If the IR program uses an address, say α, of a local variable (e.g., $\alpha \in \{p_{\text{I1}}, p_{\text{I2}}\}$) or a procedure-call argument (e.g., $\alpha \in \{p_{\text{I7}}, p_{\text{I8}}\}$) in its computation (e.g., for pointer arithmetic at lines `I3` and `I5`, or for accessing the variadic argument at p_{I8} within `printf`), validation requires a relation between α and its corresponding stack address in assembly (e.g., p_{I7} = `esp` at line `A14`).

Contributions: We formalize IR and assembly execution semantics in the presence of dynamically (de)allocated memory for local variables and procedure-call arguments, define a notion of correct translation, and provide an algorithm that converts the correctness check to first-order logic queries over bitvectors, arrays, and uninterpreted functions. Almost all production compilers (e.g., GCC) generate assembly code to dynamically allocate stack space for procedure-call arguments at the callsite, e.g., in fig. 1c, the arguments to `printf` are allocated at line `A13`. Ours is perhaps the first effort to enable validation of this common allocation strategy. Further, our work enables translation validation for programs with dynamically-allocated fixed-length and variable-length local variables for a wide set of allocation strategies used by a compiler including stack merging, stack reallocation (if the order of allocations is preserved), and intermittent register allocation.

```
C0: int fib(int n, int m) {
C1:   int v[n+2];
C2:   v[0]=0; v[1]=1;
C3:   for(int i=2; i<=m; i++)
C4:     v[i]=v[i-1]+v[i-2];
C5:   printf("fib(%d)␣=␣%d", m, v[m]);
C6:   return v[m];
C7: }
```

(a) C Program with VLA.

```
I0:  int fib(int* n, int* m):
I1:    i=p_I1=alloc 1,int,4;
I2:    v=p_I2=alloc *n+2,int,4;
I3:    v[0]=0; v[1]=1; *i=2;
I4:    if(*i >_s *m) goto I7;
I5:      v[*i]=v[*i-1]+v[*i-2];
I6:      (*i)++; goto I4;
I7:    p_I7=alloc 1,char*,4;
I8:    p_I8=alloc 1,struct{int; int;},4;
I9:    *p_I7=__S__; *p_I8=*m; *(p_I8 + 4)=v[*m];
I10:   t=call int printf(p_I7, p_I8);
I11:   dealloc p_I7;
I12:   dealloc p_I8;
I13:   r=v[*m];
I14:   dealloc p_I2;
I15:   dealloc p_I1;
I16:   ret r;
```

(b) (Abstracted) IR.

```
A0:    fib:
A1:      push ebp; ebp = esp;
A2:      push {edi, esi, ebx}; esp = esp-12;
A3:      eax = mem_4[ebp+8]; ebx = mem_4[ebp+12];
A4:      esp = esp-(0xFFFFFFF0 & (4*(eax+2)+15));
A4.1:    v_I1 = alloc_v 4,4,I1;
A4.2:    alloc_s esp,4*(eax+2),4,I2;
A5:      esi = ((esp+3)/4)*4;
A6:      mem_4[esi] = 0; mem_4[esi+4] = 1;
A7:      if(ebx ≤_s 1) jmp A12;
A8:      edi = 0; edx = 1; eax = 2;
A9:       ecx = edx+edi; edi = edx; edx = ecx;
A10:      mem_4[esi+4*eax] = ecx; eax = eax+1;
A11:      if(eax ≤_s ebx) jmp A9;
A12:     edi = mem_4[esi+4*ebx];  esp = esp-4;
A13:     push {edi, ebx, __S__}; //__S__ is ptr to format string
A13.1:   alloc_s esp,  4,4,I7;
A13.2:   alloc_s esp+4,8,4,I8;
A14:     call int printf(<char*> esp, <struct{int; int;}> esp+4)
             G ∪ {hp, cl, I7, I8};
A14.1:   dealloc_s I7;
A14.2:   dealloc_s I8;
A15:     eax = edi;
A15.1:   dealloc_s I2;
A15.2:   dealloc_v I1;
A16:     esp = ebp-12; pop {ebx, esi, edi, ebp};
A17:     ret;
```

(c) (Abstracted) 32-bit x86 Assembly Code.

Fig. 1. Example program with VLA and its lowerings to IR and assembly. Subscripts $_s$ and $_u$ denote signed and unsigned comparison respectively. Bold font (parts of) instructions are added by our algorithm.

2 EXECUTION SEMANTICS AND NOTION OF CORRECT TRANSLATION

We are interested in showing that an x86 assembly program $\mathbb{A}$ is a correct translation of the unoptimized IR representation of a C program $\mathbb{C}$. Prior TV efforts identify a lockstep correlation between (potentially unrolled) iterations of loops in the two programs to show equivalence [Churchill et al. 2019]. These correlations can be represented through a *product program* that executes $\mathbb{C}$ and $\mathbb{A}$ in lockstep, using a careful choice of program path correlations, to keep the machine states of both programs related at the ends of correlated paths [Gupta et al. 2020; Zaks and Pnueli 2008].

Our TV algorithm additionally attempts to identify a lockstep correlation between the dynamic (de)allocation events and procedure-call events performed in both programs, i.e., we require the order and values of these execution events to be identical in both programs. To identify a lockstep correlation, our algorithm annotates $\mathbb{A}$ with (de)allocation instructions and procedure-call arguments. Our key insight is to define a *refinement relation* between $\mathbb{C}$ and $\mathbb{A}$ through the existence of an annotation in $\mathbb{A}$. We also generalize the definition of a product program so it can be used to witness refinement in the presence of non-determinism due to addresses of dynamically-allocated local memory, UB, and stack overflow.

Overview through example: In $\mathbb{C}$, an alloc instruction returns a non-deterministic address of the newly allocated region with non-deterministic contents, e.g., in fig. 1b, the address ($p_{\mathtt{I2}}$) and initial contents of VLA v allocated at I2 are non-deterministic. In fig. 1c, our algorithm annotates an $\mathtt{alloc}_s$ instruction at A4.2 to correlate in lockstep with I2, so that $p_{\mathtt{I2}}$'s determinized value is identified through its first operand (esp). An $\mathtt{alloc}_s$ instruction allocates a contiguous address interval from the stack, starting at esp in this case, to a local variable. The second (4*(eax+2)), third (4), and fourth (I2) operands of $\mathtt{alloc}_s$ specify the allocation size in bytes, required alignment, and the PC of the correlated allocation instruction in $\mathbb{C}$ (which also identifies the local variable) respectively. The determinized values of the initial contents of VLA v at I2 are identified to be equal to the contents of the stack region [esp,esp+4*(eax+2)-1] at A4.1. A corresponding $\texttt{dealloc}_s$ instruction, that correlates in lockstep with I14, is annotated at A15.1 to free the memory allocated by A4.2 (both have operand I2) and return it to stack.

A procedure call appears as an x86 call instruction and we annotate the actual arguments as its operands in $\mathbb{A}$. In fig. 1c, the two operands (esp and esp+4) annotated at A14 are the determinized values of p_{I7} and p_{I8}, as obtained through x86 calling conventions. The last annotation at A14 is the set of memory regions (e.g., G, hp, cl, …, as described in section 2.2.2) observable by printf in $\mathbb{A}$ — this is equal to the set of memory regions observable by printf in $\mathbb{C}$, as obtained through an over-approximate points-to analysis. Annotations of $\texttt{alloc}_s$ at A13.{1,2} and $\texttt{dealloc}_s$ at A14.{1,2} identify the memory regions occupied by printf's parameters during printf's execution.

Consider the local variable i, allocated at I1, with address p_{I1} in fig. 1b. Because i's address is never taken in the source program, a correlation of p_{I1} with its determinized value in $\mathbb{A}$'s stack is not necessarily required. Further, the compiler may register-allocate i in which case no stack address exists for i, e.g., i lives in eax at A8-A11 in fig. 1c. The $\texttt{alloc}_v$ instruction annotated at A4.1 performs a "virtual allocation" for variable i in lockstep with I1. The first (4), second (4), and third (I1) operands of $\texttt{alloc}_v$ indicate the allocation size, required alignment, and the PC of the correlated allocation in $\mathbb{C}$ respectively. The corresponding $\texttt{dealloc}_v$ instruction, annotated at A15.2, correlates in lockstep with I15. The address and initial contents of the memory allocated by $\texttt{alloc}_v$ are chosen non-deterministically in $\mathbb{A}$, and are assumed to be equal to the address and initial contents of memory allocated by a correlated alloc in $\mathbb{C}$, e.g., $\text{v}_{\text{I1}} = p_{\text{I1}}$ at A4.1. A "virtually-allocated region" is never used by $\mathbb{A}$. We introduce the $\texttt{(de)alloc}_{s,v}$ instructions formally in section 2.4.

Consider the memory access v[*i] at I5 in fig. 1b, and assume we identify a lockstep correlation of this memory access with the assembly program's access $\texttt{mem}_4$[esi+4*eax] at A10 in fig. 1c, with value relations esi=v and eax=*i. We need to cater to the possibility where $\texttt{*i} >_s \texttt{*n+2}$ (equivalently, $\texttt{eax} >_s \texttt{mem}_4\texttt{[ebp+12]+2}$), which would trigger UB in $\mathbb{C}$, and may go out of variable bounds in stack in assembly. Our product program encodes the necessary UB semantics that allow anything to happen in assembly (including out of bound stack access) if UB is triggered in $\mathbb{C}$.

Finally, consider the stackpointer decrement instruction at A4 in fig. 1c. If eax (which corresponds to *n) is too large, this instruction at A4 may potentially overflow the stack space. Our product program encodes the assumption that an assembly program will have the necessary stack space required for execution, which is necessary to be able to validate a translation from IR to assembly.

Thus, we are interested in identifying *legal* annotations of $\texttt{(de)alloc}_{s,v}$ instructions and operands of procedure-call instructions in $\mathbb{A}$, such that the execution behaviours of $\mathbb{A}$ can be shown to refine the execution behaviours of $\mathbb{C}$, assuming $\mathbb{A}$ has the required stack space for execution. We show refinement separately for each procedure C in $\mathbb{C}$ and its corresponding implementation A in $\mathbb{A}$. Thereafter, a coinductive argument shows refinement for full programs $\mathbb{C}$ and $\mathbb{A}$ starting at the main() procedure. We do not support inter-procedural transformations.

Paper organization: Sections 2.1 to 2.3 describe a procedure's execution semantics for both IR and assembly representations. Refinement, through annotations, is defined in section 2.4. Section 3 defines a product program and its associated requirements such that refinement can be witnessed, and section 4 provides an algorithm to automatically construct such a product program.

2.1 Intermediate and Assembly Representations

2.1.1 IR. The unoptimized IR used to represent $\mathbb{C}$ is mostly a subset of LLVM — it supports all the primitive types (integer, float, code labels) and the derived types (pointer, array, struct, procedure) of LLVM. Being unoptimized, our IR does not need to support LLVM's undef and poison values,

va_start(ap, *last*)	va_arg(ap, τ)	va_copy(aq, ap)	va_end(ap)
a := va_start_ptr	a := load void∗, 4, $\langle\!| ap |\!\rangle$	a := load void∗, 4, $\langle\!| ap |\!\rangle$	store void∗, 4, 0, $\langle\!| ap |\!\rangle$
store void∗, 4, a, $\langle\!| ap |\!\rangle$	*result* := load $\langle\!| \tau |\!\rangle$, $\langle\!|$alignof(τ)$|\!\rangle$, a	store void∗, 4, a, $\langle\!| aq |\!\rangle$	
	a' := a + $\langle\!|$roundup$_4$(sizeof(τ))$|\!\rangle$		
	store void∗, 4, a', $\langle\!| ap |\!\rangle$		

Fig. 2. Translation of C's variadic macros to LLVM$_d$ instructions. roundup$_4(a)$ returns the closest multiple of 4 greater than or equal to a.

it instead treats all error conditions as UB. Syntactic conversion of C to LLVM IR entails the usual conversion of types/operators. A global variable name g or a parameter name y appearing in a C procedure body is translated to the variable's start address in IR, denoted $\boxed{\mathtt{lb}.g}$ and $\boxed{\mathtt{lb}.y}$ respectively[1]. A local variable declaration or a call to C's `alloca()` operator is converted to LLVM's `alloca` instruction, and to distinguish the two, we henceforth refer to the latter as the "`alloc`" instruction. Unlike LLVM, our IR also supports a `dealloc` instruction that deallocates a variable at the end of its scope — we use LLVM's `stack{save,restore}` intrinsics (that maintain equivalent scope information for a different purpose) to introduce explicit `dealloc` instructions in our IR. Henceforth, we refer to our IR as LLVM$_d$ (for LLVM + `dealloc`).

We discuss our logical model in the context of compilation to 32-bit x86 for the relative simplicity of the calling conventions in 32-bit mode. Like LLVM, a procedure definition in LLVM$_d$ can only return a scalar value — aggregate return value is passed in memory. Unlike LLVM which allocates memory for a parameter only if its address is taken, LLVM$_d$ allocates memory for all parameters — LLVM$_d$ thus takes all parameters through pointers, e.g., both `n` and `m` are passed through pointers in fig. 1b. This makes the translation of a procedure-call from C to LLVM$_d$ slightly more verbose, as explicit instructions to (de)allocate memory for the arguments are required. An example of this translation is shown in fig. 1 where a call to `printf` at `C5` of fig. 1a translates to instructions `I7-I12` in fig. 1b: the LLVM$_d$ program performs two allocations, one for the format string, and another for the variable argument list; the latter represented as an object of "`struct`" type containing two `int`s. The call instruction takes the pointers returned by these allocations as operands.

Figure 2 shows the C-to-LLVM$_d$ translations for variadic macros. The translation rules have template holes marked by $\langle\!|\,|\!\rangle$ for types and variables of C which are populated at the time of translation with appropriate LLVM$_d$ entities. LLVM$_d$'s `va_start_ptr` instruction returns the first address of the current procedure's variable argument list.

2.1.2 Assembly. Broadly, an assembly program $\mathbb{A}$ consists of a code section (with a sequence of assembly instructions), a data section (with read-only and read-write global variables), and a symbol table that maps string symbols to memory addresses in code and data sections. The validator checks that the regions specified by the symbol table are well-aligned and non-overlapping, and uses it to relate a global variable or procedure in $\mathbb{C}$ to its address or implementation in $\mathbb{A}$.

We assume that the OS guarantees the caller-side contract of the ABI calling conventions for the entry procedure, `main()`. For 32-bit x86, this means that at the start of program execution, the stackpointer is available in register `esp`, and the return address and input parameters `(argc,argv)` to `main()` are available in the stack region just above the stackpointer. For other procedure-calls, the validator verifies the adherence to calling conventions at a callsite (in the caller) and assumes adherence at procedure entry (in the callee). Heap allocation procedures like `malloc()` are left uninterpreted, and so, the only compiler-visible way to allocate (and deallocate) memory in $\mathbb{A}$ is through the decrement (and increment) of the stackpointer stored in register `esp`.

2.1.3 Allocation and Deallocation. Allocation and deallocation instructions appear only in $\mathbb{C}$, and do not appear in $\mathbb{A}$. Let C represent a procedure in program $\mathbb{C}$.

[1]As we will also see later, $\boxed{\mathtt{lb}.v}$ denotes the *lower bound* of the memory addresses occupied by variable with name v.

An $\texttt{LLVM}_d$ instruction "p^a_C: `v := alloc n,` τ`, align`" allocates a contiguous region of local memory with space for `n` elements of type τ aligned by `align`, and returns its start address in `v`. The PC, p^a_C, of an `alloc` instruction is also called an *allocation site* (denoted by z), and let the set of allocation sites in C be Z. During conversion of the C program to $\texttt{LLVM}_d$, we distinguish between allocation sites due to the declaration of a local variable (or a procedure-call argument) and allocation sites due to a call to `alloca()` — we use Z_l for the former and Z_a for the latter, so that $Z = Z_l \cup Z_a$.

The address of an allocated region is non-deterministic, but is subject to two *Well-Formedness (WF) constraints*: (1) the newly allocated memory region should be separate from all currently allocated memory regions, i.e., there should be no overlap; and (2) the address of the newly allocated memory region should be aligned by `align`.

An $\texttt{LLVM}_d$ instruction "p^d_C: `dealloc` z" deallocates *all* local memory regions allocated due to the execution of allocation site $z \in Z$.

2.2 Transition Graph Representation

An $\texttt{LLVM}_d$ or assembly instruction may mutate the machine state, transfer control, perform I/O, or terminate the execution. We represent a C procedure, C, as a transition graph, $C = (\mathcal{N}_C, \mathcal{E}_C)$, with a finite set of nodes $\mathcal{N}_C = \{n^s = n_1, n_2, \ldots, n_m\}$, and a finite set of labeled directed edges $\mathcal{E}_C$. A unique node n^s represents the start node or entry point of C, and every other node n_j must be reachable from n^s. A node with no outgoing edges is a *terminating node.* A variable in C is identified by its scope-resolved unique name. The machine state of C consists of the set of input parameters $\overrightarrow{y}$, set of temporary variables $\overrightarrow{t}$, and an explicit array variable M_C denoting the current state of memory. We use $\texttt{i}_N$ to denote a bitvector type of size $N > 0$. M_C is of type $\mathsf{T}(M_C) = \texttt{i}_{32} \rightarrow \texttt{i}_8$.

An assembly implementation of the C procedure C, identified through the symbol table, is the assembly procedure A. The machine state of A consists of its hardware registers $\overrightarrow{regs}$ and memory M_A. Similarly to C, $A = (\mathcal{N}_A, \mathcal{E}_A)$ is also represented as a transition graph.

Let $P \in \{C, A\}$. In addition to the memory (data) state M_P, we also need to track the allocation state, i.e., the set of intervals of addresses that have been allocated by the procedure. We use α (potentially with a subscript) to denote a memory address of bitvector type. Let $i = [\alpha_b, \alpha_e]$ represent an *address interval* starting at α_b and ending at α_e (both inclusive), such that $\alpha_b \leq_u \alpha_e$ (where $\leq_u$ is unsigned comparison operator). Let $[\alpha]_w$ be a shorthand for the address interval $[\alpha, \alpha + w - 1_{\texttt{i}_{32}}]$, where $n_{\texttt{i}_{32}}$ is the two's complement representation of integer n using 32 bits.

2.2.1 Address set. Let Σ (potentially with a sub- or superscript) represent a set of addresses, or an *address set*. An empty address set is represented by $\emptyset$, and an address set of contiguous addresses is an interval i. Two address sets overlap, written $\mathsf{ov}(\Sigma_1, \Sigma_2)$, iff $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. Extended to $m \geq 2$ sets, $\mathsf{ov}(\Sigma_1, \Sigma_2, \ldots, \Sigma_m) \Leftrightarrow \exists_{1 \leq j_1 < j_2 \leq m} \mathsf{ov}(\Sigma_{j_1}, \Sigma_{j_2})$. $|\Sigma|$ represents the number of distinct addresses in Σ. For a non-empty address set, $\mathsf{lb}(\Sigma)$ and $\mathsf{ub}(\Sigma)$ represent the smallest and largest address respectively in Σ. $\mathsf{comp}(\Sigma)$ represents the *complement* of Σ, so that: $\forall \alpha : (\alpha \in \Sigma) \Leftrightarrow (\alpha \notin \mathsf{comp}(\Sigma))$.

2.2.2 Memory regions. To support dynamic (de)allocation, an execution model needs to individually track regions of memory belonging to each variable, heap, stack, etc. We next describe the memory regions tracked by our model.

(1) Let G be the set of names of all global variables in $\mathbb{C}$. For each global variable $g \in G$, we track the memory region belonging to that variable. We use the name of a global variable $g \in G$ as its *region identifier* to identify the region belonging to g in both C and A.
(2) For a procedure C, let Y be the set of names of formal parameters, including the variadic parameter, if present. We use the special name `vrdc` for the variadic parameter. The memory region belonging to a parameter $y \in Y$ is called y in both C and A.

(3) The memory region allocated by allocation site $z \in Z$ is called z in C. In A, our algorithm potentially annotates allocation instructions corresponding to an allocation site z in C.
(4) hp denotes the region belonging to the *program heap* (managed by the OS) in both C and A.
(5) Local variables and actual arguments may be allocated by the *call chain* of a procedure (caller, caller's caller, and so on). This is denoted by region cl, or *callers' locals*, in both C and A.
(6) In procedure A, stack memory can be allocated and deallocated through stackpointer decrement and increment. The addresses belonging to the stack frame of A (but not to a local variable $z \in Z$ or a parameter $y \in Y$) belong to the stk (stack) region in A. The stk region is absent in C.
(7) Similarly, in A, we use cs (*callers' stack*) to identify the region that belongs to the stack space (but not to cl) of the call-chain of procedure A. cs is absent in C.
(8) Program $\mathbb{A}$ may use more global memory than $\mathbb{C}$, e.g., to store pre-computed constants to implement vectorizing transformations. Let F be the set of names of all *assembly-only global variables* in $\mathbb{A}$. For each $f \in F$, its memory region in A is identified by f.
(9) The region $\texttt{free}$ denotes the free space, that does not belong to any of the aforementioned regions, in both C and A,

Let $R = G \cup Y \cup Z \cup F \cup \{hp, cl, stk, cs, \texttt{free}\}$ represent all *region identifiers*; $B = G \cup Y \cup Z \cup \{hp, cl\}$ denote the regions in *b*oth C and A; and $S = \{stk, cs\}$ denote the stack regions in A.

Let $G_r \subseteq G$ be the set of read-only global variables in $\mathbb{C}$; and, let $G_w = G \setminus G_r$ denote the set of read-write global variables. We define $F_r \subseteq F$ and $F_w = F \setminus F_r$ analogously.

For each non-free region $r \in (R \setminus \{\texttt{free}\})$, the machine state of a procedure P includes a unique variable Σ_P^r that tracks region r's address set as P executes. If Σ_P^r is a contiguous non-empty interval, we also refer to it as i_P^r. For $r \in G \cup Y \cup F \cup \{hp, cl, cs\}$, Σ_P^r remains constant throughout P's execution. For $\vec{r} \subseteq R$, we define an expression $\Sigma_P^{\vec{r}} = \bigcup_{r \in \vec{r}} \Sigma_P^r$. Because C does not have a stack or an assembly-only global variable, $\Sigma_C^{F \cup S} = \emptyset$ holds throughout C's execution. At any point in P's execution, the free space can be computed as $\Sigma_P^{\texttt{free}} = \texttt{comp}(\Sigma_P^{B \cup F \cup S})$. Notice that we do not use an explicit variable to track $\Sigma_P^{\texttt{free}}$.

2.2.3 Ghost variables. Our validator introduces *ghost variables* in a procedure's execution semantics, i.e., variables that were not originally present in P. We use $\boxed{x}$ to indicate that x is a ghost variable. For each region $r \in G \cup Y \cup Z$ ($r \in F$), we introduce $\boxed{\texttt{em}.r}$, $\boxed{\texttt{lb}.r}$, and $\boxed{\texttt{ub}.r}$ in C (A) to track the *emptiness* (whether the region is empty), *lower bound* (smallest address), and *upper bound* (largest address) of Σ_C^r (Σ_A^r) respectively; for $r \in G \cup Y$ ($r \in F$), $\boxed{\texttt{sz}.r}$ tracks the size of Σ_C^r (Σ_A^r), and for $z \in Z$, $\boxed{\texttt{lstSz}.z}$ tracks the *size of last allocation* due to allocation-site z. $\boxed{\Sigma_P^{\texttt{rd}}}$ and $\boxed{\Sigma_P^{\texttt{wr}}}$ track the set of addresses read and written by P respectively. Let $\boxed{+}$ be the set of all ghost variables.

2.2.4 Error codes. Execution of C or A may terminate successfully, may never terminate, or may terminate with an error. We support two error codes to distinguish between two categories of errors: $\mathscr{U}$ and $\mathscr{W}$. In C: $\mathscr{U}$ represents an occurrence of UB, and $\mathscr{W}$ represents a violation of a WF constraint that needs to be ensured either by the compiler or the OS (both external to the program itself). In A: $\mathscr{U}$ represents UB or a translation error, and $\mathscr{W}$ represents occurrence of a condition that can be assumed to never occur, e.g., if the OS ensures that it never occurs. In summary, for a procedure P, $\mathscr{W}$ represents an error condition that P can *assume* to be absent (because the external environment ensures it), while $\mathscr{U}$ represents an error that P must *ensure* to be absent.

2.2.5 Outside world and observable trace. Let Ω_P be a state of the outside world (OS/hardware) for P that supplies external inputs whenever P reads from it, and consumes external outputs generated by P. Ω_P is assumed to mutate arbitrarily but deterministically based on the values consumed or produced due to the I/O operations performed by P during execution. Let T_P be a potentially-infinite sequence of observable trace events generated by an execution of P.

2.2.6 Expressions. Let variable v and variables $\vec{v}$ or $\vec{x}$ be drawn from $\mathtt{Vars} = (\vec{t}, \overrightarrow{regs}, M_P, \Sigma^r_P, \boxed{+})$ (for all $P \in \{C, A\}$ and for all $r \in (R \setminus \{\mathtt{free}\})$); $e(\vec{x})$ be an expression over $\vec{x}$, and $E(\vec{x})$ be a list of expressions over $\vec{x}$. An expression $e(\vec{x})$ is a well-formed combination of constants, variables $\vec{x}$, and arithmetic, logical, relational, memory access, and address set operators. For memory reads and writes, `select` (`sel` for short) and `store` (`st` for short) operations are used to access and modify M_P at a given address α. Further, the `sel` and `st` operators are associated with a `sz` parameter: $\mathtt{sel}_{\mathtt{sz}}(\mathtt{arr}, \alpha)$ returns a little-endian concatenation of `sz` bytes starting at α in the array `arr`. Similarly, $\mathtt{st}_{\mathtt{sz}}(\mathtt{arr}, \alpha, \mathtt{data})$ returns a new array that has contents identical to `arr` except for the `sz` bytes starting at α which have been replaced by `data` in little-endian format. To encode reads/writes to a region of memory, we define projection and updation operations.

Definition 2.1. $\pi_\Sigma(M_P)$ denotes the *projection* of M_P on addresses in Σ, i.e., if $M'_P = \pi_\Sigma(M_P)$, then $\forall \alpha \in \Sigma : \mathtt{sel}_1(M'_P, \alpha) = \mathtt{sel}_1(M_P, \alpha)$ and $\forall \alpha \notin \Sigma : \mathtt{sel}_1(M'_P, \alpha) = 0$. The sentinel value 0 is used for the addresses outside Σ. We use $M_{P_1} =_\Sigma M_{P_2}$ as shorthand for $(\pi_\Sigma(M_{P_1}) = \pi_\Sigma(M_{P_2}))$.

Definition 2.2. $\mathsf{upd}_\Sigma(M_P, M)$ denotes the *updation* of M_P on addresses in Σ using the values in M. If $M'_P = \mathsf{upd}_\Sigma(M_P, M)$, then $M'_P =_\Sigma M$ and $M'_P =_{\mathsf{comp}(\Sigma)} M_P$ hold.

2.2.7 Instructions. Each edge $e_P \in \mathcal{E}_P$ is labeled with one of the following *graph instructions*:

(1) A *simultaneous assignment* of the form $\vec{v} := E(\vec{x})$. Because variables $\vec{v}$ and $\vec{x}$ may include M_P, an assignment suffices for encoding memory loads and stores. Similarly, because the variables may be drawn from Σ^z_P (for an allocation site z), an assignment is also used to encode the allocation of an interval i_{new} ($\Sigma^z_P := \Sigma^z_P \cup i_{\mathsf{new}}$) and the deallocation of all addresses allocated due to z ($\Sigma^z_P := \emptyset$). Stack allocation and deallocation in A can be similarly represented as $\Sigma^{stk}_A := \Sigma^{stk}_A \cup i_{\mathsf{new}}$ and $\Sigma^{stk}_A := \Sigma^{stk}_A \setminus i_{\mathsf{new}}$ respectively.

(2) A *guard* instruction, $e(\vec{x})?$, indicates that when execution reaches its head, the edge is taken iff its *edge condition* $e(\vec{x})$ evaluates to `true`. For every other instruction, the edge is always taken upon reaching its head, i.e., its edge condition is `true`. For a non-terminating node $n_P \in \mathcal{N}_P$, the guards of all edges departing from n_P must be mutually exclusive, and their disjunction must evaluate to `true`.

(3) A type-parametric *choose* instruction $\theta(\vec{\tau})$. Instruction $\vec{v} := \theta(\vec{\tau})$ non-deterministically chooses values of types $\vec{\tau}$ and assigns them to variables $\vec{v}$, e.g., a memory with non-deterministic contents is obtained by using $\theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8)$.

(4) A *read* (`rd`) or *write* (`wr`) I/O instruction. A read instruction $\vec{v} := \mathtt{rd}(\vec{\tau})$ reads values of types $\vec{\tau}$ from the outside world into variables $\vec{v}$, e.g., an address set is read using $\Sigma := \mathtt{rd}(2^{\mathtt{i}_{32}})$.
A write instruction $\mathtt{wr}(V(E(\vec{x})))$ writes the value constructed by value constructor V using $E(\vec{x})$ to the outside world. A value constructor is defined for each type of observable event. For a procedure-call, $\mathtt{fcall}(\rho, \vec{v}, \vec{r}, M)$ represents value constructed for a procedure-call to callee with name (or address) ρ, the actual arguments $\vec{v}$, callee-observable regions $\vec{r}$, and memory M. Similarly, $\mathtt{ret}(E(\vec{x}))$ is a value constructed during procedure return that captures observable values computed through $E(\vec{x})$. Local (de)allocation events have their own value constructors, $\mathtt{allocBegin}(z, w)$, $\mathtt{allocEnd}(z, i, M)$, and $\mathtt{dealloc}(z)$, which represent (de)allocation due to allocation site z with the associated observables w (size), i (interval), and M (memory).
A read or write instruction mutates Ω_P arbitrarily based on the read and written values. Further, the data items read or written are appended to the observable trace T_P. Let $\mathtt{read}_{\vec{\tau}}(\Omega_P)$ be an uninterpreted function that reads values of types $\vec{\tau}$ from Ω_P; and $\mathtt{io}(\Omega_P, \vec{v}, \mathtt{rw})$ be an uninterpreted function that returns an updated state of Ω_P after an I/O operation of type $\mathtt{rw} \in \{\mathtt{r}, \mathtt{w}\}$ (read or write) with values $\vec{v}$. Thus, in its explicit syntax, $\vec{v} := \mathtt{rd}(\vec{\tau})$ translates to a sequence of instructions: $\vec{v} := \mathtt{read}_{\vec{\tau}}(\Omega_P)$; $\Omega_P := \mathtt{io}(\Omega_P, \vec{v}, \mathtt{r})$; $T_P := T_P \cdot \vec{v}$, where $\cdot$ is the

Operator	Definition
$\mathtt{sz}(\tau)$	Returns the size (in bytes) of type τ. For example, $\mathtt{sz}(\mathtt{i}_{32}) = 4$, $\mathtt{sz}(\mathtt{i}_8*) = 4$, and $\mathtt{sz}([80 \times \mathtt{i}_8]) = 80$.
$\mathtt{T}(a)$	Returns the type τ of a where a can be a global variable, a parameter, or a register. For example, $\mathtt{T}(\mathtt{eax}) = \mathtt{i}_{32}$.
$\triangle_\tau(\mathtt{eax}, \mathtt{edx})$	A macro operator which derives the return value of an assembly procedure with return type τ from input registers eax and edx using the calling conventions, e.g., $\triangle_{\mathtt{i}_8}(\mathtt{eax}, \mathtt{edx}) = \mathtt{extract}_{7,0}(\mathtt{eax})$, $\triangle_{\mathtt{i}_{32}}(\mathtt{eax}, \mathtt{edx}) = \mathtt{eax}$, $\triangle_{\mathtt{i}_{64}}(\mathtt{eax}, \mathtt{edx}) = \mathtt{concat}(\mathtt{edx}, \mathtt{eax})$, where $\mathtt{extract}_{h,l}(a)$ extracts bits h down to l from a and $\mathtt{concat}(a, b)$ returns the bitvector concatenation of a and b where b takes the less significant position.
$\triangledown_\tau(v)$	Inverse of $\triangle_\tau(\mathtt{eax}, \mathtt{edx})$. Distributes the packed bitvector v of type τ into two bitvectors of 32 bit-width each, setting the bits not covered by v to some non-deterministic value.
$\mathtt{ROM}^r_P(i)$	Returns a memory array containing the contents of read-only global variable named r in P. The contents are mapped at the addresses in the provided interval i.
$\mathtt{addrSets}_F()$	Returns the address sets of the assembly-only global variables F using the symbol table in $\mathbb{A}$.

Predicate	Definition
$\mathtt{aligned}_n(a)$	Bitvector a is at least n bytes aligned. Equivalent to: $a \% n = 0$, where % is remainder operator.
$\mathtt{isAlignedIntrvl}_a(p, w)$	A w-sized sequence of addresses starting at p is aligned by a and does not wraparound. Equivalent to: $\mathtt{aligned}_a(p) \wedge (p \leq_u p + w - 1_{\mathtt{i}_{32}})$.
$\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma)$	Equivalent to: $\mathtt{isAlignedIntrvl}_{\mathtt{align}(a)}(p, \mathtt{sz}(\tau)) \wedge ([p]_{\mathtt{sz}(\tau)} \subseteq \Sigma)$.
$\mathtt{addrSetsAreWF}(\Sigma^{hp}_P, \Sigma^{cl}_P, \ldots, i^g_P, \ldots, \Sigma^f_P, \ldots, i^y_P, \ldots, \Sigma^{\mathtt{vrdc}}_P)$	The address sets passed as parameter are well-formed with respect to C semantics. Equivalent to: $(0_{\mathtt{i}_{32}} \notin \Sigma^{G \cup F \cup Y \cup \{hp,cl\}}_P) \wedge \neg\mathtt{ov}(\Sigma^{hp}_P, \Sigma^{cl}_P, \ldots, i^g_P, \ldots, \Sigma^f_P, \ldots, i^y_P, \ldots, \Sigma^{\mathtt{vrdc}}_P) \wedge (\Sigma^{\mathtt{vrdc}}_P \neq \emptyset \Rightarrow \mathtt{isInterval}(\Sigma^{\mathtt{vrdc}}_P)) \wedge \forall_{r \in G \cup (Y \setminus \mathtt{vrdc}) \cup F}(\lvert i^r_P \rvert = \mathtt{sz}(\mathtt{T}(r)) \wedge \mathtt{aligned}_{\mathtt{algnmnt}(r)}(\mathtt{lb}(i^r_P)))$, where $\mathtt{isInterval}(\Sigma^{\mathtt{vrdc}}_P)$ holds iff the address set $\Sigma^{\mathtt{vrdc}}_P$ is an interval, $\mathtt{algnmnt}(r)$ returns the alignment of variable r.
$\mathtt{intrvlInSet}(\alpha_b, \alpha_e, \Sigma)$	The pair (α_b, α_e) forms a valid interval inside the address set Σ. Equivalent to: $(\alpha_b \neq 0_{\mathtt{i}_{32}}) \wedge (\alpha_b \leq_u \alpha_e) \wedge ([\alpha_b, \alpha_e] \subseteq \Sigma)$
$\mathtt{intrvlInSet}_a(\alpha_b, \alpha_e, \Sigma)$	Equivalent to: $\mathtt{aligned}_a(\alpha_b) \wedge \mathtt{intrvlInSet}(\alpha_b, \alpha_e, \Sigma)$
$\mathtt{obeyCC}(e_{\mathtt{esp}}, \vec{\tau}, \vec{x})$	Pointers $\vec{x}$ match the expected addresses of arguments for a procedure-call in assembly. Based on the calling conventions, obeyCC uses the value of the current stackpointer ($e_{\mathtt{esp}}$) and parameter types ($\vec{\tau}$) to obtain the expected addresses of the arguments. For example, $\mathtt{obeyCC}(\mathtt{esp}, (\mathtt{i}_8, \mathtt{i}_{32}), (\mathtt{esp}, \mathtt{esp} + 4_{\mathtt{i}_{32}}))$ holds.
$\mathtt{overflow}_{mul}(a, b)$	Signed multiplication of bitvectors a, b overflows. E.g., $\mathtt{overflow}_{mul}(2147483647_{\mathtt{i}_{32}}, 2_{\mathtt{i}_{32}})$ holds.
$\mathtt{stkIsWF}(\mathtt{esp}, \boxed{\mathtt{stk}_e}, \boxed{\mathtt{cs}_e}, \vec{\tau}, \Sigma^{hp}_A, \Sigma^{cl}_A, \Sigma^{G \cup F}_A, \ldots, i^y_A, \ldots, \Sigma^{\mathtt{vrdc}}_A)$	The pairs $(\mathtt{esp}, \boxed{\mathtt{stk}_e})$, $(\boxed{\mathtt{stk}_e}, \boxed{\mathtt{cs}_e})$ represent well-formed intervals for initial stk and cs regions with respect to parameter types $\vec{\tau}$ and other (input) address sets in A. Equivalent to: $\mathtt{aligned}_{16}(\mathtt{esp} + 4_{\mathtt{i}_{32}}) \wedge (\mathtt{esp} \leq_u \mathtt{esp} + 4_{\mathtt{i}_{32}}) \wedge \neg\mathtt{ov}([\mathtt{esp}]_{4_{\mathtt{i}_{32}}}, \Sigma^{G \cup F \cup Y \cup \{hp,cl\}}_A) \wedge \mathtt{obeyCC}(\mathtt{esp} + 4_{\mathtt{i}_{32}}, \vec{\tau}, \ldots, \mathtt{lb}(i^y_A), \ldots) \wedge (\boxed{\mathtt{stk}_e} <_u \boxed{\mathtt{cs}_e}) \wedge \neg\mathtt{ov}([\boxed{\mathtt{stk}_e} + 1_{\mathtt{i}_{32}}, \boxed{\mathtt{cs}_e}], \Sigma^{G \cup F \cup \{hp\}}_A) \wedge \Sigma^{cl}_A \subseteq [\boxed{\mathtt{stk}_e} + 1_{\mathtt{i}_{32}}, \boxed{\mathtt{cs}_e}]$
$\mathtt{UB}_P(\mathtt{op}, \vec{x})$	Application of operation op of procedure P on arguments $\vec{x}$ triggers UB. E.g., $\mathtt{UB}_C(\mathtt{udiv}, (1_{\mathtt{i}_{32}}, 0_{\mathtt{i}_{32}}))$ holds.

Table 1. Definitions of operators and predicates used in translations in figs. 3 to 6

trace concatenation operator. Similarly, $\mathtt{wr}(V(E(\vec{x})))$ translates to: $\Omega_P := \mathtt{io}(\Omega_P, V(E(\vec{x})), \mathtt{w})$; $T_P := T_P \cdot V(E(\vec{x}))$. Henceforth, we only use the implicit syntax for brevity.

(5) An error-free and error-indicating *halt* instruction that terminates execution. $\mathtt{halt}(\emptyset)$ indicates termination without error, and $\mathtt{halt}(\mathscr{r})$ indicates termination with error code $\mathscr{r} \in \{\mathscr{U}, \mathscr{W}\}$. Upon termination without error, a special `exit` event is appended to observable trace T_P. Upon termination with error, the error code is appended to T_P. Thus, the destination of an edge with a `halt` instruction is a terminating node. We create a unique terminating node for an error-free exit. We also create a unique terminating node for each error code, also called an *error node*; an edge terminating at an error node is called an *error edge*. $\mathscr{U}_P$ and $\mathscr{W}_P$ represent error nodes in P for error $\mathscr{U}$ and $\mathscr{W}$ respectively. Execution transfers to an error node upon encountering the corresponding error. Let $\mathcal{N}^{\cancel{\mathscr{UW}}}_P = \mathcal{N}_P \setminus \{\mathscr{U}_P, \mathscr{W}_P\}$ be the set of non-error nodes in P.

In addition to the observable trace events generated by `rd`, `wr`, and `halt` instructions, the execution of every instruction in P also appends an observable *silent trace event*, denoted $\perp$, to T_P. Silent trace events count the number of executed instructions as a proxy for observing the passage of time.

2.3 Translations of C and A to their Graph Representations

Figures 3 and 4 (and figs. 5 and 6 later) present the key translation rules from $\mathtt{LLVM}_d$ and (abstracted) assembly instructions to graph instructions. Each rule is composed of three parts separated by a horizontal line segment: on the left is the name of the rule, above the line segment is the $\mathtt{LLVM}_d$/assembly instruction, and below the line segment is the graph instructions listing. We describe the operators and predicates used in the rules in table 1. As an example, the top right

$$(\textsc{Entry}_C)\ \frac{p_C^j : \mathtt{def}\ C(\vec{\tau})}{\begin{array}{l}\Sigma_C^{hp}, \Sigma_C^{cl}, \ldots, i_C^{g}, \ldots, i_C^{y}, \ldots, \Sigma_C^{\mathsf{vrdc}} := \mathbf{rd}(2^{\mathtt{i}32}, 2^{\mathtt{i}32}, \ldots, 2^{\mathtt{i}32});\\ \Sigma_C^{stk}, \Sigma_C^{cs}, \ldots, \Sigma_C^{f}, \ldots, \Sigma_C^{z}, \ldots, \boxed{\Sigma_C^{\mathsf{rd}}}, \boxed{\Sigma_C^{\mathsf{wr}}} := \emptyset, \emptyset, \ldots, \emptyset;\\ \underline{\mathtt{if}}\ (\neg\mathtt{addrSetsAreWF}(\Sigma_C^{hp}, \Sigma_C^{cl}, \ldots, i_C^{g}, \ldots, \Sigma_C^{f}, \ldots, i_C^{y}, \ldots, \Sigma_C^{\mathsf{vrdc}}))\\ \quad \mathtt{halt}(\mathscr{W});\\ M_C := \theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8); \quad M_C := \mathsf{upd}_{\Sigma_C^B}(M_C, \mathbf{rd}(\mathtt{i}_{32} \rightarrow \mathtt{i}_8));\\ \underline{\mathtt{for}}\ g\ \mathsf{in}\ G_r\ \{\ M_C := \mathsf{upd}_{i_C^g}(M_C, \mathsf{ROM}_C^g(i_C^g));\ \}\\ \underline{\mathtt{for}}\ z\ \mathsf{in}\ Z\ \{\ \boxed{\mathsf{em}.z} := \mathsf{true}; \quad \beta_M(z) := \emptyset; \}\\ \underline{\mathtt{for}}\ r\ \mathsf{in}\ G \cup Y \cup \{hp, cl\}\ \{\ \beta_M(r) := G \cup \{hp, cl\};\ \}\\ \underline{\mathtt{for}}\ r\ \mathsf{in}\ G \cup Y\ \{\ \boxed{\mathsf{sz}.r}, \boxed{\mathsf{em}.r} := |\Sigma_C^r|, (|\Sigma_C^r| = 0_{\mathtt{i}_{32}});\\ \quad \underline{\mathtt{if}}(\neg\boxed{\mathsf{em}.r})\ \{\ \boxed{\mathtt{lb}.r}, \boxed{\mathsf{ub}.r} := \mathtt{lb}(\Sigma_C^r), \mathsf{ub}(\Sigma_C^r);\ \} \quad \beta(\boxed{\mathtt{lb}.r}) := \{r\};\\ \}\end{array}}$$

$$(\textsc{Op})\ \frac{p_C^j : v := \mathsf{op}(\vec{x})}{\begin{array}{l}\underline{\mathtt{if}}\ (\mathsf{UB}_C(\mathsf{op}, \vec{x}))\ \mathtt{halt}(\mathscr{U});\\ v := \mathsf{op}(\vec{x}); \quad \beta(v) := \beta^{\mathsf{op}}(\beta(\vec{x}));\end{array}}$$

$$(\textsc{RetV})\ \frac{p_C^j : \mathtt{ret\ void}}{\begin{array}{l}\mathbf{wr}(\mathtt{ret}(\pi_{\Sigma_C^B}(M_C)));\\ \mathtt{halt}(\emptyset);\end{array}}$$

$$(\textsc{Ret}_C)\ \frac{p_C^j : \mathtt{ret}\ v}{\begin{array}{l}\mathbf{wr}(\mathtt{ret}(v, \pi_{\Sigma_C^B}(M_C)));\\ \mathtt{halt}(\emptyset);\end{array}}$$

$$(\textsc{AssignConst})\ \frac{p_C^j : v := c}{v := c; \quad \beta(v) := \emptyset;}$$

$$(\textsc{Alloc})\ \frac{z : v := \ \mathtt{alloc}\ n,\ \tau,\ a}{\begin{array}{l}\mathtt{IF}\{z \in Z_l\}\{\ \underline{\mathtt{if}}\ (n \leq_s 0_{\mathtt{i}_{32}} \vee \mathtt{overflow}_{mul}(n, \mathtt{sz}(\tau)))\ \mathtt{halt}(\mathscr{U});\ \}\\ \mathbf{wr}(\mathtt{allocBegin}(z, n{*}\mathtt{sz}(\tau)));\\ \alpha_b := \theta(\mathtt{i}_{32}); \quad \alpha_e := \alpha_b + n{*}\mathtt{sz}(\tau) - 1_{\mathtt{i}_{32}};\\ \underline{\mathtt{if}}\ (\neg\mathtt{intrvlInSet}_a(\alpha_b, \alpha_e, \Sigma_C^{\mathsf{free}}))\ \mathtt{halt}(\mathscr{W});\\ \Sigma_C^z := \Sigma_C^z \cup [\alpha_b, \alpha_e]; \quad M_C := \mathsf{upd}_{[\alpha_b, \alpha_e]}(M_C, \theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8));\\ \boxed{\mathtt{lb}.z} := \boxed{\mathsf{em}.z}\,?\ \alpha_b : \mathsf{min}(\boxed{\mathtt{lb}.z}, \alpha_b); \quad \boxed{\mathtt{lstSz}.z} := n{*}\mathtt{sz}(\tau);\\ \boxed{\mathsf{ub}.z} := \boxed{\mathsf{em}.z}\,?\ \alpha_e : \mathsf{max}(\boxed{\mathsf{ub}.z}, \alpha_e); \quad \boxed{\mathsf{em}.z} := \mathsf{false};\\ v := \alpha_b; \quad \beta(v) := \{z\};\\ \mathbf{wr}(\mathtt{allocEnd}(z, [\alpha_b, \alpha_e], \pi_{[\alpha_b, \alpha_e]}(M_C)));\end{array}}$$

$$(\textsc{Dealloc})\ \frac{p_C^j : \mathtt{dealloc}\ z}{\begin{array}{l}\Sigma_C^z := \emptyset; \quad \boxed{\mathsf{em}.z} := \mathsf{true};\\ \mathbf{wr}(\mathtt{dealloc}(z));\end{array}}$$

$$(\textsc{VaStartPtr})\ \frac{p_C^j : p := \mathtt{va\_start\_ptr}}{\begin{array}{l}\underline{\mathtt{if}}\ (\Sigma_C^{\mathsf{vrdc}} = \emptyset)\ \{\\ \quad p := 0_{\mathtt{i}_{32}}; \quad \beta(p) := \emptyset;\\ \}\ \underline{\mathtt{else}}\ \{\\ \quad p := \boxed{\mathtt{lb.vrdc}}; \quad \beta(p) := \{\mathsf{vrdc}\};\\ \}\end{array}}$$

$$(\textsc{Load}_C)\ \frac{p_C^j : v := \mathtt{load}\ \tau,\ a,\ p}{\begin{array}{l}\underline{\mathtt{if}}\ (\neg\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma_C^{\beta(p)}))\ \mathtt{halt}(\mathscr{U});\\ v := \mathtt{sel}_{\mathtt{sz}(\tau)}(M_C, p);\\ \beta(v) := \beta_M(\beta(p)); \quad \boxed{\Sigma_C^{\mathsf{rd}}} := \boxed{\Sigma_C^{\mathsf{rd}}} \cup [p]_{\mathtt{sz}(\tau)};\end{array}}$$

$$(\textsc{Store}_C)\ \frac{p_C^j : \mathtt{store}\ \tau,\ a,\ v,\ p}{\begin{array}{l}\underline{\mathtt{if}}\ (\neg\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma_C^{\beta(p) \setminus G_r}))\ \mathtt{halt}(\mathscr{U});\\ M_C := \mathtt{st}_{\mathtt{sz}(\tau)}(M_C, p, v);\\ \beta_M(\beta(p)) := \beta_M(\beta(p)) \cup \beta(v); \quad \boxed{\Sigma_C^{\mathsf{wr}}} := \boxed{\Sigma_C^{\mathsf{wr}}} \cup [p]_{\mathtt{sz}(\tau)};\end{array}}$$

$$(\textsc{CallV})\ \frac{p_C^j : \mathtt{call\ void}\ \rho(\vec{\tau}\ \vec{x})}{\begin{array}{l}\beta^* := \beta_M^*(\bigcup_{x \in \vec{x}} \beta(x) \cup G \cup \{hp\});\\ \mathbf{wr}(\mathtt{fcall}(\rho, \vec{x}, \beta^*, \pi_{\Sigma_C^{\beta^*}}(M_C)));\\ M_C := \mathsf{upd}_{\Sigma_C^{\beta^* \setminus G_r}}(M_C, \mathbf{rd}(\mathtt{i}_{32} \rightarrow \mathtt{i}_8));\\ \beta_M(\beta^* \setminus G_r) := \beta^*;\end{array}}$$

$$(\textsc{Call}_C)\ \frac{p_C^j : v := \mathtt{call}\ \gamma\ \rho(\vec{\tau}\ \vec{x}) \quad \gamma \neq \mathtt{void}}{\begin{array}{l}\beta^* := \beta_M^*(\bigcup_{x \in \vec{x}} \beta(x) \cup G \cup \{hp\});\\ \mathbf{wr}(\mathtt{fcall}(\rho, \vec{x}, \beta^*, \pi_{\Sigma_C^{\beta^*}}(M_C)));\\ M_C := \mathsf{upd}_{\Sigma_C^{\beta^* \setminus G_r}}(M_C, \mathbf{rd}(\mathtt{i}_{32} \rightarrow \mathtt{i}_8));\\ v := \mathbf{rd}(\gamma); \ \beta(v), \beta_M(\beta^* \setminus G_r) := \beta^*, \beta^*;\end{array}}$$

Fig. 3. Translation rules for converting $\mathtt{LLVM}_d$ instructions to graph instructions.

corner of fig. 3 shows the parametric (Op) rule which gives the translation of an operation using arithmetic/logical/relational operator op in $\mathtt{LLVM}_d$ to corresponding graph instructions. We use C-like constructs in graph instructions as syntactic sugar for brevity, e.g. ';' is used for sequencing, '?:' is used for conditional assignment, and <u>if</u>, <u>else</u>, and <u>for</u> are used for control flow transfer. We highlight the read and write I/O instructions with a shaded background, and use **bold**, colored fonts for `halt` instructions. We use macros `IF` and `ELSE` to choose translations based on a boolean condition on the input syntax.

2.3.1 Translation of C. Figure 3 shows the translation rules for converting $\mathtt{LLVM}_d$ instructions to graph instructions. The (Entry$_C$) rule presents the initialization performed at the entry of a procedure C. The address sets and memory state of C are initialized using reads from the outside world Ω_C. The address sets that are read are checked for well-formedness with respect to C semantics, or else error $\mathscr{W}$ is triggered. The ghost variables are also appropriately initialized.

The (Alloc) and (Dealloc) rules provide semantics for the allocation and deallocation of local memory at allocation site z — if $z \in Z_l$, n (the number of elements allocated) has additional constraints for a UB-free execution. A (de)allocation instruction generates observable traces using the `wr` instruction at the beginning and end of each execution of that instruction. We will later use these traces to identify a lockstep correlation of (de)allocation events between C and A, towards validating a translation.

In (Op), an application of op may trigger UB for certain inputs, as abstracted through the $\mathsf{UB}_C(\mathsf{op}, \vec{x})$ operation. While there are many UBs in the C standard, we model only the ones that we have seen getting exploited by the compiler for optimization. These include the UB associated with a logical or arithmetic shift operation (second operand should be bounded by a limit which is determined by the size of the first operand), address computation (no over- and under-flow), and division operation (second operand should be non-zero). In (Load$_C$) and (Store$_C$), a UB-free execution requires the dereferenced pointer p to be non-NULL ($\neq 0_{\mathtt{i}_{32}}$ in our modeling), aligned by a, and have its access interval belong to the regions which p may *point to*, or p may be *based on* (§6.5.6p8 of the C17 standard).

To identify the regions a pointer p may point to, we define two maps: (1) $\beta : \mathtt{Vars} \longrightarrow 2^R$, so that for a (pointer) variable $x \in \mathtt{Vars}$, $\beta(x)$ returns the set of regions x may point to; and (2) $\beta_M : R \longrightarrow 2^R$, so that for a region $r \in R$, $\beta_M(r)$ returns the set of regions that some (pointer) value stored in $\pi_{\Sigma_C^r}(M_C)$ may point to. $\beta(\vec{x})$ is equivalent to $\bigcup_{x\in\vec{x}} \beta(x)$, and $\beta_M(\vec{r})$ is equivalent to $\bigcup_{r\in\vec{r}} \beta_M(r)$. Similarly, $\beta_M(\vec{r_1}) := \vec{r_2}$ is equivalent to '$\mathtt{for}\ r_1\ \mathtt{in}\ \vec{r_1}\ \{\ \beta_M(r_1) := \vec{r_2};\ \}$'. The initialization and updation of β and β_M due to each LLVM_d instruction can be seen in fig. 3. For an operation op, $\beta^{\mathsf{op}} : 2^R \longrightarrow 2^R$ represents the over-approximate abstract transfer function for $v := \mathsf{op}(\vec{x})$, that takes as input $\beta(\vec{x})$ and returns $\beta(v)$. We use $\beta^{\mathsf{op}}(\vec{r}) = \vec{r}$ if op is bitwise complement and unary negation. We use $\beta^{\mathsf{op}}(\vec{r_1}, \ldots, \vec{r_m}) = \bigcup_{1\leq j\leq m} \vec{r_j}$ if op is bitvector addition, subtraction, shift, bitwise-{and,or}, extraction, or concatenation. We use $\beta^{\mathsf{op}}(\vec{r}) = \emptyset$ if op is bitvector multiplication, division, logical, relational or any other remaining operator.

The translation of an LLVM_d procedure-call is given by the rules (CallV) and (Call$_C$) and involves producing non-silent observable trace events using the wr instruction for the callee name/address, arguments, and callee-accessible regions and memory state. To model return values and side-effects to the memory state due to a callee, rd instructions are used. A callee may access a memory region iff it is *transitively reachable* from a global variable $g \in G$, the heap hp, or one of the arguments $x \in \vec{x}$. The transitively reachable memory regions are over-approximately computed through a reflexive-transitive closure of β_M, denoted β_M^*.

A rd instruction clobbers the callee-observable state elements arbitrarily. Thus, if a callee procedure terminates, wr and rd instructions over-approximately model the execution of a procedure-call. Later, our definition of refinement (section 2.4) caters to the case when a callee procedure may not terminate.

2.3.2 *Translation of A.*

The translation rules for converting assembly instructions to graph instructions are shown in fig. 4. The assembly opcodes are abstracted to an IR-like syntax for ease of exposition. For example, in (Load$_A$), a memory read operation is represented by a load instruction which is annotated with address p, access size w (in bytes), and required alignment a (in bytes). Similarly, in (Store$_A$), a memory write operation is represented by a store instruction with similar operands. Both (Load$_A$) and (Store$_A$) translations update the ghost address sets $\boxed{\Sigma_A^{\mathsf{rd}}}$ and $\boxed{\Sigma_A^{\mathsf{wr}}}$, in the same manner as done in C. Exceptions like division-by-zero or memory-access errors are modeled as UB in $\mathbb{A}$ through UB_A (rules (Op-esp) and (Op-Nesp))

(Op-esp) shows the translation of an instruction that updates the stackpointer. Assignment to the stackpointer register esp may indicate allocation (push) or deallocation (pop) of stack space. A stackpointer assignment which corresponds to a stackpointer decrement (push) is identified through predicate $\mathtt{isPush}(p_A^j, \iota_b, \iota_a)$ where ι_b and ι_a are the values of esp *b*efore and *a*fter the execution of the instruction. We use $\mathtt{isPush}(p_A^j, \iota_b, \iota_a) \Leftrightarrow (\iota_b >_u \iota_a)$. While this choice of isPush suffices for most TV settings, we show in our technical report [Rose and Bansal 2024b] that if the translation is

```
(Op-esp)   $p_A^j$ : esp := op($\vec{x}$)
           ------------------------------------------
           if (UB$_A$(op, $\vec{x}$)) halt($\mathscr{U}$);
           $t$ := op($\vec{x}$);
           if (isPush($p_A^j$, esp, $t$)) {
               if ($\neg$intrvlInSet($t$, esp $- 1_{i_{32}}$, $\Sigma_A^{\text{free}}$)) halt($\mathscr{W}$);
               $\Sigma_A^{stk} := \Sigma_A^{stk} \cup [t, \text{esp} - 1_{i_{32}}]$;
               $M_A := \text{upd}_{[t,\text{esp}-1_{i_{32}}]}(M_A, \theta(i_{32} \to i_8))$;
           } else if ($t \neq$ esp) {
               if ($\neg$intrvlInSet(esp, $t - 1_{i_{32}}$, $\Sigma_A^{stk}$)) halt($\mathscr{U}$);
               $\Sigma_A^{stk} := \Sigma_A^{stk} \setminus [\text{esp}, t - 1_{i_{32}}]$;
           }
           esp := $t$;   $\boxed{\text{sp}.p_A^j}$ := $t$;
```

```
(Load$_A$)   $p_A^j$ : $v$ := load $w$, $a$, $p$
             ------------------------------------------
             if (   $\neg$isAlignedIntrvl$_a$($p$, $w$)
                  $\vee$ ov($[p]_w$, $\Sigma_A^{\text{free}}$)) halt($\mathscr{U}$);
             $v$ := sel$_w$($M_A$, $p$);
             $\boxed{\Sigma_A^{\text{rd}}} := \boxed{\Sigma_A^{\text{rd}}} \cup [p]_w$;
```

```
(Store$_A$)   $p_A^j$ : store $w$, $a$, $p$, $v$
              ------------------------------------------
              if (   $\neg$isAlignedIntrvl$_a$($p$, $w$)
                   $\vee$ ov($[p]_w$, $\Sigma_A^{\{\text{free}\} \cup G_r \cup F_r}$))
                halt($\mathscr{U}$);
              $M_A$ := st$_w$($M_A$, $p$, $v$);
              $\boxed{\Sigma_A^{\text{wr}}} := \boxed{\Sigma_A^{\text{wr}}} \cup [p]_w$;
```

```
(Entry$_A$)   $p_A^j$ : def $A(\vec{\tau})$
              ------------------------------------------
              $\Sigma_A^{hp}, \Sigma_A^{cl}, \ldots, i_A^g, \ldots, i_A^y, \ldots, \Sigma_A^{\text{vrdc}}$ := rd($2^{i_{32}}, 2^{i_{32}}, \ldots, 2^{i_{32}}$);
              $\ldots, \Sigma_C^f, \ldots$ := addrSets$_F$();      $\ldots, \Sigma_A^z, \ldots := \ldots, \emptyset, \ldots$;
              if ($\neg$addrSetsAreWF($\Sigma_A^{hp}, \Sigma_A^{cl}, \ldots, i_A^g, \ldots, \Sigma_A^f, \ldots, i_A^y, \ldots, \Sigma_A^{\text{vrdc}}$))
                halt($\mathscr{W}$);
              $M_A := \theta(i_{32} \to i_8)$;   $M_A := \text{upd}_{\Sigma_A^B}(M_A, \textbf{rd}(i_{32} \to i_8))$;
              for $r$ in $G_r \cup F_r$ { $M_A := \text{upd}_{i_A^r}(M_A, \text{ROM}_A^r(i_A^r))$; }
              for $r$ in $\overrightarrow{regs}$ { $r := \theta(\text{T}(r))$; }
              $\boxed{\text{stk}_e} := \Sigma_A^Y \neq \emptyset \,?\, \text{ub}(\Sigma_A^Y) : \text{esp} + 3_{i_{32}}$;      $\boxed{\text{cs}_e} := \theta(i_{32})$;
              if ($\neg$stkIsWF(esp, $\boxed{\text{stk}_e}$, $\boxed{\text{cs}_e}$, $\vec{\tau}, \Sigma_A^{hp}, \Sigma_A^{cl}, \Sigma_A^{G \cup F}, \ldots, i_A^y, \ldots, \Sigma_A^{\text{vrdc}}$))
                halt($\mathscr{W}$);
              $\Sigma_A^{stk} := [\text{esp}, \boxed{\text{stk}_e}] \setminus \Sigma_A^Y$;   $\Sigma_A^{cs} := [\boxed{\text{stk}_e} + 1_{i_{32}}, \boxed{\text{cs}_e}] \setminus \Sigma_A^{cl}$;
              $\boxed{\text{sp}.entry}$ := esp;   $\boxed{M^{cs}} := \pi_{\Sigma_A^{cs}}(M_A)$;   $\boxed{\Sigma_A^{\text{rd}}}, \boxed{\Sigma_A^{\text{wr}}} := \emptyset, \emptyset$;
              $\boxed{ebp}, \boxed{esi}, \boxed{edi}, \boxed{ebx}, \boxed{eip}$ := ebp, esi, edi, ebx, sel$_4$($M_A$, esp);
              for $f$ in $F$ { $\boxed{\text{sz}.f}, \boxed{\text{em}.f}, \boxed{\text{lb}.f}, \boxed{\text{ub}.f} := |\Sigma_A^f|, |\Sigma_A^f| = 0_{i_{32}}, \text{lb}(\Sigma_A^f), \text{ub}(\Sigma_A^f)$; }
```

```
(Op-Nesp)   $p_A^j$ : $r$ := op($\vec{x}$)      $r \neq$ esp
            ------------------------------------------
            if (UB$_A$(op, $\vec{x}$)) halt($\mathscr{U}$);
            $r$ := op($\vec{x}$);
```

```
(Ret$_A$)   $p_A^j$ : ret $\tau$
            ------------------------------------------
            if (   $\boxed{\text{sp}.entry} \neq$ esp
                 $\vee$ $\boxed{ebp} \neq$ ebp $\vee$ $\boxed{esi} \neq$ esi
                 $\vee$ $\boxed{edi} \neq$ edi $\vee$ $\boxed{ebx} \neq$ ebx
                 $\vee$ $\boxed{eip} \neq$ sel$_4$($M_A$, esp)
                 $\vee$ $\neg$($\boxed{M^{cs}} =_{\Sigma_A^{cs}} M_A$)) halt($\mathscr{U}$);
            IF{$\tau$ = void}{ wr($\pi_{\Sigma_A^B}(M_A)$); }
            ELSE{
             wr(ret($\triangle_\tau$(eax, edx), $\pi_{\Sigma_A^B}(M_A)$));
            }
            halt($\emptyset$);
```

Fig. 4. Translation rules for converting pseudo-assembly instructions to graph instructions.

performed by an adversarial compiler, discriminating a stack push from a pop is trickier and may require external trusted guidance from the user. For a stackpointer decrement, a failure to allocate stack space, either due to wraparound or overlap with other allocated space, triggers $\mathscr{W}$, i.e., we expect the environment (e.g., OS) to ensure that the required stack space is available to A. For a stackpointer increment, it is a translation error if the stackpointer moves out of stack frame bounds (captured by error code $\mathscr{U}$). The stackpointer value at the end of an assignment instruction at PC p_A^j is saved in a ghost variable named $\boxed{\text{sp}.p_A^j}$. These ghost variables help with inference of invariants that relate a local variable's address with stack addresses (discussed in section 4.1). During push, the initial contents of the newly allocated stack region are chosen non-deterministically using θ — this admits the possibility of arbitrary clobbering of the unallocated stack region below the stackpointer due to asynchronous external interrupts, before it is allocated again.

(Entry$_A$) shows the initialization of state elements of procedure A. For region $r \in B$, the initialization of Σ_A^r and $\pi_{\Sigma_A^r}(M_{\mathbb{A}})$ is similar to (Entry$_C$). The address sets of all assembly-only regions $f \in F$ are initialized using $\mathbb{A}$'s symbol table (addrSets$_F$()). The memory contents of a read-only global variable $r \in G_r \cup F_r$ are initialized using ROM$_A^r(i_A^r)$ (defined in table 1). The machine registers are initialized with arbitrary contents (θ) — the constraints on the esp register are checked later, and $\mathscr{W}$ is generated if a constraint is violated. The x86 stack of an assembly procedure includes the stack frame Σ_A^{stk} of the currently executing procedure A, the parameters Σ_A^Y of A, and the remaining space which includes caller-stack Σ_A^{cs} and, possibly, the locals Σ_A^{cl} defined in the call chain of A. Ghost variable $\boxed{\text{sp}.entry}$ holds the esp value at entry of A. $\boxed{\text{stk}_e}$ represents the largest address in $\Sigma_A^{Y \cup \{stk\}}$ so that at entry, $\Sigma_A^{stk} = [\boxed{\text{sp}.entry}, \boxed{\text{stk}_e}] \setminus \Sigma_A^Y$. If there are no parameters, $\boxed{\text{stk}_e} = \text{esp} + 3_{i_{32}}$ represents the end of the region that holds the return address. Ghost variable

$\boxed{\mathtt{cs}_e}$ holds the largest address in $\Sigma_A^{\{stk,cs,cl\}\cup Y}$. At entry, due to the calling conventions, we assume (through `stkIsWF()`) that: (1) the parameters are laid out at addresses above the stackpointer as per calling conventions (`obeyCC`); (2) the value $\mathtt{esp} + 4_{\mathtt{i}_{32}}$ is 16-byte aligned; and (3) the caller stack is above A's stack frame Σ_A^{stk}. A violation of these conditions trigger $\mathscr{W}$.

Upon return (rule (RET$_A$)), we require that the callee-save registers, caller stack, and the return address remain preserved — a violation of these conditions trigger $\mathscr{U}$. For simplicity, we only tackle scalar return values, and ignore aggregate return values that need to be passed in memory.

2.4 Observable traces and Refinement Definition

Recall that a procedure execution yields an observable trace containing silent and non-silent events. The error code of a trace T, written $e(T)$, is either $\emptyset$ (indicating either non-termination or error-free termination), or one of $\mathscr{r} \in \{\mathscr{U}, \mathscr{W}\}$ (indicating termination with error code $\mathscr{r}$). The non-error part of a trace T, written $\tilde{e}(T)$, is T when $e(T) = \emptyset$, and T' such that $T = T' \cdot e(T)$ otherwise.

Definition 2.3. $P \downarrow_\Omega T$ denotes the condition that for an initial outside world Ω, the execution of a procedure P may produce an observable trace T (for some sequence of non-deterministic choices).

Definition 2.4. Traces T and T' are *stuttering equivalent*, written $T =_{st} T'$, iff they differ only by finite sequences of silent events $\bot$. A trace T is a *stuttering prefix* of trace T', written $T \leq_{st} T'$, iff $(T' =_{st} T) \vee (\exists T^{\mathsf{r}} : T' =_{st} (T \cdot T^{\mathsf{r}}))$.

Definition 2.5. $U_{\mathsf{pre}}^{\Omega,T_A}(C)$ denotes the condition: $\exists T_C : (C \downarrow_\Omega T_C \cdot \mathscr{U}) \wedge (T_C \leq_{st} T_A)$.

Definition 2.6. $W_{\mathsf{pre}}^{\Omega,T_A}(C)$ denotes the condition: $e(T_A) = \mathscr{W} \wedge (\exists T_C : (C \downarrow_\Omega T_C) \wedge (\tilde{e}(T_A) \leq_{st} T_C))$

Definition 2.7. $C \sqsupseteq A$, read A refines C (or C is refined by A), iff:

$$\forall \Omega : (A \downarrow_\Omega T_A) \Rightarrow (W_{\mathsf{pre}}^{\Omega,T_A}(C) \vee U_{\mathsf{pre}}^{\Omega,T_A}(C) \vee (\exists T_C : (C \downarrow_\Omega T_C) \wedge (T_A =_{st} T_C)))$$

The $W_{\mathsf{pre}}^{\Omega,T_A}(C)$ and $U_{\mathsf{pre}}^{\Omega,T_A}(C)$ conditions cater to the cases where A triggers $\mathscr{W}$ and C triggers $\mathscr{U}$ respectively; the constituent $\leq_{st}$ conditions ensure that a procedure call in A has identical termination behaviour to a procedure-call in C before an error is triggered. If neither A triggers $\mathscr{W}$ nor C triggers $\mathscr{U}$, $T_A =_{st} T_C$ ensures that A and C produce identical non-silent events at similar speeds. In the absence of local variables and procedure-calls in C, $C \sqsupseteq A$ implies a correct translation from C to A.

2.4.1 Refinement definition in the presence of local variables and procedure-calls when all local variables are allocated on the stack in A.

For each local variable (de)allocation and for each procedure-call, our execution semantics generate a `wr` trace event in C (fig. 3). Thus, to reason about refinement, we require correlated and equivalent trace events to be generated in A. For this, we annotate A with two types of annotations to obtain $\dot{A}$:

(1) $\mathtt{alloc}_s$ and $\mathtt{dealloc}_s$ instructions are added to explicitly indicate the (de)allocation of a local variable $z \in Z$, e.g., a stack region may be marked as belonging to z through these annotations.
(2) A procedure-call, direct or indirect, is annotated with the types and addresses of the arguments and the set of memory regions observable by the callee.

These annotations are intended to encode the correlations with the corresponding allocation, deallocation, and procedure-call events in the source procedure C. For now, we assume that the locations and values of these annotations in $\dot{A}$ are coming from an oracle — later in section 4, we present an algorithm to identify these annotations automatically in a best-effort manner.

Figure 5 presents three new instructions in $\dot{A}$ — $\mathtt{alloc}_s$, $\mathtt{dealloc}_s$, and `call` — and their translations to graph instructions.

$$(\textsc{AllocS})\ \frac{p^j_{\dot{A}} : \texttt{alloc}_s\ e_v,\ e_w,\ a,\ z}{\begin{array}{l}\mathbf{wr}(\texttt{allocBegin}(z, e_w));\quad v, w := e_v, e_w;\\ \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i}_{32}}, \Sigma^{stk}_{\dot{A}}))\ \texttt{halt}(\mathscr{U});\\ \Sigma^{stk}_{\dot{A}}, \Sigma^{z}_{\dot{A}} := \Sigma^{stk}_{\dot{A}} \setminus [v]_w, \Sigma^{z}_{\dot{A}} \cup [v]_w;\\ \mathbf{wr}(\texttt{allocEnd}(z, [v]_w, \pi_{[v]_w}(M_{\dot{A}})));\end{array}}$$

$$(\textsc{DeallocS})\ \frac{p^j_{\dot{A}} : \texttt{dealloc}_s\ z}{\begin{array}{l}\Sigma^{z}_{\dot{A}}, \Sigma^{stk}_{\dot{A}} := \emptyset, \Sigma^{stk}_{\dot{A}} \cup \Sigma^{z}_{\dot{A}};\\ \mathbf{wr}(\texttt{dealloc}(z));\end{array}}$$

$$(\textsc{Call}_{\dot{A}})\ \frac{p^j_{\dot{A}} : \texttt{call}\ \gamma\ \rho(\vec{\tau}\ \vec{x})\ \beta^*}{\begin{array}{l}\underline{\texttt{if}}\ (\neg\texttt{aligned}_{16}(\texttt{esp}) \vee \neg\texttt{obeyCC}(\texttt{esp}, \vec{\tau}, \vec{x}))\\ \quad \texttt{halt}(\mathscr{U});\\ \mathbf{wr}(\texttt{fcall}(\rho, \vec{x}, \beta^*, \pi_{\Sigma^{\beta^*}_{\dot{A}}}(M_{\dot{A}})));\\ M_{\dot{A}} := \texttt{upd}_{\Sigma^{\beta^*\setminus G_r}_{\dot{A}}}(M_{\dot{A}}, \mathbf{rd}(\texttt{i}_{32} \to \texttt{i}_8));\\ \texttt{ecx} := \theta(\texttt{i}_{32});\\ \texttt{IF}\{\gamma = \texttt{void}\}\{\ \texttt{eax}, \texttt{edx} := \theta(\texttt{i}_{32}, \texttt{i}_{32});\ \}\\ \texttt{ELSE}\{\ \texttt{eax}, \texttt{edx} := \nabla_\gamma(\mathbf{rd}(\gamma));\ \}\end{array}}$$

Fig. 5. Additional translation rules for converting pseudo-assembly instructions to graph instructions for procedures with only stack-allocated locals.

An instruction '$p^j_{\dot{A}}$: $\texttt{alloc}_s$ e_v, e_w, a, z' represents the stack allocation of a local variable identified by allocation site z. e_v is the expression for start address, e_w is the expression for allocation size, and a is the required alignment of the start address. During stack allocation of a local variable (AllocS), the allocated address must satisfy the required alignment and separation constraints, or else $\mathscr{U}$ is triggered. An allocation removes an address interval from $\Sigma^{stk}_{\dot{A}}$ and adds it to $\Sigma^{z}_{\dot{A}}$.

A '$p^j_{\dot{A}}$: $\texttt{dealloc}_s$ z' instruction represents the deallocation of z and empties the address set $\Sigma^{z}_{\dot{A}}$, adding the removed addresses to $\Sigma^{stk}_{\dot{A}}$ (DeallocS).

For procedure-calls ($\textsc{Call}_{\dot{A}}$), we annotate the `call` instruction in assembly to explicitly specify the start addresses of the address regions belonging to the arguments (shown as $\vec{x}$ in fig. 5). The address region of an argument should have previously been demarcated using an $\texttt{alloc}_s$ instruction. Additionally, these address regions should satisfy the constraints imposed by the calling conventions (`obeyCC`). The calling conventions also require the `esp` value to be 16-byte aligned. A procedure-call is recorded as an observable event, along with the observation of the callee name (or address), the addresses of the arguments, callee-observable regions and their memory contents. The returned values, modeled through $\texttt{rd}(\texttt{i}_{32} \to \texttt{i}_8)$ and $\texttt{rd}(\gamma)$, include the contents of the callee-observable memory regions and the scalar values returned by the callee (in registers `eax`, `edx`). The callee additionally clobbers the caller-save registers using θ.

Definition 2.8 (Refinement in the presence of only stack-allocated locals). $C \succ A$ iff: $\exists \dot{A} : C \sqsupseteq \dot{A}$

$C \succ A$ encodes the property that it is possible to annotate A to obtain $\dot{A}$ so that the local variable (de)allocation and procedure-call events of C and the annotated $\dot{A}$ can be correlated in lockstep. In the presence of stack-allocated local variables and procedure-calls, $C \succ A$ implies a correct translation from C to A. In the absence of local variables and procedure calls, $C \succ A$ reduces to $C \sqsupseteq A$ with $\dot{A} = A$.

2.4.2 Capabilities and limitations of $C \succ A$. $C \succ A$ requires that for allocations and procedure calls that reuse the same stack space, their relative order remains preserved. This requirement is sound but may be too strict for certain (arguably rare) compiler transformations that may reorder the (de)allocation instructions that reuse the same stack space. Our refinement definition admits intermittent register-allocation of (parts of) a local variable.

$C \succ A$ supports *merging* of multiple allocations into a single stackpointer decrement instruction. Let p^s_A be the PC of a single stackpointer decrement instruction that implements multiple allocations. Merging can be encoded by adding multiple $\texttt{alloc}_s$ instructions to A, in the same order as they appear in C, to obtain $\dot{A}$, so that these $\texttt{alloc}_s$ instructions execute only after p^s_A executes; similarly, the corresponding $\texttt{dealloc}_s$ instructions must execute before a stackpointer increment instruction deallocates this stack space.

CompCert's preallocation is a special case of merging where stack space for all local variables is allocated in the assembly procedure's prologue and deallocated in the epilogue (with no reuse of

$$(\textsc{AllocV})\ \frac{p^j_{\ddot{A}} : v := \texttt{alloc}_v\ e_w,\ a,\ zl}{\begin{array}{l}\texttt{wr}(\texttt{allocBegin}(zl, e_w));\quad v, w := \theta(\texttt{i}_{32}), e_w;\\ \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i}_{32}}, \texttt{comp}(\Sigma^B_{\ddot{A}})))\ \texttt{halt}(\mathscr{W});\\ \Sigma^{zl}_{\ddot{A}}|^v := \Sigma^{zl}_{\ddot{A}}|^v \cup [v]_w;\\ \texttt{wr}(\texttt{allocEnd}(zl, [v]_w, \pi_{[v]_w}(M_{\ddot{A}})));\end{array}}$$

$$(\textsc{DeallocV})\ \frac{p^j_{\ddot{A}} : \texttt{dealloc}_v\ zl}{\begin{array}{l}\Sigma^{zl}_{\ddot{A}}|^v := \emptyset;\\ \texttt{wr}(\texttt{dealloc}(zl));\end{array}}$$

$$(\textsc{Entry}_{\ddot{A}})\ \frac{p^j_{\ddot{A}} : \texttt{def}\ \ddot{A}(\vec{\tau})}{\begin{array}{l}\ldots\\ (\text{same as fig. 4})\\ \ldots\\ \underline{\texttt{for}}\ z\ \texttt{in}\ Z_l\ \{\ \Sigma^z_{\ddot{A}}|^s, \Sigma^z_{\ddot{A}}|^v := \emptyset, \emptyset;\ \}\end{array}}$$

$$(\textsc{AllocS'})\ \frac{p^j_{\ddot{A}} : \texttt{alloc}_s\ e_v,\ e_w,\ a,\ z}{\begin{array}{l}\ldots\\ \underline{\texttt{if}}\ (\texttt{ov}([v]_w, \Sigma^{Z_l}_{\ddot{A}}|^v))\ \texttt{halt}(\mathscr{W});\\ \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i}_{32}}, \Sigma^{stk}_{\dot{A}}))\ \texttt{halt}(\mathscr{U});\\ \cancel{\Sigma^{stk}_{\dot{A}}, \Sigma^z_{\dot{A}} := \Sigma^{stk}_{\dot{A}} \setminus [v]_w, \Sigma^z_{\dot{A}} \cup [v]_w;}\\ \texttt{IF}\{z \in Z_l\}\ \{\ \Sigma^{stk}_{\ddot{A}}, \Sigma^z_{\ddot{A}}|^s := \Sigma^{stk}_{\ddot{A}} \setminus [v]_w, \Sigma^z_{\ddot{A}}|^s \cup [v]_w;\ \}\\ \texttt{ELSE}\ \{\ \Sigma^{stk}_{\ddot{A}}, \Sigma^z_{\ddot{A}} := \Sigma^{stk}_{\ddot{A}} \setminus [v]_w, \Sigma^z_{\ddot{A}} \cup [v]_w;\ \}\\ \ldots\end{array}}$$

$$(\textsc{Load}_{\ddot{A}})\ \frac{p^j_{\ddot{A}} : v := \texttt{load}\ w\ a\ p}{\begin{array}{l}\ldots\\ \texttt{ov}([p]_w, \Sigma^{\texttt{free}}_{\ddot{A}} \cup ((\Sigma^{Z_l}_{\ddot{A}}|^v) \setminus \Sigma^{F \cup S}_{\ddot{A}}))\\ \ldots\end{array}}$$

$$(\textsc{Store}_{\ddot{A}})\ \frac{p^j_{\ddot{A}} : \texttt{store}\ w\ a\ p\ v}{\begin{array}{l}\ldots\\ \texttt{ov}([p]_w, \Sigma^{\{\texttt{free}\} \cup Gr \cup Fr}_{\ddot{A}} \cup ((\Sigma^{Z_l}_{\ddot{A}}|^v) \setminus \Sigma^{Fw \cup S}_{\ddot{A}}))\\ \ldots\end{array}}$$

$$(\textsc{Op-esp'})\ \frac{p^j_{\ddot{A}} : \texttt{esp} := \texttt{op}(\vec{x})}{\begin{array}{l}\ldots\\ \texttt{intrvlInSet}(t, \texttt{esp} - 1_{\texttt{i}_{32}}, \Sigma^{\texttt{free}}_{\ddot{A}} \cup \Sigma^{Z_l}_{\ddot{A}}|^v)\\ \ldots\end{array}}$$

Fig. 6. Additional and revised translation rules for converting pseudo-assembly instructions to graph instructions for procedures with both stack and register allocated (or eliminated) locals.

stack space). In this case, our approach annotates A with (de)alloc$_s$ instructions, potentially in the middle of the procedure body, such that they execute in lockstep with the (de)allocations in C.

A compiler may *reallocate* stack space by reusing the same space for two or more local variables with non-overlapping lifetimes (potentially without an intervening stackpointer increment instruction). If the relative order of (de)allocations is preserved, reallocation can be encoded by annotating $\dot{A}$ with a `dealloc`$_s$ instruction (for deallocating the first variable) immediately followed by an `alloc`$_s$ instruction, such that the allocated region potentially overlaps with the previously deallocated region. Our refinement definition may not be able to cater to a translation that changes the relative order of (de)allocation instructions during reallocation.

Because our execution model observes each (de)allocation event (due to the `wr` instruction), a successful refinement check ensures that the allocation states of $\dot{A}$ and C are identical at every correlated callsite. An inductive argument over $\mathbb{C}$ and $\mathbb{A}$ is thus used to show that the address set for region identifier cl is identical at the beginning of each correlated pair of procedures C and A (as modeled through identical reads from the outside world in ($\textsc{Entry}_P$) ($P \in \{C, A\}$) of figs. 3 and 4).

2.4.3 Refinement definition in the presence of potentially register-allocated or eliminated local variables in A. If a local variable $zl \in Z_l$ is either register-allocated or eliminated in A, there exists no stack region in A that can be associated with zl. However, recall that our execution model observes each allocation event in C through the `wr` instruction. Thus, for a successful refinement check, a correlated allocation event still needs to be annotated in A. We pretend that a correlated allocation occurs in A by introducing the notion of a *virtual allocation* instruction, called `alloc`$_v$, in A. Figure 6 shows the virtual (de)allocation instructions, `alloc`$_v$ and `dealloc`$_v$, and the revised translations of procedure-entry and `alloc`$_s$, `dealloc`$_s$, `load`, `store`, and esp-modifying instructions. Instead of reproducing the full translations, we only show the changes with appropriate context. The additions have a highlighted background and deletions are ~~striked out~~. We update and annotate A with the translations and instructions in figs. 5 and 6 to obtain $\ddot{A}$.

A '$p^j_{\ddot{A}}$: v := `alloc`$_v$ e_w, a, z' instruction non-deterministically chooses the start address (using $\theta(\texttt{i}_{32})$) of a local variable z of size e_w and alignment a, performs a virtual allocation, and returns the start address in v. The chosen start address is *assumed* to satisfy the desired WF constraints, such as separation (non-overlap) and alignment; error $\mathscr{W}$ is triggered otherwise. Notice that this is in contrast to $\texttt{alloc}_s$ where error $\mathscr{U}$ is triggered on WF violation to indicate that it is the compiler's responsibility to ensure the satisfaction of WF constraints. Unlike a stack allocation where the compiler chooses the allocated region (and the validator identifies it through an $\texttt{alloc}_s$ annotation), a virtual allocation is only a validation construct (the compiler is not involved) that is used only to enforce a lockstep correlation of allocation events. By triggering $\mathscr{W}$ on a failure during a virtual allocation, we effectively assume that allocation through $\texttt{alloc}_v$ satisfies the required WF conditions.

For simplicity, we support virtual allocations only for a variable declaration $zl \in Z_l$. Thus, we expect a call to `alloca()` at $za \in Z_a$ to always be stack-allocated in $\ddot{A}$. In $\ddot{A}$, we replace the single variable $\Sigma^{zl}_{\ddot{A}}$ with two variables $\Sigma^{zl}_{\ddot{A}}|^s$ and $\Sigma^{zl}_{\ddot{A}}|^v$ that represent the address sets corresponding to the stack and virtual-allocations due to allocation-site zl respectively. We compute $\Sigma^{zl}_{\ddot{A}} = \Sigma^{zl}_{\ddot{A}}|^s \cup \Sigma^{zl}_{\ddot{A}}|^v$ (but we do not maintain a separate variable $\Sigma^{zl}_{\ddot{A}}$). We also assume that a single variable declaration zl in C may either correlate with only stack-allocations (through $\texttt{alloc}_s$) or only virtual-allocations (through $\texttt{alloc}_v$) in $\ddot{A}$[2], i.e., $\Sigma^{zl}_{\ddot{A}}|^s \cap \Sigma^{zl}_{\ddot{A}}|^v = \emptyset$ holds at all times. For convenience, we define $\Sigma^{Z_l}_{\ddot{A}}|^v = \bigcup_{zl \in Z_l}(\Sigma^{zl}_{\ddot{A}}|^v)$.

Importantly, a virtual allocation must be separate from other C allocated regions (B) but may overlap with assembly-only regions ($F \cup S$). Thus, in the revised semantics of (Op-esp'), a stack push is allowed to overstep a virtually-allocated region.

The revised semantics of the $\texttt{alloc}_s$ instruction (AllocS') assume that stack-allocated local memory is separate from virtually-allocated regions. The revised semantics of memory access instructions ((Load$_{\ddot{A}}$) and (Store$_{\ddot{A}}$)) enforce that a virtually-allocated region must never be accessed in $\ddot{A}$, unless it also happens to belong to the assembly-only regions ($F \cup S$).

Effectively, a lockstep correlation of virtual allocations in $\ddot{A}$ with allocations in C ensures that the allocation states of both procedures always agree for regions $r \in B$.

Definition 2.9 (Refinement with stack and virtually-allocated locals). $C \Supset A$ iff: $\exists \ddot{A} : C \sqsupseteq \ddot{A}$

Recall that $C \sqsupseteq \ddot{A}$ requires that *for all* non-deterministic choices of a virtually allocated local variable address in $\ddot{A}$ (v in (AllocV)), there *exists* a non-deterministic choice for the correlated local variable address in C (v in (Alloc) in fig. 3) such that: if $\ddot{A}$'s execution is well-formed (does not trigger $\mathscr{W}$), and C's execution is UB-free (does not trigger $\mathscr{U}$), then the two allocated intervals are identical (the observable values created through `allocBegin` and `allocEnd` must be equal).

In the presence of potentially register-allocated and eliminated local variables, $C \Supset A$ implies a correct translation from C to A. If all local variables are allocated in stack, $C \Supset A$ reduces to $C \succ A$ with $\ddot{A} = \dot{A}$. Figure 1c is an example of an annotated $\ddot{A}$.

3 WITNESSING REFINEMENT THROUGH A DETERMINIZED CROSS-PRODUCT $\ddot{A} \boxtimes C$

We first introduce program paths and their properties. Let $P \in \{C, \ddot{A}\}$. Let $e_P = (n_P \to n^t_P) \in \mathcal{E}_P$ represent an edge from node n_P to node n^t_P, both drawn from $\mathcal{N}_P$. A *path* ξ_P from n_P to n^t_P, written $\xi_P = n_P \twoheadrightarrow n^t_P$, is a sequence of $m \geq 0$ edges $(e^1_P, e^2_P, \ldots, e^m_P)$ with $\forall_{1 \leq j \leq m} : e^j_P = (n^{f,j}_P \to n^{t,j}_P) \in \mathcal{E}_P$, such that $n^{f,1}_P = n_P$, $n^{t,m}_P = n^t_P$, and $\bigwedge_{j=1}^{m-1}(n^{t,j}_P = n^{f,j+1}_P)$. Nodes n_P and n^t_P are called the *source* and *sink* nodes of ξ_P respectively. Edge e^j_P (for some $1 \leq j \leq m$) is said to be present in ξ_P, written $e^j_P \in \xi_P$. An empty sequence, written ϵ, represents the *empty path*. The *path condition* of a path

[2]For simplicity, we do not tackle path-specializing transformations that may require, for a single variable declaration zl, a stack-allocation on one assembly path and a virtual-allocation on another. Such transformations are arguably rare.

$\xi_P = n_P \twoheadrightarrow n_P^t$, written $pathcond(\xi_P)$, is a conjunction of the edge conditions of the constituent edges. Starting at n_P, $pathcond(\xi_P)$ represents the condition that ξ_P executes to completion.

A sequence of edges corresponding to a shaded statement in the translations (figs. 3 to 6) is distinguished and identified as an *I/O path*. An I/O path must contain either a single `rd` or a single `wr` instruction. For example, the sequence of edges corresponding to "$\texttt{wr}(\texttt{fcall}(\rho, \vec{x}, \beta^*, \pi_{(\Sigma_C^{\beta^*}}(M_C)))$" and "$M_C := \texttt{upd}_{\Sigma_C^{\beta^*} \setminus G_r}(M_C, \texttt{rd}(\texttt{i}_{32} \to \texttt{i}_8))$" in (Call$_C$) (fig. 3) refer to two separate I/O paths. A path without any `rd` or `wr` instructions is called an *I/O-free path*.

3.1 Determinized product graph as a transition graph

A product program, represented as a *determinized product graph*, also called a comparison graph or a cross-product, $X = \ddot{A} \boxtimes C = (\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X)$, is a directed multigraph with finite sets of nodes $\mathcal{N}_X$ and edges $\mathcal{E}_X$, and a *deterministic choice map* $\mathcal{D}_X$. X is used to encode a lockstep execution of $\ddot{A}$ and C, such that $\mathcal{N}_X \subseteq \mathcal{N}_{\ddot{A}} \times \mathcal{N}_C$. The start node of X is $n_X^s = (n_{\ddot{A}}^s, n_C^s)$ and all nodes in $\mathcal{N}_X$ must be reachable from n_X^s. A node $n_X = (n_{\ddot{A}}, n_C)$ is an error node iff either $n_{\ddot{A}}$ or n_C is an error node. $\mathcal{N}_X^{\cancel{UW}}$ denotes the set of non-error nodes in X, such that $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\cancel{UW}} \Leftrightarrow (n_{\ddot{A}} \in \mathcal{N}_{\ddot{A}}^{\cancel{UW}} \wedge n_C \in \mathcal{N}_C^{\cancel{UW}})$.

Let $n_X = (n_{\ddot{A}}, n_C)$ and $n_X^t = (n_{\ddot{A}}^t, n_C^t)$ be nodes in $\mathcal{N}_X$; let $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow n_{\ddot{A}}^t$ be a finite path in $\ddot{A}$; and let $\xi_C = n_C \twoheadrightarrow n_C^t$ be a finite path in C. Each edge, $e_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi_C} n_X^t) \in \mathcal{E}_X$, is defined as a sequential execution of $\xi_{\ddot{A}}$ followed by ξ_C. The execution of e_X thus transfers control of X from n_X to n_X^t. The machine state of X is the concatenation of the machine states of $\ddot{A}$ and C. The outside world of X, written Ω_X, is a pair of the outside worlds of $\ddot{A}$ and C, i.e., $\Omega_X = (\Omega_{\ddot{A}}, \Omega_C)$. Similarly, the trace generated by X, written T_X, is a pair of the traces generated by $\ddot{A}$ and C, i.e., $T_X = (T_{\ddot{A}}, T_C)$.

During an execution of $e_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi_C} n_X^t) \in \mathcal{E}_X$, let $\vec{x}_{\ddot{A}}$ be variables in $\ddot{A}$ just at the end of the execution of path $\xi_{\ddot{A}}$ (at $n_{\ddot{A}}^t$) but before the execution of path ξ_C (recall, $\xi_{\ddot{A}}$ executes before ξ_C). $\mathcal{D}_X : ((\mathcal{E}_X \times \mathcal{E}_C \times \mathbb{N}) \to \texttt{ExprList})$, called a *deterministic choice map*, is a partial function that maps edge $e_X \in \mathcal{E}_X$, and the n^{th} (for $n \in \mathbb{N}$) occurrence of an edge '$e_C^\theta \in \xi_C$' labeled with instruction $\vec{v} := \theta(\vec{\tau})$ to a list of expressions $E(\vec{x}_{\ddot{A}})$. The semantics of $\mathcal{D}_X$ are such that, if $\mathcal{D}_X(e_X, e_C^\theta, n)$ is defined, then during an execution of e_X, an execution of the n^{th} occurrence of edge $e_C^\theta \in \xi_C$ labeled with $\vec{v} := \theta(\vec{\tau})$ is semantically equivalent to an execution of $\vec{v} := \mathcal{D}_X(e_X, e_C^\theta, n)$; otherwise, the original non-deterministic semantics of θ are used.

$\mathcal{D}_X$ determinizes (or refines) the non-deterministic choices in C. For example, in a product graph X that correlates the programs in fig. 1b and fig. 1c, let $e_X^2 \in \mathcal{E}_X$ correlate single instructions `I2` and `A4.2`. Let $e_C^{\texttt{I2},\theta_a}$ represent the edge labeled with $\alpha_b := \theta(\texttt{i}_{32})$ as a part of the translation of the `alloc` instruction at `I2`, as seen in (Alloc). Then, $\mathcal{D}_X(e_X^2, e_C^{\texttt{I2},\theta_a}, 1) = \texttt{esp}$ is identified by the first operand of the annotated $\texttt{alloc}_s$ instruction at `A4.2`. Similarly, if another edge $e_C^{\texttt{I2},\theta_m}$ (in the translation of `alloc` at `I2`) is labeled with $\theta(\texttt{i}_{32} \to \texttt{i}_8)$ (due to $M_C := \texttt{upd}_{[\alpha_b, \alpha_e]}(M_C, \theta(\texttt{i}_{32} \to \texttt{i}_8))$), then $\mathcal{D}_X(e_X^2, e_C^{\texttt{I2},\theta_m}, 1) = M_{\ddot{A}}$, i.e., the initial contents of the newly-allocated region in C are based on the contents of the correlated uninitialized stack region in $\ddot{A}$. Similarly, let $e_X^1 \in \mathcal{E}_X$ correlate single instructions `I1` and `A4.1` so that $\mathcal{D}_X(e_X^1, e_C^{\texttt{I1},\theta_a}, 1) = \texttt{v}_{\texttt{I1}}$ and $\mathcal{D}_X(e_X^1, e_C^{\texttt{I1},\theta_m}, 1) = M_{\ddot{A}}$.

For a path ξ_C in C, $[\xi_C]_{\mathcal{D}_X}^{e_X}$ denotes a *determinized path* that is identical to ξ_C except that: if $\mathcal{D}_X(e_X, e_C^\theta, n)$ is defined, then the n^{th} occurrence of edge $e_C^\theta \in \xi_C$, labeled with $\vec{v} := \theta(\vec{\tau})$, is replaced with a new edge $e_C^{\theta'_n}$ labeled with $\vec{v} := \mathcal{D}_X(e_X, e_C^\theta, n)$.

Execution of a product graph X must begin at node n_X^s in an initial machine state where $\Omega_{\ddot{A}} = \Omega_C$ and $T_{\ddot{A}} =_{st} T_C$ hold. Thus, X is a transition graph with its execution semantics derived from the semantics of $\ddot{A}$ and C, and the map $\mathcal{D}_X$.

3.2 Analysis of the determinized product graph

Let $X = \ddot{A} \boxtimes C = (\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X)$ be a determinized product graph. At each non-error node $n_X \in \mathcal{N}_X^{\overline{\mathcal{UW}}}$, we infer a node invariant, ϕ_{n_X}, which is a first-order logic predicate over state elements of X at node n_X that holds for all possible executions of X. A node invariant ϕ_{n_X} relates the values of state elements of C and $\ddot{A}$ that can be observed at n_X.

Definition 3.1 (Hoare Triple). Let $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\overline{\mathcal{UW}}}$. Let $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow n^t_{\ddot{A}}$ and $\xi_C = n_C \twoheadrightarrow n^t_C$ be paths in $\ddot{A}$ and C. A *Hoare triple*, written $\{pre\}(\xi_{\ddot{A}}; \xi_C)\{post\}$, denotes the statement: if execution starts at node n_X in state σ such that predicate $pre(\sigma)$ holds, and if paths $\xi_{\ddot{A}}; \xi_C$ are executed in sequence to completion finishing in state σ', then predicate $post(\sigma')$ holds.

Definition 3.2 (Path cover). At a node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X$, for a path $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow n^t_{\ddot{A}}$, let $\forall_{1\le j\le m} : e^j_X = n_X \xrightarrow{\xi_{\ddot{A}}; \xi^j_C} n^{t_j}_X$ be all edges in $\mathcal{E}_X$, such that $n^{t_j}_X = (n^t_{\ddot{A}}, n^{t_j}_C)$. The set of edges $\{e^1_X, e^2_X, \ldots, e^m_X\}$ *covers path* $\xi_{\ddot{A}}$, written $\{e^1_X, e^2_X, \ldots, e^m_X\}\langle \mathcal{D}_X, \xi_{\ddot{A}}\rangle$, iff $\{\phi_{n_X}\}(\xi_{\ddot{A}}; \epsilon)\{\bigvee_{j=1}^{m} pathcond([\xi^j_C]^{e^j_X}_{\mathcal{D}_X})\}$ holds.

Definition 3.3 (Path infeasibility). At a node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X$, a path $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow n^t_{\ddot{A}}$ is *infeasible* at n_X iff $\{\phi_{n_X}\}(\xi_{\ddot{A}}; \epsilon)\{\texttt{false}\}$ holds.

Definition 3.4 (Mutually exclusive paths). Two paths, $\xi^1_P = n_P \twoheadrightarrow n^{t_1}_P$ and $\xi^2_P = n_P \twoheadrightarrow n^{t_2}_P$, both originating at node n_P are *mutually-exclusive*, written $\xi^1_P \asymp \xi^2_P$, iff neither is a prefix of the other.

Definition 3.5. A *pathset* $\langle\xi\rangle_P$ is a set of pairwise mutually-exclusive paths $\langle\xi\rangle_P = \{\xi^1_P, \xi^2_P, \ldots, \xi^m_P\}$ originating at the same node n_P, i.e., $\forall_{1\le j\le m} : \xi^j_P = n_P \twoheadrightarrow n^j_P$ and $\forall_{1\le j_1<j_2\le m} : (\xi^{j_1}_P \asymp \xi^{j_2}_P)$.

3.2.1 X requirements. The following requirements on X help witness $C \sqsupseteq \ddot{A}$:

1. (Mutex$\ddot{A}$): For each node n_X with *all* outgoing edges $\{e^1_X, e^2_X, \ldots, e^m_X\}$ such that $e^j_X = (n_X \xrightarrow{\xi^j_{\ddot{A}}; \xi^j_C} n^j_X)$ (for $1 \le j \le m$), the following holds: $\forall_{1\le j_1,j_2\le m} : ((\xi^{j_1}_{\ddot{A}} = \xi^{j_2}_{\ddot{A}}) \vee (\xi^{j_1}_{\ddot{A}} \asymp \xi^{j_2}_{\ddot{A}}))$.
2. (Mutex$C$): At each node n_X, for a path $\xi_{\ddot{A}}$, let $\{e^1_X, e^2_X, \ldots, e^m_X\}$ be a set of *all* outgoing edges such that $e^j_X = n_X \xrightarrow{\xi_{\ddot{A}}; \xi^j_C} n^t_X$ (for $1 \le j \le m$). Then, the set $\{\xi^1_C, \xi^2_C, \ldots, \xi^m_C\}$ must be a pathset.
3. (Termination) For each non-error node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\overline{\mathcal{UW}}}$, $n_{\ddot{A}}$ is a terminating node iff n_C is a terminating node.
4. (SingleIO): For each edge $e_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi_C} n^t_X) \in \mathcal{E}_X$, either both $\xi_{\ddot{A}}$ and ξ_C are I/O paths, or both $\xi_{\ddot{A}}$ and ξ_C are I/O-free.
5. (Well-formedness): If a node of the form $n_X = (\_, \mathcal{W}_C)$ exists in $\mathcal{N}_X$, then n_X must be $(\mathcal{W}_{\ddot{A}}, \mathcal{W}_C)$.
6. (Safety): If a node of the form $n_X = (\mathcal{U}_{\ddot{A}}, \_)$ exists in $\mathcal{N}_X$, then n_X must be $(\mathcal{U}_{\ddot{A}}, \mathcal{U}_C)$.
7. (Similar-speed): Let $(e^1_X, e^2_X, \ldots, e^m_X)$ be a cyclic path, so that $\forall_{1\le j\le m} : e^j_X = (n^{f,j}_X \xrightarrow{\xi^j_{\ddot{A}}; \xi^j_C} n^{t,j}_X) \in \mathcal{E}_X$; $n^{f,1}_X = n^{t,m}_X$; and $\bigwedge_{j=1}^{m-1} (n^{t,j}_X = n^{f,j+1}_X)$. For each cyclic path, $(\neg \bigwedge_{j=1}^{m} (\xi^j_{\ddot{A}} = \epsilon)) \wedge (\neg \bigwedge_{j=1}^{m} (\xi^j_C = \epsilon))$ holds.
8. (Coverage$\ddot{A}$): For each non-error node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\overline{\mathcal{UW}}}$ and for each possible outgoing path $\xi^o_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow n^o_{\ddot{A}}$, either $\xi^o_{\ddot{A}}$ is infeasible at n_X, or there exists $e_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi_C} n^t_X) \in \mathcal{E}_X$ such that either $\xi_{\ddot{A}}$ is a prefix of $\xi^o_{\ddot{A}}$ or $\xi^o_{\ddot{A}}$ is a prefix of $\xi_{\ddot{A}}$.
9. (Coverage$C$): At node n_X, for some $\xi_{\ddot{A}}$, let $\{e^1_X, e^2_X, \ldots, e^m_X\}$ be the set of *all* outgoing edges such that $e^j_X = n_X \xrightarrow{\xi_{\ddot{A}}; \xi^j_C} (n^t_{\ddot{A}}, n^{t_j}_C)$ (for $1 \le j \le m$). Then, $\{e^1_X, e^2_X, \ldots, e^m_X\}\langle \mathcal{D}_X, \xi_{\ddot{A}}\rangle$ holds.
10. (Inductive): For each non-error edge $e_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi_C} n^t_X) \in \mathcal{E}_X$, $\{\phi_{n_X}\}(\xi_{\ddot{A}}; [\xi_C]^{e_X}_{\mathcal{D}_X})\{\phi_{n^t_X}\}$ holds.

11. (Equivalence): For each non-error node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\cancel{\mathcal{UW}}}$, $\Omega_{\ddot{A}} = \Omega_C$ must belong to ϕ_{n_X}.
12. (Memory Access Correspondence) or (MAC): For each edge $e_X = (n_X \xrightarrow{\xi_{\ddot{A}};\xi_C} n_X^t) \in \mathcal{E}_X$, such that $n_X^t \neq (\_, \mathcal{U}_C)$, $\{\phi_{n_X} \wedge (\boxed{\Sigma_{\ddot{A}}^{\mathsf{rd}}} = \boxed{\Sigma_C^{\mathsf{rd}}} = \emptyset)\}(\xi_{\ddot{A}}; [\xi_C]_{\mathcal{D}_X}^{e_X})\{(\boxed{\Sigma_{\ddot{A}}^{\mathsf{rd}}} \setminus \boxed{\Sigma_C^{\mathsf{rd}}}) \subseteq \Sigma_{\ddot{A}}^{G \cup F} \cup [\mathsf{esp}, \boxed{\mathsf{stk}_e}]\}$ and $\{\phi_{n_X} \wedge (\boxed{\Sigma_{\ddot{A}}^{\mathsf{wr}}} = \boxed{\Sigma_C^{\mathsf{wr}}} = \emptyset)\}(\xi_{\ddot{A}}; [\xi_C]_{\mathcal{D}_X}^{e_X})\{(\boxed{\Sigma_{\ddot{A}}^{\mathsf{wr}}} \setminus \boxed{\Sigma_C^{\mathsf{wr}}}) \subseteq \Sigma_{\ddot{A}}^{G_w \cup F_w} \cup [\mathsf{esp}, \boxed{\mathsf{stk}_e}]\}$ hold.
13. (MemEq): For each non-error node $n_X \in \mathcal{N}_X^{\cancel{\mathcal{UW}}}$, $M_{\ddot{A}} =_{\Sigma_{\ddot{A}}^B \setminus (\Sigma_{\ddot{A}}^{Z_l}|^v)} M_C$ must belong to ϕ_{n_X}.

(MAC) effectively requires that for every access on path $\xi_{\ddot{A}}$ to an address α belonging to region $r \in \{hp, cl\}$, there exists an access to α of the same read/write type on path $[\xi_C]_{\mathcal{D}_X}^{e_X}$. This requirement allows us to soundly over-approximate the set of addresses belonging to hp and cl for a faster SMT encoding (theorem 3.7 and section 4.2.3). For (MAC) to be meaningful, $\boxed{\Sigma_{\ddot{A},C}^{\mathsf{rd}}}$ and $\boxed{\Sigma_{\ddot{A},C}^{\mathsf{wr}}}$ must not be included in X's state elements over which a node invariant ϕ_{n_X} is inferred.

The first seven are *structural requirements* (constraints on the graph structure of X) and the remaining six are *semantic requirements* (require discharge of proof obligations). The first eleven are *soundness requirements* (required for theorem 3.6), the first twelve are *fast-encoding requirements*, and all thirteen are *search-algorithm requirements* (required for search optimizations). Excluding (Coverage$\ddot{A}$) and (Coverage$C$), the remaining eleven are called *non-coverage requirements.*

Theorem 3.6. *If there exists $X = \ddot{A} \boxtimes C$ that satisfies the soundness requirements, then $C \sqsupseteq \ddot{A}$ holds.*

Proof sketch. (Coverage$\ddot{A}$) and (Coverage$C$) ensure the coverage of $\ddot{A}$'s and C's traces in X. For an error-free execution of X, (Equivalence) and (Similar-speed) ensure that the generated traces are stuttering equivalent; for executions terminating in an error, (SingleIO), (Well-formedness), and (Safety) ensure that $C \sqsupseteq \ddot{A}$ holds by definition. See our technical report [Rose and Bansal 2024b] for the coinductive proof. □

3.2.2 Safety-relaxed semantics. Construct A' from A by using new *safety-relaxed semantics* for the assembly procedure such that: (1) a $\varphi_l = \mathsf{ov}([p]_w, \Sigma_{\ddot{A}}^{\mathsf{free}} \cup ((\Sigma_{\ddot{A}}^{Z_l}|^v) \setminus \Sigma_{\ddot{A}}^{F \cup S}))$ check in (Load$_{\ddot{A}}$) in A is replaced with $\varphi_l' = \mathsf{ov}([p]_w, (\Sigma_{\ddot{A}}^{Z_l}|^v) \setminus (\Sigma_{\ddot{A}}^F \cup [\mathsf{esp}, \boxed{\mathsf{cs}_e}]))$ in A'; (2) a $\varphi_s = \mathsf{ov}([p]_w, \Sigma_{\ddot{A}}^{\{\mathsf{free}\} \cup G_r \cup F_r} \cup ((\Sigma_{\ddot{A}}^{Z_l}|^v) \setminus \Sigma_{\ddot{A}}^{F_w \cup S}))$ check in (Store$_{\ddot{A}}$) in A is replaced with $\varphi_s' = \mathsf{ov}([p]_w, (\Sigma_{\ddot{A}}^{Z_l}|^v) \setminus (\Sigma_{\ddot{A}}^{F_w} \cup [\mathsf{esp}, \boxed{\mathsf{cs}_e}]))$ in A'; and (3) a $\varphi_r = \neg(\boxed{M^{cs}} =_{\Sigma_A^{cs}} M_A)$ check in (Ret$_A$) in A is replaced with $\varphi_r' = \mathtt{false}$ in A'. Let $\ddot{A}'$ be obtained by annotating A' using instructions described in section 2.4.3. Let $\ddot{A}$ be the annotated version of A, such that the annotations made in $\ddot{A}$ and $\ddot{A}'$ are identical.

Theorem 3.7. *Given $X' = \ddot{A}' \boxtimes C$ that satisfies the fast-encoding requirements, it is possible to construct $X = \ddot{A} \boxtimes C$ that also satisfies the fast-encoding requirements.*

Proof sketch. Start by constructing $X = X'$. Because $\varphi_{l,s,r}' \Rightarrow \varphi_{l,s,r}$, $\ddot{A}$ may include more executions of a path of form $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow \mathcal{U}_{\ddot{A}}$. Add new edges to $\mathcal{E}_X$, where each new edge correlates $\xi_{\ddot{A}}$ with some $\xi_C = n_C \twoheadrightarrow \mathcal{U}_C$. Because X' satisfies (MAC), the addition of such new edges will ensure that X satisfies (Coverage$C$). See our technical report [Rose and Bansal 2024b] for the proof. □

Using theorem 3.7, hereafter, we will use only the safety-relaxed semantics of the assembly procedure. We will continue to refer to the assembly procedure with the safety-relaxed semantics as A, and the corresponding annotated procedure $\ddot{A}$.

4 AUTOMATIC CONSTRUCTION OF A CROSS-PRODUCT

We now describe Dynamo, an algorithm that takes as input, the transition graphs corresponding to procedures C and A, and an *unroll factor* μ, and returns as output, annotated $\ddot{A}$ and product graph $X = \ddot{A} \boxtimes C = (\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X)$, such that all thirteen search-algorithm requirements are met. It identifies an inductive invariant network ϕ_X that maps each non-error node $n_X \in \mathcal{N}_X^{\overline{\mathcal{U}\mathcal{W}}}$ to its node invariant ϕ_{n_X}. Given enough computational time, Dynamo is guaranteed to find the required $(\ddot{A}, X)$ if: (a) A is a translation of C through bisimilar transformations up to a maximum unrolling of μ; (b) for two or more allocations or procedure calls that reuse stack space in A, their relative order in C is preserved in A; (c) the desired annotation to $\ddot{A}$ is identifiable either through search heuristics or through compiler hints; and (d) our invariant inference procedure is able to identify the required invariant network ϕ_X that captures the compiler transformations across C and A. Dynamo constructs the solution incrementally, by relying on the property that for a non-coverage requirement to hold for fully-annotated $\ddot{A}$ and fully-constructed X, it must also hold for partially-annotated $\ddot{A}$ and a partially-constructed subgraph of X rooted at its entry node n_X^s.

Dynamo is presented in algorithm 1. It assumes the availability of a **chooseFrom** operator, such that $\rho \hookleftarrow$ **chooseFrom** $\vec{\rho}$ chooses a quantity ρ from a finite set $\vec{\rho}$, such that Dynamo is able to complete the refinement proof, if such a choice exists. If the search space is limited, an exhaustive search could be used to implement **chooseFrom**. Otherwise, a counterexample-guided best-first search procedure (described later) is employed to approximate **chooseFrom**.

$\texttt{io}(n_P)$ evaluates to `true` iff n_P is either a source or sink node of an I/O path. $\texttt{term}(n_P)$ evaluates to `true` iff n_P is a terminating node. Dynamo first identifies an ordered set of nodes $Q_P \subseteq \mathcal{N}_P$, called the *cut points* in procedure P (*getCutPointsInRPO*), such that $Q_P \supseteq \{n_P : n_P \in \mathcal{N}_P \land (n_P = n_P^s \lor \texttt{io}(n_P) \lor \texttt{term}(n_P))\}$ and the maximum length of a path between two nodes in Q_P (not containing any other intermediate node that belongs to Q_P) is finite.

The algorithm to identify Q_P first initializes $Q_P \coloneqq \{n_P : n_P \in \mathcal{N}_P \land (n_P = n_P^s \lor \texttt{io}(n_P) \lor \texttt{term}(n_P))\}$, and then identifies all cycles in the transition graph that do not already contain a cut point; for each such cycle, the first node belonging to that cycle in reverse postorder is added to Q_P. In fig. 1c, $Q_{\ddot{A}}$ includes constituent nodes of assembly instructions at `A1`, `A9`, `A14`, and `exit`, where `exit` is the destination node of the error-free `halt` instruction due to the procedure return at `A17`.

A *simple path* $q_P \twoheadrightarrow q_P^t$ is a path connecting two cut points $q_P, q_P^t \in Q_P$, and not containing any other cut point as an intermediate node; q_P^t is called a *cut-point successor* of q_P. By definition, a simple path must be finite. The *cutPointSuccessors()* function takes a cut point q_P and returns all its cut-point successors in reverse postorder. In our example, the cut-point successors of a node at instruction `A9` are (constituent nodes of) `A9`, `A14`, $\mathcal{U}_{\ddot{A}}$, and $\mathcal{W}_{\ddot{A}}$. *getAllSimplePathsBetweenCutPoints(q_P, q_P^t, P)* returns *all* simple paths of the form $q_P \twoheadrightarrow q_P^t$, for $q_P, q_P^t \in Q_P$. Given a simple path $\xi_{\ddot{A}}$, *pathIsInfeasible*$(\xi_{\ddot{A}}, q_{\ddot{A}}, \mathcal{N}_X, \phi_X)$ returns `true` iff $\xi_{\ddot{A}}$ is infeasible at every node $n_X = (q_{\ddot{A}}, \_) \in \mathcal{N}_X$; our algorithm ensures there can be at most one $n_X = (q_{\ddot{A}}, \_) \in \mathcal{N}_X$ for each $q_{\ddot{A}} \in Q_{\ddot{A}}$.

correlatedPathsInCOptions(). *correlatedPathsInCOptions($\xi_{\ddot{A}}, \dots$)* identifies options for candidate pathsets $[\langle\xi\rangle_C]$, that can potentially be correlated with $\xi_{\ddot{A}} = q_{\ddot{A}} \twoheadrightarrow q_{\ddot{A}}^t$, and the **chooseFrom** operator chooses a pathset $\langle\xi\rangle_C$ from it. A path $\xi_C \in \langle\xi\rangle_C$ need not be a simple path, and can visit any node $n_C \in \mathcal{N}_C$ up to μ times. All paths in $\langle\xi\rangle_C$ must originate at a unique cut-point q_C such that $(q_{\ddot{A}}, q_C) \in \mathcal{N}_X$. By construction, there will be exactly one such $(q_{\ddot{A}}, q_C)$ in $\mathcal{N}_X$. Paths in $\langle\xi\rangle_C$ may have different end points however. For example, $\langle\xi\rangle_C = \{\epsilon\}$ and $\langle\xi\rangle_C = \{\texttt{I3}\longrightarrow\texttt{I4}\longrightarrow\texttt{I7}, \texttt{I3}\longrightarrow\mathcal{U}_C, \texttt{I3}\longrightarrow\texttt{I4}\longrightarrow\mathcal{U}_C\}$ may be potential candidates for $\xi_{\ddot{A}} = \texttt{A9}\longrightarrow\texttt{A10}\longrightarrow\texttt{A11}\longrightarrow\texttt{A9}$ in fig. 1.

If $q_{\ddot{A}}^t \notin \{\mathcal{U}_{\ddot{A}}, \mathcal{W}_{\ddot{A}}\}$, *correlatedPathsInCOptions()* returns candidates, where a candidate pathset $\langle\xi\rangle_C$ is a maximal set such that each path $\xi_C \in \langle\xi\rangle_C$ either (a) ends at a unique non-error destination cut-point node, say q_C^t (i.e., all paths $\xi_C \in \langle\xi\rangle_C$ ending at a non-error node end at q_C^t), or (b) ends at error node $\mathcal{U}_C$. This path enumeration strategy is the same as the one used in Counter [Gupta et al. 2020]; this strategy supports path specializing compiler transformations like loop peeling, unrolling, splitting, unswitching, etc., but does not support a path de-specializing transformation

```
Function Dynamo(A, C, μ)
    $\ddot{A} \hookleftarrow A$;   $\mathcal{N}_X \hookleftarrow \{(n^s_{\ddot{A}}, n^s_C)\}$;   $\mathcal{E}_X \hookleftarrow \{\}$;   $\mathcal{D}_X \hookleftarrow \emptyset$;
    $\phi_X \hookleftarrow \{(n^s_{\ddot{A}}, n^s_C) \mapsto (\Omega_{\ddot{A}} = \Omega_C)\}$;   $Q_{\ddot{A}} \hookleftarrow$ getCutPointsInRPO($\ddot{A}$);
    foreach $q_{\ddot{A}}$ in $Q_{\ddot{A}}$ do
        foreach $q^t_{\ddot{A}}$ in cutPointSuccessors($q_{\ddot{A}}, Q_{\ddot{A}}, \ddot{A}$) do
            foreach $\xi_{\ddot{A}}$ in getAllSimplePathsBetweenCutPoints($q_{\ddot{A}}, q^t_{\ddot{A}}, \ddot{A}$) do
                if pathIsInfeasible($\xi_{\ddot{A}}, q_{\ddot{A}}, \mathcal{N}_X, \phi_X$) then
                    continue
                end
                foreach $\xi_C$ in chooseFrom correlatedPathsInCOptions($\xi_{\ddot{A}}, \mu, \mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X, \ddot{A}, C$) do
                    $(\ddot{A}, \vec{\xi}'_{\ddot{A}}) \hookleftarrow$ chooseFrom asmAnnotOptions($\xi_{\ddot{A}}, \xi_C, \mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X, \ddot{A}, C$);
                    $\vec{\xi}'_C \hookleftarrow$ breakIntoSingleIOPaths($\xi_C$);
                    if ¬haveSimilarStructure($\vec{\xi}'_{\ddot{A}}, \vec{\xi}'_C$) then
                        return Failure
                    end
                    foreach $(\xi'_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow n^t_{\ddot{A}})), (\xi'_C = (n_C \twoheadrightarrow n^t_C))$ in zip($\vec{\xi}'_{\ddot{A}}, \vec{\xi}'_C$) do
                        $\mathcal{N}_X \hookleftarrow \mathcal{N}_X \cup \{(n^t_{\ddot{A}}, n^t_C)\}$;   //unlike $\mathcal{E}_X$, $\mathcal{N}_X$ may not grow on each iteration
                        $e_X \hookleftarrow (\xi'_{\ddot{A}}; \xi'_C)$;   $\mathcal{E}_X \hookleftarrow \mathcal{E}_X \cup \{e_X\}$;   $\mathcal{D}_X \hookleftarrow$ addDetMappings ($e_X$);
                        $\phi_X \hookleftarrow$ inferInvariantsAndCounterexamples($\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X, \ddot{A}, C$);
                        if ¬checkSemanticRequirementsExceptCoverage($\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X, \ddot{A}, C$) then
                            return Failure
                        end
                    end
                end
            end
        end
    end
    if ¬checkCoverageRequirements($\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X, \ddot{A}, C$) then
        return Failure
    end
    return Success($\ddot{A}, \mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X, \phi_X$)
end
```

Algorithm 1: Automatic construction of X.

like loop re-rolling. **If** $q^t_{\ddot{A}} = \mathscr{U}_{\ddot{A}}$, *correlatedPathsInCOptions()* returns candidates, where a candidate pathset $\langle\xi\rangle_C$ is a maximal set such that each path $\xi_C \in \langle\xi\rangle_C$ ends at $\mathscr{U}_C$. The algorithm identifies a correlation for a path $\xi_{\ddot{A}} = q_{\ddot{A}} \twoheadrightarrow \mathscr{W}_{\ddot{A}}$ only after correlations for all other paths of the form $\xi^{\mathscr{W}}_{\ddot{A}} = q_{\ddot{A}} \twoheadrightarrow q^{\mathscr{W}}_{\ddot{A}}$ (for $q^{\mathscr{W}}_{\ddot{A}} \neq \mathscr{W}_{\ddot{A}}$) have been identified: a pathset candidate $\langle\xi\rangle_C$ that has already been correlated with some other path $\xi^{\mathscr{W}}_{\ddot{A}}$ is then prioritized for correlation with $\xi_{\ddot{A}}$.

For example, in fig. 1c, for a cyclic path $\xi_{\ddot{A}}$ from a node at A9 to itself, one of the candidate pathsets, $\langle\xi\rangle_C$, returned by this procedure (at $\mu = 1$) contains eleven paths originating at I4 in fig. 1b: one that cycles back to I4 and ten that terminate at $\mathscr{U}_C$ (for each of the ten memory accesses in the path). For example, to evaluate the expression `v[*i]`, two memory loads are required, one at address `i` and another at `&v[*i]`, and each such load may potentially transition to $\mathscr{U}_C$ due to the $\texttt{accessIsSafeC}_{\tau,a}$ check evaluating to `false` in (Load$_C$). A path that terminates at $\mathscr{U}_C$ represents correlated transitions from node `(A9,I4)` in X such that $\ddot{A}$ remains error-free (to end at A9) but C triggers $\mathscr{U}$, e.g., if the memory access $\texttt{mem}_4\texttt{[esi+4*eax]}$ in $\ddot{A}$ (corresponding to `v[*i]` in C) overshoots the stack space corresponding to variable `v` but still lies within the stack region *stk*.

asmAnnotOptions(). For each simple path $\xi_{\ddot{A}}$, and each (potentially non-simple) path ξ_C in $\langle\xi\rangle_C$ [3], *asmAnnotOptions()* enumerates the options for annotating $\xi_{\ddot{A}}$ with $\texttt{alloc}_{s,v}$, $\texttt{dealloc}_{s,v}$ instructions and operands for `call` instructions, and the **chooseFrom** operator chooses one.

An annotation option includes the positions and the operands of the (de)allocation instructions (allocation site, alignment, address, and size). For a procedure-call, an annotation option also includes the arguments' types and values, and the set of callee-observable regions. The annotations for the callee name/address and the (de)allocations of procedure-call arguments in $\xi_{\ddot{A}}$ are uniquely identified using the number and type of arguments in the candidate correlated path ξ_C using the calling conventions. Similarly, the annotation of callee-observable regions follows from the regions observable by the correlated procedure call in ξ_C.

These annotations thus update A to incrementally construct $\ddot{A}$. If untrusted compiler hints are available, they are used to precisely identify these annotations. In a blackbox setting, where no compiler hints are available, we reduce the search space for annotations (at the cost of reduced generality) using the following three restrictions: (1) An $\texttt{alloc}_{s,v}$ ($\texttt{dealloc}_{s,v}$) annotation is annotated in $\xi_{\ddot{A}}$ only if an `alloc` (`dealloc`) instruction is present in ξ_C; (2) an $\texttt{alloc}_{s,v}$ ($\texttt{dealloc}_{s,v}$) annotation is added only after (before) an instruction that updates `esp`; moreover, for $\texttt{alloc}_s$, `esp` is used as the local variable's address expression; (3) for a single allocation site in C, at most one $\texttt{alloc}_{s,v}$ instruction (but potentially multiple $\texttt{dealloc}_{s,v}$ instructions) is added to $\ddot{A}$. Thus, in a blackbox setting, due to the third restriction, a refinement proof may fail if the compiler specializes a path containing a local variable allocation. Due to the second restriction, a refinement proof may fail for certain (arguably rare) types of order-preserving stack reallocation and stack merging performed by the compiler. Note that these limitations hold only for the blackbox setting.

After annotations, $\xi_{\ddot{A}}$ may become a non-simple path due to the extra I/O instructions introduced by the annotations. *asmAnnotOptions* therefore additionally returns $\overrightarrow{\xi'}_{\ddot{A}}$, which is a sequence of the simple paths constituting $\xi_{\ddot{A}}$. The (potentially non-simple) path ξ_C is then broken into a sequence of constituent paths $\overrightarrow{\xi'}_C$ (*breakIntoSingleIOPaths*) so that each I/O path appears by itself (and not as a sub-path of a longer constituent path) in $\overrightarrow{\xi'}_C$ — this caters to the (SingleIO) requirement. A failure is returned if the sequences $\overrightarrow{\xi'}_{\ddot{A}}$ and $\overrightarrow{\xi'}_C$ do not have similar structures (*haveSimilarStructure*). Let $\texttt{pos}(\xi, \overrightarrow{\xi})$ represent the position of path ξ in a sequence of paths $\overrightarrow{\xi}$. *haveSimilarStructure(*$\overrightarrow{\xi'}_{\ddot{A}}$, $\overrightarrow{\xi'}_C$*)* returns true iff $\overrightarrow{\xi'}_{\ddot{A}}$ and $\overrightarrow{\xi'}_C$ are of the same size, and for paths $\xi'_C \in \overrightarrow{\xi'}_C$ and $\xi'_{\ddot{A}} \in \overrightarrow{\xi'}_{\ddot{A}}$, if $\texttt{pos}(\xi'_C, \overrightarrow{\xi'}_C) = \texttt{pos}(\xi'_{\ddot{A}}, \overrightarrow{\xi'}_{\ddot{A}})$, then either both ξ'_C and $\xi'_{\ddot{A}}$ are I/O paths of same structure (i.e., they are either both reads or both writes for the same type of value) or both are I/O free.

Incremental construction of $(\ddot{A}, X)$**.** For each simple path $\xi'_{\ddot{A}}$ in $\overrightarrow{\xi'}_{\ddot{A}}$ enumerated in execution order, Dynamo correlates it with ξ'_C, such that $\texttt{pos}(\xi'_C, \overrightarrow{\xi'_C}) = \texttt{pos}(\xi'_{\ddot{A}}, \overrightarrow{\xi'}_{\ddot{A}})$ (through `zip` in algorithm 1). This candidate correlation $(\xi'_{\ddot{A}}; \xi'_C)$ is added as an edge e_X to $\mathcal{E}_X$, adding the destination node to $\mathcal{N}_X$ if not already present.

If ξ'_C represents a path between `wr(allocBegin(...))` and `wr(allocEnd(...))` for an `alloc` instruction in C, and $\xi'_{\ddot{A}}$ is a corresponding path due to an $\texttt{alloc}_{s,v}$ instruction, and edges $e_C^{\theta_a}$ and $e_C^{\theta_m}$ in ξ'_C are labeled with instructions $\alpha_b \coloneqq \theta(\texttt{i}_{32})$ and $\theta(\texttt{i}_{32} \rightarrow \texttt{i}_8)$ respectively due to (Alloc), we add mappings $\mathcal{D}_X(e_X, e^{\theta_a}, 1) = v$ and $\mathcal{D}_X(e_X, e^{\theta_m}, 1) = M_{\ddot{A}}$, where v is the address defined in $\xi'_{\ddot{A}}$ due to either (AllocS) or (AllocV) (*addDetMappings* (e_X)). Notice that our algorithm only populates $\mathcal{D}_X(e_X, e_C^{\theta}, n)$ for $n = 1$, even though section 3.1 defines $\mathcal{D}_X$ more generally.

[3]The number of paths can be exponential in procedure size, and so our implementation represents a pathset using a series-parallel digraph [Gupta et al. 2020] and annotates a pathset in $\ddot{A}$ in a single step. Similarly, a pathset in $\ddot{A}$ is correlated with a pathset in C in a single step. For easier exposition, the presented algorithm correlates each path individually.

$$\boxed{\texttt{affine}}\ \textstyle\sum_i c_i v_i = c \qquad \boxed{\texttt{ineqC}}\ \pm v \leq_s 2^c \qquad \boxed{\texttt{ineq}}\ v_1 \odot v_2 \qquad \boxed{\texttt{spOrd}}\ \texttt{sp}.p^{j_1}_{\ddot{A}} \leq_u (\texttt{sp}.p^{j_2}_{\ddot{A}} - v)$$

$$\boxed{\texttt{AllocEq}}\ \forall_{r\in B}\Sigma^r_C = \Sigma^r_{\ddot{A}} \qquad \boxed{\texttt{MemEq}}\ M_C =_{\Sigma^B_{\ddot{A}}\backslash(\Sigma^{Z_l}_{\ddot{A}}|^v)} M_{\ddot{A}} \qquad \boxed{\texttt{zEmpty}}\ \{\Sigma^z_C, \Sigma^z_{\ddot{A}}|^s, \Sigma^z_{\ddot{A}}|^v\}\ \{=,\neq\}\ \emptyset$$

$$\boxed{\texttt{spzBd}}\ \texttt{em}.z \vee (\texttt{sp}.p^j_{\ddot{A}} \odot \{\texttt{lb}.z, \texttt{ub}.z\}) \qquad \boxed{\texttt{spzBd'}}\ \texttt{em}.z \vee (\texttt{sp}.p^j_{\ddot{A}} \leq_u (\texttt{lb}.z - \texttt{lstSz}.z))$$

$$\boxed{\boxed{\texttt{gfySz}}}\ \forall_{r\in G\cup F\cup Y\backslash\{\texttt{vrdc}\}}(\texttt{sz}.r = \texttt{sz}(\texttt{T}(r))) \quad \boxed{\boxed{\texttt{vrdcSz}}}\ (\texttt{em.vrdc} \Leftrightarrow \texttt{sz.vrdc} = 0) \quad \boxed{\boxed{\texttt{Empty}}}\ \forall_{r\in G\cup F\cup Y\cup Z}(\Sigma^r_C = \emptyset \Leftrightarrow \texttt{em}.r)$$

$$\boxed{\boxed{\texttt{gfyIntvl}}}\ \forall_{r\in G\cup F\cup Y}((\texttt{sz}.r = 0) \vee ((\texttt{lb}.r \leq_u \texttt{ub}.r) \wedge (\texttt{ub}.r = \texttt{lb}.r + \texttt{sz}.r - 1_{\texttt{i}_{32}}) \wedge ([\texttt{lb}.r, \texttt{ub}.r] = \Sigma^r_{\ddot{A}})))$$

$$\boxed{\boxed{\texttt{zlIntvl}}}\ \texttt{em}.zl \vee ((\texttt{lb}.zl \leq_u \texttt{ub}.zl) \wedge (\texttt{lb}.zl + \texttt{lstSz}.zl - 1_{\texttt{i}_{32}} = \texttt{ub}.zl) \wedge ([\texttt{lb}.zl, \texttt{lb}.zl] = \Sigma^{zl}_C)))$$

$$\boxed{\boxed{\texttt{zaBd}}}\ \texttt{em}.za \vee ((\texttt{lb}.za \leq_u \texttt{ub}.za) \wedge (\texttt{lb}.za + \texttt{lstSz}.za - 1_{\texttt{i}_{32}} \leq_u \texttt{ub}.za) \wedge (\texttt{lb}.za = \texttt{lb}(\Sigma^{za}_C) \wedge \texttt{ub}.za = \texttt{ub}(\Sigma^{za}_C)))$$

$$\boxed{\boxed{\texttt{StkBd}}}\ \Sigma^{\{stk\}\cup Y}_{\ddot{A}} \cup (\Sigma^Z_{\ddot{A}} \setminus (\Sigma^{Z_l}_{\ddot{A}}|^v)) = [\texttt{esp}, \texttt{stk}_e] \qquad \boxed{\boxed{\texttt{csBd}}}\ \Sigma^{\{cs,cl\}}_{\ddot{A}} = [\texttt{stk}_e + 1, \texttt{cs}_e]$$

$$\boxed{\boxed{\texttt{NoOverlap}C}}\ \neg\texttt{ov}(\Sigma^{hp}_{\ddot{A}}, \Sigma^{cl}_{\ddot{A}}, \Sigma^{\texttt{vrdc}}_{\ddot{A}}, \ldots, i^g_{\ddot{A}}, \ldots, i^y_{\ddot{A}}, \ldots, \Sigma^z_{\ddot{A}}) \qquad \boxed{\boxed{\texttt{ROM}C}}\ \forall_{r\in G_r} M_C =_{i^r_C} \texttt{ROM}^r_C(i^r_C)$$

$$\boxed{\boxed{\texttt{NoOverlap}A}}\ \neg\texttt{ov}(\Sigma^{\{hp,cl\}\cup G\cup Y}_{\ddot{A}}, \ldots, \Sigma^z_{\ddot{A}}|^s, \ldots, \Sigma^{stk}_{\ddot{A}}, \Sigma^{cs}_{\ddot{A}}, \Sigma^F_{\ddot{A}}) \qquad \boxed{\boxed{\texttt{ROM}A}}\ \forall_{r\in F_r} M_{\ddot{A}} =_{i^r_{\ddot{A}}} \texttt{ROM}^r_{\ddot{A}}(i^r_{\ddot{A}})$$

Fig. 7. Predicate grammar for constructing candidate invariants. v represents a bitvector variable (registers, stack slots, and ghost variables), c represents a bitvector constant. $\odot \in \{\leq_{s,u}, <_{s,u}, >_{s,u}, \geq_{s,u}\}$.

If the destination node is not an error node, then the *inferInvariantsAndCounterexamples()* procedure updates the invariant network ϕ_X due to the addition of this new edge. The non-coverage requirements are checked after invariant inference (*checkSemanticRequirementsExceptCoverage*) and a candidate is discarded if the check fails.

When all simple paths between the cut points of $\ddot{A}$ are exhausted, the (Coverage$\ddot{A}$) requirement must be satisfied by construction. *checkCoverageRequirements()* further checks the satisfaction of (Coverage$C$) before returning Success. Dynamo is sound because it returns Success only if all the thirteen search-algorithm requirements are satisfied.

The **chooseFrom** operator must attempt to maximize the chances of returning Success, even if only a fraction of the search space has been explored. Dynamo uses the counterexamples generated when a proof obligation is falsified (e.g., during invariant inference) to guide the search towards the more promising options. A counterexample is a proxy for the machine states of C and $\ddot{A}$ that may appear at a node n_X during the lockstep execution encoded by X. Thus, if at any step during the construction of X, the execution of a counterexample for a candidate partial solution $(\ddot{A}, X)$ results in the violation of a non-coverage requirement, that candidate is discarded. Further, counterexample execution opportunistically weakens the node invariants in X. Like Counter, we use the number of live registers in $\ddot{A}$ related through the current invariants in ϕ_X to rank the enumerated partial candidate solutions to implement a best-first search.

4.1 Invariant Inference

We use a counterexample-guided inference algorithm to identify node invariants [Gupta et al. 2020]. Candidate invariants at a node n_X of a partial product-graph are formed by conjuncting predicates drawn from the grammar shown in fig. 7. Apart from affine ($\boxed{\texttt{affine}}$) and inequality relations ($\boxed{\texttt{ineq}}$ and $\boxed{\texttt{ineqC}}$) for relating values across C and $\ddot{A}$, the guesses attempt to equate the allocation and memory state of common regions across the two procedures ($\boxed{\texttt{AllocEq}}$ and $\boxed{\texttt{MemEq}}$).

Recall that we save stackpointer value at the boundary of a stackpointer updating instruction at PC $p^j_{\ddot{A}}$ in ghost variable $\texttt{sp}.p^j_{\ddot{A}}$ ((Op-esp) in fig. 4). To prove separation between different local variables, we require invariants that lower-bound the gap between two ghost variables, say $\texttt{sp}.p^{j_1}_{\ddot{A}}$ and $\texttt{sp}.p^{j_2}_{\ddot{A}}$, by some value v that depends on the allocation size operand of an $\texttt{alloc}_s$ instruction ($\boxed{\texttt{spOrd}}$). To capture the various relations between lower bounds, upper bounds, region sizes, and $\texttt{sp}.p^j_{\ddot{A}}$, the guessing grammar includes shapes $\boxed{\texttt{spzBd}}$ and $\boxed{\texttt{spzBd'}}$ that are of the form: "either a local variable region is empty or its bounds are related to $\texttt{sp}.p^j_{\ddot{A}}$ in these possible ways". $\boxed{\texttt{zEmpty}}$ tracks the emptiness of the address-set of a local region. Together, these predicate shapes (along with $\boxed{\texttt{affine}}$ and $\boxed{\texttt{ineq}}$ relations between $\boxed{\texttt{sp}.p^j_{\ddot{A}}}$) enable disambiguation between stack writes involving spilled pseudo-registers and stack-allocated locals.

The predicate shapes listed below the dividing line segment in fig. 7 encode the *global invariants* that hold by construction (due to our execution semantics) at every non-error product-graph node n_X. $\boxed{\texttt{gfySz}}$, $\boxed{\texttt{vrdcSz}}$, and $\boxed{\texttt{gfyIntvl}}$ together encode the fact that the ghost variables associated with a region $r \in G \cup F \cup Y$ track its bounds, size, and that the address set of r is an interval. $\boxed{\texttt{Empty}}$ encodes that the ghost variable $\boxed{\texttt{em}.r}$ for $r \in G \cup F \cup Y \cup Z$ tracks the emptiness of the region r. $\boxed{\texttt{zlIntvl}}$ captures the property that a local variable region zl, if non-empty, must be an interval of size $\boxed{\texttt{lstSz}.zl}$. $\boxed{\texttt{zaBd}}$ captures a weaker property for a local region za (allocated using `alloca()`): if non-empty, this region must be bounded by its ghost variables and the region must be at least $\boxed{\texttt{lstSz}.za}$ bytes large. $\boxed{\texttt{StkBd}}$ encodes the invariant that the interval $[\texttt{esp}, \boxed{\texttt{stk}_e}]$ represents the union of the address sets of stk, regions in Y, and stack-allocated local regions $(\Sigma^{Z}_{\ddot{A}} \setminus (\Sigma^{Z_l}_{\ddot{A}}|^{v}))$; $\boxed{\texttt{csBd}}$ is similarly shaped and encodes that the interval $[\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$ represents the union of the address sets of regions cs and cl. $\boxed{\texttt{NoOverlap}C}$ encodes the disjointedness of all regions $r \in B$. $\boxed{\texttt{NoOverlap}A}$ encodes the disjointedness of all regions in $\ddot{A}$ except virtually-allocated regions. Finally, $\boxed{\texttt{ROMC}}$ and $\boxed{\texttt{ROM}A}$ encode the preservation of memory contents of read-only regions in C and $\ddot{A}$.

A dataflow analysis [Andersen 1994] computes the possible states of $\beta()$ and $\beta_M()$ maps at each $n_C \in \mathcal{N}_C$, and the over-approximate solution is added to ϕ_{n_X} for each $n_X = (\_, n_C)$.

4.2 SMT Encoding

At a non-error node n_X, a proof obligation is represented as a first-order logic predicate over the state elements at n_X and discharged using an SMT solver. The machine states of C and $\ddot{A}$ are represented using bitvectors (for a register/variable), arrays (for memory), and uninterpreted functions (for $\texttt{read}_{\vec{\tau}}(\Omega_P)$ and $\texttt{io}(\Omega_P, \vec{v}, \texttt{rw})$). For address sets, we encode the set-membership predicate $\alpha \in \Sigma^r_P$ for an arbitrary address α, region identifier r, and procedure $P \in \{C, \ddot{A}\}$. All other address set operations can be expressed in terms of the set-membership predicate. To simplify the encodings, we rely on the correct-by-construction invariants in fig. 7 and assume that ϕ_{n_X} satisfies the (Equivalence), (MAC), and (MemEq) requirements. Notice that (Equivalence) implies $\boxed{\texttt{AllocEq}}$.

Recall that for $z \in Z_l$, at a node $n_X \in \mathcal{N}_X$, $\Sigma^z_{\ddot{A}}|^s$ and $\Sigma^z_{\ddot{A}}|^v$ represent the address sets corresponding to the stack and virtual allocations performed in $\ddot{A}$ for z. Let $Zls = \{z \mid z \in Z_l \wedge \Sigma^z_{\ddot{A}}|^s \neq \emptyset\}$ and $Zlv = \{z \mid z \in Z_l \wedge \Sigma^z_{\ddot{A}}|^v \neq \emptyset\}$ represent the set of stack-allocated locals and virtually-allocated at n_X respectively. Recall that we restrict ourselves to only those compiler transformations that ensure the validity of $Zls \cap Zlv = \emptyset$ at each n_X (section 2.4.3).

4.2.1 Representing address-sets using allocation state array. Let $\mathcal{L}_P : \text{i}_{32} \to R$ be an *allocation state array* that maps an address to a region identifier in procedure P. For $r \notin Zlv$, $\alpha \in \Sigma^r_P$ is encoded as $\texttt{sel}_1(\mathcal{L}_P, \alpha) = r$. Allocation of an address α to region r ($\Sigma^r_P := \Sigma^r_P \cup \{\alpha\}$) is encoded as $\mathcal{L}_P := \texttt{st}_1(\mathcal{L}_P, \alpha, r)$. Similarly, deallocation ($\Sigma^r_P := \Sigma^r_P \setminus \{\alpha\}$) is encoded as $\mathcal{L}_P := \texttt{st}_1(\mathcal{L}_P, \alpha, \texttt{free})$.

For $zlv \in Zlv$, both $\alpha \in \Sigma^{zlv}_C$ and $\alpha \in \Sigma^{zlv}_{\ddot{A}}$ are encoded as $\texttt{sel}_1(\mathcal{L}_C, \alpha) = zlv$, i.e., the set-membership encodings for both procedures use $\mathcal{L}_C$ for virtually-allocated locals (by relying on the $\boxed{\texttt{AllocEq}}$ invariant at n_X). In other words, $\mathcal{L}_{\ddot{A}}$ is not used to track the virtually-allocated locals; instead, an address belonging to a virtually allocated-region maps to one of $\{\texttt{free}, stk, cs\} \cup F$ regions in $\mathcal{L}_{\ddot{A}}$. Consequently, the (de)allocation instructions $\Sigma^{zlv}_{\ddot{A}}|^v := \Sigma^{zlv}_{\ddot{A}}|^v \cup [v]_w$ and $\Sigma^{zlv}_{\ddot{A}}|^v := \emptyset$ are vacuous in $\ddot{A}$, i.e., they do not change any state element in $\ddot{A}$ (fig. 6).

$\alpha \in \Sigma_P^r$	Full-array encoding		Partial-interval encoding ($\Sigma_P^{Za} \neq \emptyset$)	Full-interval encoding ($\Sigma_P^{Za} = \emptyset$)
	$P = C$	$P = A$		
$r = hp$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\alpha \notin (\Sigma_{\ddot{A}}^{G \cup F} \cup Zlv_U(\xi_{\ddot{A}}) \cup [SP_{min}(\xi_{\ddot{A}}), \boxed{\mathsf{cs}_e}])$	$\alpha \notin (\Sigma_{\ddot{A}}^{G \cup F} \cup Zlv_U(\xi_{\ddot{A}}) \cup [SP_{min}(\xi_{\ddot{A}}), \boxed{\mathsf{cs}_e}])$
$r = cl$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\alpha \in [\boxed{\mathsf{stk}_e} + 1, \boxed{\mathsf{cs}_e}] \land \alpha \notin Zlv_U(\xi_{\ddot{A}})$	$\alpha \in [\boxed{\mathsf{stk}_e} + 1, \boxed{\mathsf{cs}_e}] \land \alpha \notin Zlv_U(\xi_{\ddot{A}})$
$r \in G \cup Zlv$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$	$\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$
$r \in Y \cup Z_a \cup Zls$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$	$\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$
$r \in F$	$\mathtt{false}$	$\mathtt{sel}_1(\mathcal{L}_{\ddot{A}}, \alpha) = r$	$\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$	$\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$
$r = cs$	$\mathtt{false}$	$\mathtt{sel}_1(\mathcal{L}_{\ddot{A}}, \alpha) = r$	$\alpha \in [\boxed{\mathsf{stk}_e} + 1, \boxed{\mathsf{cs}_e}] \land \alpha \in Zlv_U(\xi_{\ddot{A}})$	$\alpha \in [\boxed{\mathsf{stk}_e} + 1, \boxed{\mathsf{cs}_e}] \land \alpha \in Zlv_U(\xi_{\ddot{A}})$
$r = stk$	$\mathtt{false}$	$\mathtt{sel}_1(\mathcal{L}_{\ddot{A}}, \alpha) = r$	$\mathtt{sel}_1(\mathcal{L}_{\ddot{A}}, \alpha) = r$	$\alpha \in [SP_{min}(\xi_{\ddot{A}}), \boxed{\mathsf{stk}_e}] \land \bigwedge_{r \in Y \cup Zls}(\alpha \notin \Sigma_{\ddot{A}}^r)$

Table 2. SMT encoding of $\alpha \in \Sigma_P^r$ for Dynamo's proof obligation O with outgoing assembly path $\xi_{\ddot{A}}$.

This encoding, based on allocation state arrays $\mathcal{L}_C$ and $\mathcal{L}_{\ddot{A}}$, is called the *full-array encoding*. The second and third columns of table 2 describe the full-array encoding for $P = C$ and $P = \ddot{A}$. In the table, we use $\boxed{\mathtt{AllocEq}}$ to replace $\mathtt{sel}_1(\mathcal{L}_{\ddot{A}}, \alpha)$ with $\mathtt{sel}_1(\mathcal{L}_C, \alpha)$ for $r \in B$. For example, in the full-array encoding, the (MemEq) requirement $M_C =_{\Sigma_{\ddot{A}}^B \setminus (\Sigma_{\ddot{A}}^{Z_l}|^v)} M_{\ddot{A}}$ becomes $\forall\alpha : ((\mathtt{sel}_1(\mathcal{L}_C, \alpha) \in G \cup \{hp, cl\} \cup Y \cup Zls \cup Z_a) \Rightarrow (\mathtt{sel}_1(M_C, \alpha) = \mathtt{sel}_1(M_{\ddot{A}}, \alpha)))$.

4.2.2 Interval encodings for $r \in G \cup F \cup Y \cup Z_l \cup \{stk\}$. We use $\boxed{\boxed{\mathtt{gfyIntvl}}}$, $\boxed{\boxed{\mathtt{zlIntvl}}}$, and $\boxed{\mathtt{AllocEq}}$ invariants for a more efficient *interval encoding*: for $r \in G \cup F \cup Y \cup Z_l$, we encode $\alpha \in \Sigma_P^r$ as $\neg\boxed{\mathsf{em}.r} \land (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathsf{ub}.r})$. Moreover, if there are no local variables allocated due to the `alloca()` operator (i.e., $\Sigma_P^{Z_a} = \emptyset$), then all local variables are contiguous, and so, due to $\boxed{\boxed{\mathtt{StkBd}}}$, the *stk* region can be identified as $[\mathsf{esp}, \boxed{\mathsf{stk}_e}] \setminus \Sigma_{\ddot{A}}^{Y \cup Zls}$ — the corresponding interval encoding is shown in the right-most cell of $r = stk$ row in table 2.

4.2.3 Interval encodings for $r \in \{hp, cl, cs\}$. Even though hp, cl, cs can be discontiguous regions in general, we over-approximate these regions to their contiguous covers to be able to soundly encode them using intervals. At a node $n_X = (n_{\ddot{A}}, n_C)$, Dynamo may generate a proof obligation O of the form $\{pre\}(\xi_{\ddot{A}}; [\xi_C]_{\mathcal{D}_X}^{ex})\{post\}$ — recall that path-cover and path-infeasibility conditions are also represented as Hoare triples with $\xi_C = \epsilon$. If $\xi_{\ddot{A}}$ is an I/O path, its execution interacts with the outside world, and so an over-approximation of an externally-visible address set is unsound. We thus restrict our attention to an I/O-free $\xi_{\ddot{A}}$ for interval encoding.

Let $n_{\ddot{A}}^1, n_{\ddot{A}}^2, \ldots, n_{\ddot{A}}^m$ be the nodes on path $\xi_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow n_{\ddot{A}}^t)$, such that $n_{\ddot{A}}^1 = n_{\ddot{A}}$ and $n_{\ddot{A}}^m = n_{\ddot{A}}^t$. Let $SP_{min}(\xi_{\ddot{A}})$ represent the the minimum value of esp observed at any node $n_{\ddot{A}}^j$ $(1 \leq j \leq m)$ visited during the execution of path $\xi_{\ddot{A}}$. Similarly, let $Zlv_U(\xi_{\ddot{A}})$ be the union of the values of set $\Sigma_{\ddot{A}}^{Zlv}$ observed at any $n_{\ddot{A}}^j$ $(1 \leq j \leq m)$ visited during $\xi_{\ddot{A}}$'s execution.

Let $HP(\xi_{\ddot{A}}) = \mathsf{comp}(\Sigma_{\ddot{A}}^{G \cup F} \cup Zlv_U(\xi_{\ddot{A}}) \cup [SP_{min}(\xi_{\ddot{A}}), \boxed{\mathsf{cs}_e}])$, $CL(\xi_{\ddot{A}}) = [\boxed{\mathsf{stk}_e} + 1_{\mathtt{i}_{32}}, \boxed{\mathsf{cs}_e}] \setminus Zlv_U(\xi_{\ddot{A}})$, and $CS(\xi_{\ddot{A}}) = [\boxed{\mathsf{stk}_e} + 1_{\mathtt{i}_{32}}, \boxed{\mathsf{cs}_e}] \cap Zlv_U(\xi_{\ddot{A}})$.

Theorem 4.1. *Let $O = \{pre\}(\xi_{\ddot{A}}; [\xi_C]_{\mathcal{D}_X}^{ex})\{post\}$ be a proof obligation generated by Dynamo. Let O' be obtained from O by strengthening precondition pre to $pre' = (pre \land ((\Sigma_{\ddot{A}}^{hp} = HP(\xi_{\ddot{A}})) \land (\Sigma_{\ddot{A}}^{cl} = CL(\xi_{\ddot{A}})) \land (\Sigma_{\ddot{A}}^{cs} = CS(\xi_{\ddot{A}})))$. If $\xi_{\ddot{A}}$ is I/O-free, $O \Leftrightarrow O'$ holds.*

Proof sketch. $O \Rightarrow O'$ is trivial. The proof for $O' \Rightarrow O$, available in [Rose and Bansal 2024b], relies on the limited shapes of predicates that may appear in *pre*, *post* — for I/O-free $\xi_{\ddot{A}}$, these shapes are limited by our invariant grammar (fig. 7), and the edge conditions appearing in our execution semantics (figs. 3 to 6). The proof holds only if the safety-relaxed semantics are used. □

Using theorem 4.1, we rewrite $\alpha \in \Sigma_P^{hp}$ to $\alpha \in HP(\xi_{\ddot{A}})$, $\alpha \in \Sigma_P^{cl}$ to $\alpha \in CL(\xi_{\ddot{A}})$, and $\alpha \in \Sigma_P^{cs}$ to $\alpha \in CS(\xi_{\ddot{A}})$ in proof obligation O. As shown in table 2, if $\Sigma_P^{Z_a} = \emptyset$ holds at n_X, we encode all non-free regions using intervals (called *full-interval encoding*); else, we encode regions in $Y \cup Z_a \cup Zls \cup \{stk\}$

Name	Programming pattern	
ats	Address-taken local scalar	`int ats() { int ret; foo(&ret); return ret; }`
atc	Address taken conditionally	`int atc(int* p) { int x; if (!p) p = &x; foo(p); return *p }`
ata	Local array	`int ata() { char ret[8]; foo(ret); return bar(ret, 0, 16); }`
vwl	Variadic procedure	`int vwl(int n, ...) { va_list a; va_start(a, n); for(...){ va_arg(a,int) ... } }`
as	GCC alloca()	`int as(int n){...int* p=alloca(n*sizeof(n)); for(...){/*write to p*/}...}`
vsl	VLA with loop	`int vsl(int n){... int v[n]; for(...){/*write to v*/}...}`
vcu	VLA conditional use	`int vcu(int n,int k){ int a[n]; if (...) { /*rd/wr to a*/}...}`
min	minprintf procedure from K&R [Kernighan and Ritchie 1988]	
ac	alloca() conditional use	`int ac(char*a) {..if (!a) a=alloca(n); for(...)/*r/w to a*/}`
all	alloca() in a loop to form a linked list	`all(){..hd=NULL; for(...){..n=alloca(..);..n->nxt=hd; hd=n;} while(...){/* traverse the list starting at hd */}}`
atail	Local array alloc. in loop	`int atail(..){..for(..){ char a[4096]; f(a..); b(a..);...}...}`
vil*N*	*N* VLA(s) in a loop	`int vilN(..){..for(i=1;i<n;++i){ int v1[4*i], ... vN[4*i]; fooN(...); ..}.. }`
vilcc	VLA in loop with continue	`int vilcc(..){..while(i<n){ char v[i];...if(..) continue;..}..}`
fib	Program from fig. 1	
vilce	VLA in loop with break	`int vilce(..){..while(i<n){ char v[i];...if(..) break;..}..}`
rod	A local char array initialized using string; a VLA; a for loop	Available in [Rose and Bansal 2024b].

Table 3. Benchmarks and their programming patterns. N in vil$N$ is substituted to obtain vil1, vil2, and vil3. Program listings available in [Rose and Bansal 2024b].

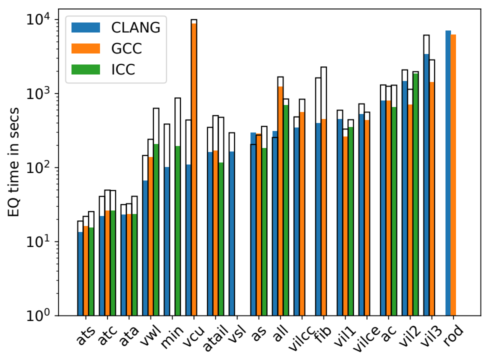
using an allocation state array, and $G \cup F \cup Zlv \cup \{hp, cl, cs\}$ using intervals (called *partial-interval encoding*).

5 EXPERIMENTS

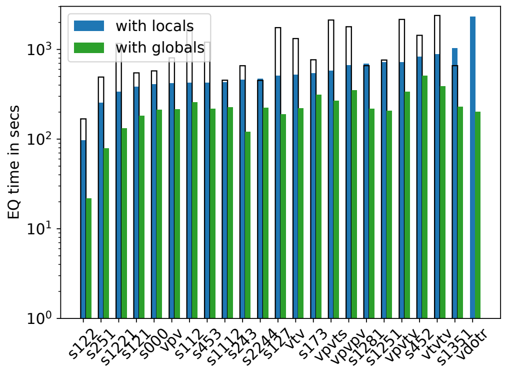
Dynamo uses four SMT solvers running in parallel for discharging proof obligations: z3-4.8.7, z3-4.8.14, Yices2-45e38fc, and cvc4-1.7. Unless otherwise specified, we use $\mu = 64$, a timeout of ten minutes for an SMT query, and a timeout of eight hours for a refinement check.

Before checking refinement, if the address of a local variable l is never taken in C, we transform C to register-allocate l (LLVM's mem2reg). This reduces the proof effort, at the cost of having to trust the pseudo-register allocation logic. mem2reg does not register-allocate local arrays and structs in LLVM$_d$, even though an optimizing compiler may register-allocate them in assembly — virtual allocations help validate such translations.

We first evaluate the efficacy of our implementation to handle the diverse programming patterns seen with local allocations (table 3). These include variadic procedures, VLAs allocated in loops, alloca() in loops, etc. Figure 8a shows the results of our experiments for these 18 programming patterns from table 3 and three compilers, namely Clang/LLVM v12.0.0, GCC v8.4.0, and ICC v2021.8.0, to generate 32-bit x86 executables at -O3 optimization with inter-procedural analyses disabled using the compilers' command-line flags. The X-axis lists the benchmarks and the Y-axis represents the total time taken in seconds (log scale) for a refinement check — to study the performance implications, we run a check with all three encodings for these benchmarks. The filled and empty bars represent the time taken with full-interval and partial-interval SMT encodings respectively. The figure does not show the results for the full-array encoding. A missing bar represents a failure to compute the proof. Of 54 procedure pairs, our implementation is able to check refinement for 45, 43, and 37 pairs while using full-interval, partial-interval, and full-array encodings respectively. For benchmarks where a refinement check succeeds for all encodings, the full-interval encoding performs 1.7-2.2x and 3.5-4.9x faster on average (for each compiler) than the partial-interval and full-array encodings respectively. The reasons for nine failures are: (a)



(a) Comparison of running times with full- (filled bars) and partial- (empty bars) interval encoding.



(b) Comparison of running times of benchmarks with exactly same code modulo allocation.

Fig. 8. Experiments with procedures in table 3 and TSVC. Y-axis is logarithmically scaled.

Name	SLOC	ALOC	$\#_{al}$	$\#_{loop}$	$\#_{fcall}$	D	eqT	Nodes	Edges	EXP	BT	$\#_q$	Avg. qT
generateMTFValues	76	144	1	6	1	2	4k	14	30	60	16	3860	0.56
recvDecodingTables	70	199	2	14	10	3	3k	38	66	102	15	5611	0.21
undoReversible-Transformation_fast	116	221	1	7	6	2	2k	21	34	43	6	2998	0.23

Table 4. Statistics obtained by running Dynamo on procedures in the bzip2 program.

limitation of the blackbox annotation algorithm for one procedure-pair; (b) incompleteness of invariant inference for six procedure-pairs (e.g., requirement of non-affine invariants, choice of program variables); and (c) SMT solver timeouts for two procedure-pairs. vilcc and vilce require multiple $\texttt{dealloc}_s$ instructions to be added to A for a single dealloc in C. An $\texttt{alloc}_v$ annotation is required for the 'va_list a' variable in the GCC and ICC compilations of vwl (see table 3) — while GCC and ICC register-allocate a, it is allocated in memory using alloc in $\texttt{LLVM}_d$ (even after mem2reg). The average number of best-first search backtrackings across all benchmarks is 2.8. The time spent in constructing the correct product graph forms around 70-80% of the total search time.

We next evaluate Dynamo on the TSVC suite of vectorization benchmarks with arrays and loops [Maleki et al. 2011], also used in previous work [Churchill et al. 2019; Gupta et al. 2020]. We use two versions of these benchmarks: (1) 'globals' where global variables are used for storing the output array values, and (2) 'locals' where local array variables are used for storing the output values and a procedure call is added at the end of the procedure body to print the contents of the local array variables. The compiler performs the same vectorizing transformations on both versions. Unlike globals, locals additionally requires the automatic identification of required annotations.

Figure 8b shows the execution times of Dynamo for validating the compilations produced by Clang/LLVM v12.0.0 (at -O3) for these two versions of the TSVC benchmarks. Dynamo can successfully validate these compilations. Compared to globals, refinement checks are 2.5x slower for locals (on average) due to the extra overhead of identifying the required annotations.

Our third experiment is on SPEC CPU2000's bzip2[Henning 2000] program compiled using Clang/LLVM v12.0.0 at three optimization levels: O1, O2, and O1-. O1- is a custom optimization level configured by us that enables all optimizations at O1 *except* (a) merging of multiple procedure calls on different paths into a single call, (b) early-CSE (common subexpression elimination), (c) loop-invariant code motion at both LLVM IR and Machine IR, (d) dead-argument elimination, (e) inter-procedural sparse conditional constant propagation, and (f) dead-code elimination of procedure calls. bzip2 runs 2% slower with O1- than with O1; this is still 5% faster than the executable produced by CompCert, for example. Of all 72 procedures in `bzip2`, Dynamo successfully validates the translations for 64, 60, and 54 procedures at `O1-`, `O1`, and `O2` respectively at $\mu = 1$. At `O1-`, Dynamo takes around four CPU hours to compute refinement proofs for the 64 procedures. Dynamo times out for the remaining eight procedures, all of which are bigger than 190 ALOC.

Three of `bzip2`'s procedures for which refinement proofs are successfully computed at both `O1-` and `O1` contain at least one local array, and table 4 presents statistics for the `O1-` validation experiments for these procedures. For each procedure, we show the number of source lines of code in C (`SLOC`), the number of assembly instructions in A (`ALOC`), the number of local variables ($\#_{al}$), the number of loops ($\#_{loop}$), the number of procedure calls ($\#_{fcall}$), and the maximum loop nest depth (`D`). The `eqT` column shows the validation times (in seconds). The Nodes and Edges columns show the number of nodes and edges in the final product graph, and `BT` and `EXP` is the number of backtrackings and the number of (partial) candidate product graphs explored by Dynamo respectively. $\#_q$ is the total number of SMT queries discharged, and `Avg. qT` is the average time taken by an SMT query in seconds for the refinement check.

In a separate experiment, we split the large procedures in `bzip2` into smaller procedures, so that Dynamo successfully validates the `O1-` compilation of the full modified `bzip2` program: the splitting disables some compiler transformations and also reduces the correlation search space.

Through our experiments, we uncovered and reported a bug in recent versions of `z3`, including `z3-4.8.14` and `z3-4.12.5`, where for an input satisfiability query Ψ, the SMT solver returns an unsound model (counterexample) that evaluates Ψ to false [z3b 2024]. When a modern SMT solver is used to validate compilations produced by a mature compiler, a bug may be found on either side.

6 RELATED WORK AND CONCLUSIONS

CoVaC [Zaks and Pnueli 2008] automatically identifies a product program that demonstrates observable equivalence for deterministic programs. Counter [Gupta et al. 2020] extends CoVaC to support path-specializing transformations, such as loop unrolling, through counterexample-guided search heuristics. We extend these prior works to support refinement between programs performing dynamic allocations with non-deterministic addresses for local variables and stack.

Recent work on bounded TV [Lee et al. 2021] models allocations through *separate blocks*, so a pointer is represented as a combination of a block-ID and an offset into a block. While this suffices for the bounded TV setting, our problem setting requires a more general representation of a dynamically-allocated variable (e.g., allocation-site) and a more general SMT encoding.

CompCert provides axiomatic semantics for memory (de)allocation in the source Clight program, and proves their preservation along the compilation pipeline [Leroy and Blazy 2008]. They restrict their proof method to CompCert's preallocation strategy for local variables, possibly to avoid the manual effort required to write mechanized proofs for a more general allocation strategy. Preallocation of local variables has also been used in prior work on TV for a verified OS kernel [Sewell et al. 2013]. Preallocation can be space inefficient and cannot support VLAs and `alloca()`. Further, TV for a third-party compiler cannot assume a particular allocation strategy.

We provide a semantic model, refinement definition, and an algorithm to determine the correctness of a third-party translation from an unoptimized high-level representation of a C program to an optimized assembly program in the presence of dynamically-allocated local memory. Our semantic model and definition of refinement require that for allocations and procedure calls that reuse stack space, their relative order is preserved in both programs. While our experiments show that this suffices in practice, a more general definition of refinement, that admits transformations that may reorder (de)allocations while reusing stack space, is perhaps a good candidate for future work.

DATA-AVAILABILITY STATEMENT

The Dynamo tool that supports section 5 is available on Zenodo [Rose and Bansal 2024a] with instructions for complete reproducibility of the presented results.

ACKNOWLEDGMENTS

We thank Shubhani Gupta for contributing towards scalability improvements of the translation validation tool. We thank Abhishek Dang for carefully reading previous drafts of the paper, and pointing out several errors, and making important suggestions that improved the paper significantly.

REFERENCES

2024. Z3 bug report for an unsound model. https://github.com/Z3Prover/z3/issues/7132.

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-Guided Correlation Algorithm for Translation Validation. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 221 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428289

John L. Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millenium. *IEEE Computer* 33, 7 (July 2000), 28–35.

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 631–645. https://doi.org/10.1145/3192366.3192377

Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 1004–1019.

Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice Hall Professional Technical Reference.

Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 752–776.

Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages.* ACM Press, 42–54. http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf

Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Washington, DC, USA, 372–382. https://doi.org/10.1109/PACT.2011.68

David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

KedarS. Namjoshi and LenoreD. Zuck. 2013. Witnessing Program Transformations. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Lecture Notes in Computer Science, Vol. 7935. Springer Berlin Heidelberg, 304–323. https://doi.org/10.1007/978-3-642-38856-9_17

George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

Abhishek Rose and Sorav Bansal. 2024a. *Artifact for paper "Modeling Dynamic (De)Allocations of Local Memory for Translation Validation"*. https://doi.org/10.5281/zenodo.10797459

Abhishek Rose and Sorav Bansal. 2024b. *Modeling Dynamic (De)Allocations of Local Memory for Translation Validation.* Technical Report. IIT Delhi. https://arxiv.org/abs/2403.05302

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 32–41.

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based Translation Validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742. http://dl.acm.org/citation.cfm?id=2032305.2032364

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the 15th International Symposium on Formal Methods* (Turku, Finland) *(FM '08)*. Springer-Verlag, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-540-68237-0_5

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/2491956.2462164

Received 21-OCT-2023; accepted 2024-02-24