arXiv:submit/8302156 [cs.PL] 29 Oct 2026

CIVIC: CHECKING INDEX LOWERING IN MACHINE LEARNING COMPILERS

Samarth Narang [1] **Brandon Reagen** [2]

ABSTRACT

Indexing errors in ML compilers can select the wrong tensor elements while preserving types, shapes, and allocation bounds. We present CIVIC, a framework for checking selected indexing properties at compiler transformation boundaries. Compiler-specific frontends independently derive expected mappings from source semantics and express them in the execution coordinates used by the generated computation. Each frontend records these mappings, domains, access predicates, and storage layouts in a witness contract, adding expressions recovered from compiler output when checking generated accesses. A shared checker uses exact Presburger reasoning to check bounds, coverage, write uniqueness, and address correspondence over the encoded domains, returning integer counterexamples for violations. Applying these guarantees to compiler behavior requires faithful witness construction, which we assess by comparing CIVIC's verdicts with independent semantic references. Across 300 parameter configurations from three transformation families, the verdicts agree in every case. Using the same checker across four compiler integrations, CIVIC detects 26 bugs in MLIR, five in XLA, and one in IREE, and reproduces six known TorchInductor bugs.

1 INTRODUCTION

ML compilers use tiling, vectorization, layout conversion, and sharding to assign tensor computations to loop iterations, vector lanes, threads, and devices. As these assignments change, the compiler rewrites the index expressions and access predicates that determine which tensor elements each execution point reads or writes. Errors in this correspondence can produce incorrect results while preserving the expected types and shapes, even when every access remains within its allocation.

An XLA optimization that combines nested dynamic slices preserves the output shape and stays within the allocation but selects the wrong elements because it fails to preserve slice clamping (Section 2).

Translation validators such as Alive2 (Lopes et al., 2021) and MLIR-TV (Bang et al., 2022) check transformations within supported intermediate representations, while Clément & Cohen (2022) connect generated Halide assignments to tensor specifications through compiler-propagated annotations. Across ML compilers, indexing requirements appear in different forms, including transformed IR, printed indexing expressions, and compiler layout objects. These representations nevertheless share requirements on which tensor elements an execution reaches, whether required elements are omitted, and whether generated addresses agree with source semantics.

CIVIC (*Compiler Index Verification with Independent Contracts*; Figure 1) expresses these requirements through a common witness contract. Each frontend independently derives the expected mapping from source semantics in the generated computation's execution coordinates and supplies the domains, predicates, layouts, and observed expressions needed by the selected obligations. The shared checker evaluates the resulting integer relations without modeling the tensor-value computation or executing the generated program. This separation supports four compiler integrations while placing responsibility for faithful semantic extraction and source-to-output correspondence in the frontends.

Across four compiler integrations, CIVIC detects 26 bugs in MLIR, five in XLA, and one in IREE, and reproduces six known TorchInductor bugs. We assess frontend fidelity by comparing CIVIC's verdicts with independent semantic references on 300 parameter configurations from three transformation families, and by testing its response to controlled changes in compiler output and witness encodings. We also measure how often CIVIC produces a checkable contract, the time spent in the checker, and the additional cost of capturing compiler output and constructing witnesses.

The paper makes the following contributions to checking index transformations in ML compilers.

- A witness contract that relates source regions and expected accesses to transformed execution coordinates,

[1]NVIDIA, New York, NY, USA [2]New York University, New York, NY, USA. Correspondence to: Samarth Narang <sanarang@nvidia.com>, Brandon Reagen <bjr5@nyu.edu>.

Preprint.

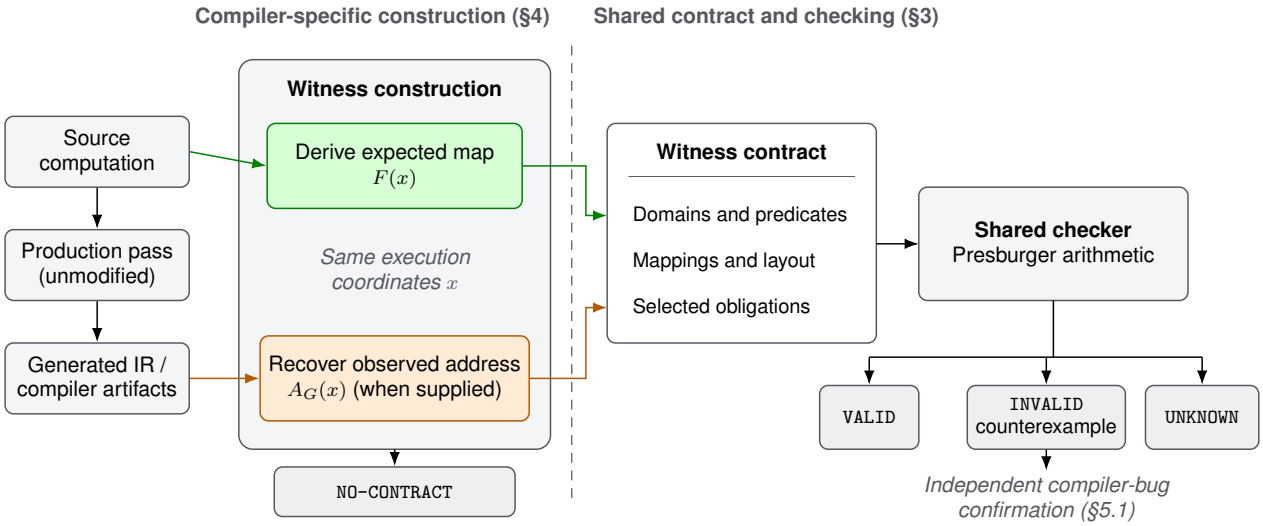


Figure 1. CIVIC architecture. Compiler-specific frontends construct witnesses from source semantics and compiler output, and a shared checker evaluates the selected obligations. Failed construction yields `NO-CONTRACT`, while an `INVALID` verdict requires independent confirmation before supporting a compiler-bug claim (Section 5.1).

allowing one checker to distinguish address errors, out-of-bounds accesses, omitted work, and duplicate writes.

- Compiler-specific frontends that establish correspondence between source computations and compiler output and construct witnesses for one shared checker. The integrations support native pass audits, saved XLA intermediate representations, and TorchInductor capture during compilation with checking deferred until compilation returns.
- An evaluation across four open-source compilers that independently confirms reported bugs, assesses frontend fidelity and sensitivity to controlled changes, and measures contract construction rates, checking cost, and integration overhead.

2 Motivating Example and Design Requirements

We illustrate CIVIC's approach with a bug in XLA's algebraic simplifier, where combining nested dynamic slices fails to preserve the source program's indexing semantics. The source program first extracts nine elements from an 18-element tensor at a requested start p, then extracts four elements from that result at a requested start q. Each slice clamps its start to keep its window within its own input, so output coordinate i, with $0 \le i < 4$, accesses input element

$$F(i) = \mathrm{clamp}(p, 0, 9) + \mathrm{clamp}(q, 0, 5) + i.$$

The first clamp uses $18 - 9 = 9$, while the second uses $9 - 4 = 5$ because it operates on the nine-element intermediate tensor. The affected simplifier replaces these two slices with a four-element slice of the original tensor, using $18 - 4 = 14$ as its upper bound and computing

$$G(i) = \mathrm{clamp}\big(q + \mathrm{clamp}(p, 0, 14), 0, 14\big) + i.$$

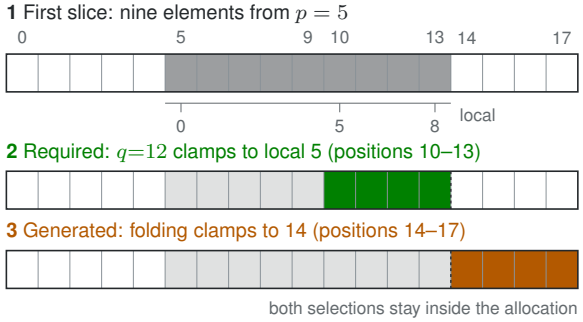


Figure 2. Nested-slice bug in XLA at $p = 5$ and $q = 12$. The second slice should select positions 10–13 within the shaded intermediate tensor, but faulty folding selects 14–17. Both selections remain within the original allocation, so allocation-bounds checks cannot detect the error.

For $p = 5$ and $q = 12$, the source selects elements 10 through 13 after clamping q to 5, while the simplified program selects elements 14 through 17 (Figure 2). Both programs return four elements and keep every access within the 18-element allocation, so shape and allocation-bounds checks do not detect this change in element selection.

To check this transformation, CIVIC compares $F(i)$, the input index required by the source operations, with $G(i)$, the

input index selected by the generated program. Its frontend derives F independently from the source slice semantics and recovers G from the production simplifier's output. With $p = 5$ and $q = 12$, the checker reports a mismatch at $i = 3$, where the generated program selects element 17 although the source requires element 13. We confirm that this indexing error produces incorrect outputs on both CPU and GPU by comparing the program's results with an independent reference.

This comparison requires the frontend to associate source and generated accesses from the same transformation instance and express their mappings in common execution coordinates. Transformations that also change iteration domains, access predicates, or write ownership require further checks, which Section 3 defines as separate obligations in the witness contract.

3 THE INDEX WITNESS CONTRACT

A *witness contract* combines an index witness with the obligations selected by the frontend for the shared checker to evaluate. The witness records the source and execution domains, expected mapping, access predicate, and storage information needed by those obligations. The frontend derives the expected mapping from source semantics and supplies an observed expression when checking correspondence with compiler output. Figure 1 shows where the contract sits between the two.

Witness representation. An index witness connects positions in the generated computation to elements of the source tensor. Lowering can change the coordinates used to describe these positions. For example, tiling replaces a single tensor index with a tile index and a position within that tile. The witness records these two coordinate spaces as the logical source domain $S \subseteq \mathbb{Z}^r$ and the generated execution domain $E \subseteq \mathbb{Z}^n$. When coverage is selected, S denotes the logical region that the audited computation is required to reach, which need not be the entire tensor.

The frontend derives $F : E \to \mathbb{Z}^r$ from source semantics, with $F(x)$ specifying the logical tensor coordinate expected at execution point x. In a partially filled tile, some execution points map outside the tensor, and a guard or mask determines which accesses execute. The witness records this condition as a predicate $P(x)$, taken to be true for an unguarded access, and defines the enabled domain $D = \{x \in E \mid P(x)\}$.

The storage layout maps the expected logical coordinate $F(x)$ to a location in memory. The witness records this layout as a base element offset o and constant element strides $s_0, \ldots, s_{r-1}$, which determine the expected element address

$$A_F(x) = o + \sum_{k=0}^{r-1} s_k F_k(x),$$

where $F_k(x)$ is coordinate k of $F(x)$.

The witness also records the allocation extent N, measured in elements, so the checker can determine whether the expected address $A_F(x)$ lies within the allocation. To check a generated memory access, the frontend recovers the expression $A_G(x)$ for its element address from compiler output. CIVIC then checks whether $A_G(x) = A_F(x)$ for every enabled execution point $x \in D$, comparing the address computed by the generated code with the address required by the source mapping and storage layout.

The frontend also records whether the access is a read or a write and selects the obligations appropriate to the audited transformation. These choices determine whether the checker requires write uniqueness, under which distinct enabled execution points must write to distinct logical destinations.

Correctness obligations. Logical bounds keep enabled accesses within the source region S, while coverage ensures that they reach every element of that region. Together, these properties establish $F(D) = S$, where $F(D)$ is the set of logical elements reached by enabled accesses, without restricting how often an element is accessed.

That distinction matters when several execution points must access the same element. If lanes 0 and 1 must both read element 7, a mask that disables lane 1 leaves $F(D)$ unchanged but removes a required read. When the source requires an access at every point with $F(x) \in S$, *mask completeness* requires the generated mask to enable exactly those points. The frontend supplies the predicate $P(x)$ extracted from that mask.

For writes, the transformation may additionally require each logical destination to be written at most once. Write uniqueness enforces this requirement within a single witness, without comparing physical addresses or separate store sites. Reads and reduction updates omit this obligation because they permit repeated accesses to an element.

The pointwise conditions below apply to every enabled point $x \in D$. Write uniqueness also considers a second point $x' \in D$, and its condition applies to every pair.

Label	Obligation	Required condition
Log.	Logical bounds	$F(D) \subseteq S$
Cov.	Coverage	$S \subseteq F(D)$
Uniq.	Write uniqueness	$F(x) = F(x') \implies x = x'$
Phys.	Physical bounds	$0 \le A_F(x) < N$
Addr.	Address correspondence	$A_G(x) = A_F(x)$
Mask	Mask completeness	$D = \{x \in E \mid F(x) \in S\}$

The frontend selects a checking mode using the combinations below, whose labels refer to the preceding conditions. A `VALID` verdict establishes only the obligations selected by that mode over the encoded domain.

Mode	Log.	Cov.	Uniq.	Phys.	Addr.	Mask
`address`	—	—	—	—	✓	—
`in_bounds`	✓	—	—	—	—	—
`mask`	✓	—	—	—	—	✓
`physical_bounds`	—	—	—	✓	✓*	—
`full`	✓	✓	✓†	✓	✓*	—

*Address correspondence is checked only when the witness supplies A_G. †Write uniqueness is checked only for writes.

Physical bounds constrain the expected address $A_F(x)$, while address correspondence establishes that the generated address $A_G(x)$ uses the same location. Together, they show that generated accesses follow the specified layout and stay within the allocation; without A_G, the bounds guarantee applies only to the expected address. Numerical correctness, synchronization, and reduction semantics remain outside the contract, as discussed in Section 6.

Exact checking. CIVIC checks each selected obligation by searching for an integer counterexample using MLIR's Presburger library (Pitchanathan et al., 2022). The checker first represents the expected mapping as a relation R, pairing each enabled execution point x with the logical coordinate $y = F(x)$ that it is expected to access,

$$R = \{(x, y) \mid x \in D \wedge y = F(x)\}.$$

For each selected obligation, the checker constructs a separate relation containing the integer assignments that violate it. An empty violation relation establishes that the obligation holds throughout the encoded domain, while an integer point in a nonempty relation provides a counterexample.

The bounds queries look for enabled accesses whose logical coordinates lie outside S or whose expected storage addresses lie outside $[0, N)$. To check coverage, the checker forms the set $F(D)$ of logical elements reached by at least one enabled execution point. A logical coordinate y belongs to this set exactly when R contains a pair (x, y) for some $x \in D$. The difference $S \setminus F(D)$ identifies required elements for which no such execution point exists. Write uniqueness retains the association between execution points and logical elements, pairing entries (x, y) and (x', y) that share the same logical coordinate. Adding the condition $x \neq x'$ makes each solution identify two distinct execution points writing to the same element.

For address correspondence, the checker forms the difference $\Delta(x) = A_G(x) - A_F(x)$ between the generated and expected addresses. Since the addresses are integers, they differ exactly when $\Delta(x) \geq 1$ or $\Delta(x) \leq -1$, so the checker searches for enabled points satisfying either condition. Before reporting a counterexample for any obligation, CIVIC verifies that its integer coordinates satisfy the corresponding violation relation.

These checks use Presburger arithmetic, which supports affine expressions together with floor division and remainder by constant divisors, but reasons over mathematical integers. The frontend must therefore preserve the compiler's integer widths, cast behavior, overflow behavior, and division rounding when constructing the witness. For example, a cast to a narrower integer type can change a value by discarding its high bits. The frontend can treat the cast as leaving the value unchanged only when range analysis proves that every possible input fits in the narrower type.

For piecewise mappings such as clamps, the frontend records each expected and generated expression with the condition under which it applies. It constructs address and logical-bounds witnesses over the intersection of each pair of conditions, so both expressions describe the same execution points.

The frontend must provide enough cases to cover every required execution, since the checker evaluates only the cases it receives and does not verify that none are missing. The checker also does not check coverage or write uniqueness across cases.

Verdicts and scope. CIVIC returns `VALID` when the selected obligations hold throughout the encoded domain and `INVALID` with a counterexample when an obligation fails. When the checker cannot resolve an obligation, it returns `UNKNOWN`, while the frontend records `NO-CONTRACT` when it cannot construct the required witness. We report fixed-input and symbolic results separately because they establish guarantees over different sets of parameter values.

For audits that produce several witnesses, we give `UNKNOWN` precedence over other checker outcomes and otherwise report `INVALID` if any witness fails or `VALID` if all pass. We record construction failures and runs without checker verdicts separately, so a combined `VALID` result covers only the witnesses checked.

These guarantees rely on the source specification, frontend, checker, and Presburger library, with the frontend responsible for preserving the computation's mappings, domains, and arithmetic. Since the checker does not validate that extraction, we independently confirm reported violations before counting compiler bugs (Section 5.1).

4 WITNESS CONSTRUCTION

4.1 Capturing the Transformation Boundary

The MLIR and IREE frontends keep the original code and run the production compiler pass on a copy, using its production options. Before running the pass, they label selected operations so they can find the corresponding operations in the transformed code. This preserves the source code for deriving the expected accesses and the transformed code for recovering the generated accesses.

Three IREE integrations can add a helper call to the clone to expose how the pass rewrites indices. The materialize-encoding integration can also supply target information required by an embedded encoding resolver. These additions change the pass input while preserving the loop bounds, steps, and mapping attributes used by the audit.

A pass may duplicate or remove operations, so the frontend must check how the marked source operations relate to the transformed code. It uses numbered markers or lists of identifiers when it needs to distinguish several marked operations or generated copies. It then checks that the number of matches agrees with what the transformation should produce, accounting for copies introduced by unrolling and operations removed during cleanup. When the correspondence also depends on the surrounding loops, the frontend checks their structure. Missing required matches, unexpected copies, or an incompatible loop structure cause the frontend to report `NO-CONTRACT`.

The XLA frontend matches instructions by name across saved snapshots of XLA's intermediate representation (HLO). It rejects duplicate names that would make this association ambiguous and retains both the source operations and the indexing expressions produced by the compiler. The TorchInductor frontend captures compiler objects directly during compilation, so it can construct witnesses from those objects without matching names across saved snapshots.

4.2 Deriving and Observing Accesses

In Figure 3, the transformation splits a four-element reduction into two groups of two elements, using p to identify a group and r to identify an element within it. Both indices range from zero to one, so the source element at (p, r) is $k = 2p + r$. The frontend derives the expected mapping $F(p, r) = 2p + r$ from this split and the original access, independently of the transformed code.

Because the tensor uses consecutive storage beginning at offset zero, the expected address is $A_F(p, r) = 2p + r$, while the frontend extracts $A_G(p, r) = p$ from the generated code. At $(p, r) = (1, 1)$, the source requires element 3 but the generated computation reads element 1, so CIVIC reports an address-correspondence violation even though

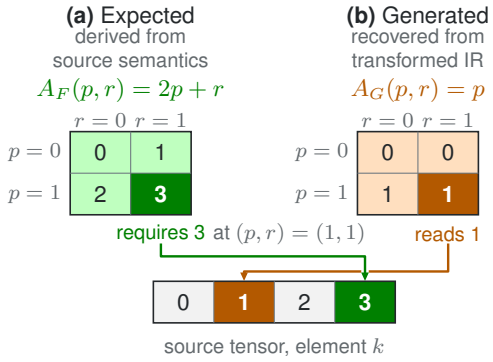


Figure 3. Witness construction for split reduction (Section 4.2). CIVIC derives the expected address $2p + r$ from the source access and recovers the generated address p from transformed IR. At $(p, r) = (1, 1)$, these select elements 3 and 1, respectively, yielding `INVALID`.

both addresses remain in bounds. Deriving the expected address from the generated expression would instead compare p with itself and miss the bug.

The MLIR and IREE frontends recover address expressions by tracing the arithmetic that computes each memory access's indices. They express the address and any condition controlling the access in terms of constants and execution coordinates, such as loop indices and thread identifiers. The XLA frontend reads indexing expressions already printed by the compiler, while the TorchInductor frontend reads live compiler objects, including layouts and rewritten index expressions. All four frontends must preserve the compiler's arithmetic behavior in the expressions they encode, as described in Section 3.

4.3 Integrating Compiler Frontends

All four frontends use the same witness format and 636-line checker. MLIR and IREE invoke it within their audit tools, while XLA and TorchInductor serialize witnesses for checking. XLA processes saved compiler output; TorchInductor records information during compilation and checks it after compilation returns.

Supporting another transformation expressible by the contract requires a frontend to derive its expected indexing, recover the relevant compiler output, and establish their correspondence in common execution coordinates. The shared checker then applies the existing obligations to the resulting witness. Table 1 summarizes the information used by the evaluated frontends and the properties they check. When a frontend cannot establish the required correspondence or represent the necessary ranges, layouts, allocation sizes, or arithmetic, it reports `NO-CONTRACT`.

Frontend or check group	Information used	What CIVIC checks
MLIR	Indices and access conditions in transformed IR; for scalar state, exit values encoded as a one-point address claim	Address correspondence, bounds, mask completeness, coverage, and write uniqueness
IREE	Indexing expressions from the audited transformation	Address correspondence and bounds
XLA	Indexing expressions printed by the compiler	Address correspondence, bounds, and mask completeness
Inductor algebra	Two symbolic forms of an indexing expression	Equality under the frontend's arithmetic interpretation
Inductor layout	Source and compiler-produced layouts	Whether the produced layout gives the expected indexing, together with bounds checks
Inductor index expressions	An input expression and the compiler's returned rewrite	Expression equality and bounds
Inductor predicates	Compiler structure or emitted source text	Whether a structural or textual test passes, with its result encoded as address equality
Inductor extents	Source and generated extents	Bounds, coverage, and write uniqueness; the address query compares identity-shaped expressions that are equal by construction.
Inductor printing	Printed index expressions and copied expected addresses	Printing syntax for floor division and remainder in `index_to_str`. Copied-address equality holds automatically.

Table 1. Information and checks used by CIVIC's frontends. Each row summarizes a group of implementations, and individual witnesses may select subsets of the listed properties. Coverage and write uniqueness use concrete parameters, while address, bounds, and mask checks may retain constrained runtime variables. The appendix provides further details on frontend capabilities and restrictions.

5 EVALUATION

We evaluate CIVIC across MLIR, XLA, IREE, and TorchInductor, examining the compiler bugs it detects, the fidelity of its frontends, and the applicability and cost of checking. We also run witness construction and checking without GPU access to assess whether CIVIC can audit retained compiler artifacts when the target accelerator is unavailable.

5.1 Evaluation Methodology

We construct witnesses from executions of production compiler passes or from saved compiler output, keeping each witness associated with the transformation it describes. For each `INVALID` result, we examine the source computation, the witness, and the corresponding compiler output to determine whether the violation reflects a compiler bug or an error in witness construction. We include a finding in the bug count only after independently confirming the incorrect behavior, and count several failing cases as one bug only when the evidence shows that they arise from the same compiler defect.

We confirm 11 of the 26 MLIR bugs by comparing the transformed computation's output with an independent reference, while the compiler's verifier rejects the generated IR in one further case. For the remaining 14, we establish the error by comparing the generated operations and index expressions with the applicable specification. For each reported bug, we record the compiler revision tested and the evidence used to confirm the error.

We use GPT 5.6 Sol agents to develop the frontends and Claude Fable 5 agents to guide review and adversarial testing. An incorrectly constructed witness can hide a compiler bug or cause CIVIC to report a violation in correct code, so we evaluate frontend accuracy separately in Section 5.3.

5.2 Compiler Bugs Detected

CIVIC detects 26 confirmed bugs in MLIR, five in XLA, and one in IREE, and reproduces six previously reported TorchInductor bugs (Table 2). Our searches found no prior upstream report for 27 of the 32 MLIR, XLA, and IREE bugs, while the remaining five correspond to known issues or earlier fix attempts. The evaluation also led to 36 filed issues, including 13 with identified fixes merged upstream and one additional issue that no longer reproduces on the tested prerelease build.

For the previously reported TorchInductor bugs, CIVIC identifies the violated indexing obligation and provides a counterexample that characterizes the failure. In one CPU loop-splitting bug, the transformation partitions a 3×50 output region using integer division but fails to handle the remainder, producing only 135 of the 150 required writes. When we run only the bounds checks on this witness, CIVIC returns `VALID` because the represented writes stay within bounds, even though 15 output elements remain unwritten. Coverage additionally requires every output element to receive a write, so it detects the omission and returns `INVALID`. The appendix reports comparisons across checking modes for this case and two MLIR bugs.

Compiler	Bugs	Prior	New	Violated obligations
MLIR	26	4	22	logical 13, address 11, coverage 5
XLA	5	1	4	address 4, mask 1, logical 1
IREE	1	0	1	address 1
TorchInductor	6	6	0	coverage 2, logical 2, physical 2
Total	38	11	27	

Table 2. Confirmed compiler bugs, counted once per root cause, with *Prior* indicating an earlier upstream report or fix attempt and *New* indicating that our search found neither. We show the three most frequent violated obligations for each compiler, whose counts overlap when a bug violates more than one obligation. Address violations include scalar-state equalities encoded as one-point address claims.

5.3 Frontend Validation

We assess witness construction through independent semantic references, instruction specifications, controlled changes to compiler artifacts and encodings, and historical-bug replays. These experiments test whether frontends preserve the required indexing information and whether the selected obligations expose violations represented in a witness.

We first evaluate 100 parameter configurations for each of three programs covering SCF loop peeling, XLA slice/update-slice, and XLA reduce-window. We choose parameter configurations not used during frontend development and keep the frontend implementations fixed during evaluation. The appendix records the corrections to input construction and capture parsing made while establishing the comparison. For SCF, we execute the transformed loop and compare its iteration sequence with the source loop. For XLA, we enumerate the captured index calculation's domain and compare each selected tensor element with the element required by the source operation. All 300 configurations produce witnesses, and every verdict agrees with the independent reference. CIVIC accepts the 66 SCF configurations with matching iteration sequences and rejects the 34 that execute extra iterations, while all 200 XLA configurations agree with source semantics and receive `VALID`.

We also check GPU mappings against PTX ISA 9.0 (NVIDIA Corporation, 2025), which specifies which tensor elements each thread accesses and how those elements are arranged in its registers. All 20 matrix-instruction mappings and six `ldmatrix` address mappings we evaluate agree with the specification for every supported thread and register.

After these reference comparisons, we deliberately alter the encodings and test whether our evaluation inputs expose the changes. These mutation tests revealed that address correspondence could still hold for indices outside the source region, leaving violations undetected when the witness omitted logical-bounds checks. We added those checks and repeated each mutation experiment using its existing inputs. For XLA, the fraction of mutations changing a `VALID` baseline to another outcome increased from 60.2% to 95.5%; every change we observed was `INVALID`. TorchInductor counts only mutations producing `INVALID`, and that fraction increased from 86.3% to 90.8%.

We test whether the frontends faithfully capture indexing changes using 24 configurations across six XLA templates and 16 across four TorchInductor view forms. For each configuration, we retain the original compiler output and create two variants, one equivalent and one with an indexing error, as established by an independent reference against the unchanged source operation. We modify XLA's printed index expressions or TorchInductor's returned view objects before witness construction.

CIVIC returns `VALID` for all original and equivalent versions and `INVALID` for all 40 versions with confirmed indexing errors. Independent checks also confirm the reported mismatches against the modified compiler output. The TorchInductor errors remain within the backing allocation, so detecting them requires checking which element an access selects rather than whether its address is in bounds. For all 72 XLA versions, we additionally compare the witness's execution domain and both mappings with the independent reference, finding agreement throughout. The appendix describes the controls and distinguishes the original test plan from revisions to inputs and fault-injection procedures made after initial results were inspected.

Earlier development testing exposed an error in the XLA frontend's treatment of 32-bit integer arithmetic. It encoded these values as though they could grow without overflowing. For inputs that cause overflow, the witness and the generated code could therefore compute different indices from the same inputs. The submitted frontend declines any region whose arithmetic can leave the 32-bit range over its admitted domain, bounding every intermediate rather than only the input symbols, and continues to check nearby regions in which no intermediate can wrap.

Lastly, we replay historical compiler bugs by removing their fixes and rerunning the same audited inputs with the frontends and selected checks unchanged. In IREE, six of seven experiments change from `VALID` to `INVALID`, with independent checks of the compiler output confirming the corresponding errors on those same inputs. The remaining experiment produces an unsupported form, for which the frontend reports `NO-CONTRACT`. The frontends were developed with knowledge of these defects, so these replays assess detection on known cases.

This method also revealed a limitation of the XLA replay

experiments, whose captured index expressions did not expose the historical bugs. Removing the fixes changed the printed formulas, but those formulas still computed the same indices under the frontend's floor-division interpretation, so CIVIC returned `VALID` in both cases. The appendix also documents frontend failures found during development and the resulting repairs or restrictions, including a false `VALID` that the revised frontend avoids by declining the affected cases.

5.4 Applicability

Figure 4 reports `VALID` or `INVALID` verdicts, `NO-CONTRACT` outcomes, compiler failures, and unresolved harness outcomes for each corpus, using the counting units described below.

We draw the MLIR and IREE inputs from existing compiler tests, supplementing MLIR's regression suite with generated programs that vary rank, divisibility, and vectorization. We count each selected MLIR transformation stage separately. For IREE, compiling 199 end-to-end tests for CUDA and CPU yields 1,317 dispatch sources,[1] on which we perform 6,699 audit invocations. The XLA corpus comes from compiled JAX programs, where we collect the output instruction of each fused computation for which the compiler prints an indexing map. We retain all selected cases, including compiler failures and cases where the frontend cannot construct a witness.

The TorchInductor frontend captures information during compilation for comparisons and additional bounds checks, so a single compilation can contribute many records. Across 22,161 compilation cases, we collect 1,951,123 records covering the different checks described in Table 1. Table 3 groups the records from the 12,000-program seeded campaign by kind of check and reports how many receive `VALID` or `INVALID`.

In the IREE and TorchInductor corpora, the checker resolves every constructed witness as `VALID` or `INVALID`, while the frontends report `NO-CONTRACT` for 42.8% of IREE audit invocations and 23.5% of TorchInductor records. These rates include cases where the audited transformation does not occur as well as cases where it occurs but the frontend cannot construct a witness.

Construction can fail when the frontend cannot recover an index range, represent the relevant arithmetic, or establish correspondence between source and generated operations. For example, the XLA frontend declines runtime-dependent index definitions that it cannot encode, while a narrowing cast can prevent construction when the frontend cannot represent its effect on the index value. The appendix describes these restrictions and reports the reasons retained for the analyzed cases.

[1] An IREE dispatch is a unit of work executed on a target device; its dispatch source is the corresponding compiler IR.

Kind of record	Total records	`VALID` or `INVALID`
Compiler-produced layouts	32,720	30,728
Compiler-returned index expressions	115,700	86,420
Structural or textual tests	89,124	67,030
Copied address expressions*	30,520	19,382
Algebraic rewrites	835,545	608,337
Coverage and bounds	20,492	17,602
Additional logical-bounds checks	81,276	62,392
Total	1,205,377	891,891

*For copied address expressions, address equality holds by construction.

Table 3. TorchInductor records from the 12,000-program seeded campaign, grouped by construction. The final column counts records receiving `VALID` or `INVALID`. Additional logical-bounds checks produce separate records associated with other checks, so the total does not count independent memory accesses. All 16 `INVALID` outcomes are logical-bounds violations in companion records, corresponding to one previously reported storage-bounds bug.

5.5 Checking Cost

We time individual obligations on witnesses from the MLIR census, excluding witness construction, parsing, and process startup. Address correspondence and bounds checks remain below 0.3 ms at the 95th percentile, while write uniqueness takes 16.5 ms at that percentile. Coverage varies more widely across witnesses, with a median of 3.0 ms and a 95th-percentile time of 182.7 ms.

In a separate comparison on identical witness queries, Presburger has lower decision times for bounds and address correspondence, while Z3 is faster for coverage and write uniqueness; the appendix reports the protocol and full results.

5.6 Integration Cost

We measure the cost of using CIVIC through each frontend, including witness construction and checker invocation. The experiments run on an AMD Ryzen 7 7800X3D host, with compiler revisions and measurement protocols reported in the appendix.

For TorchInductor, we compare compilation with and without CIVIC recording the compiler information needed for its checks. Each run starts with fresh compilation caches, and we measure elapsed time from creation of the `torch.compile` wrapper until the first compiled call returns, covering both compilation and the initial execution.

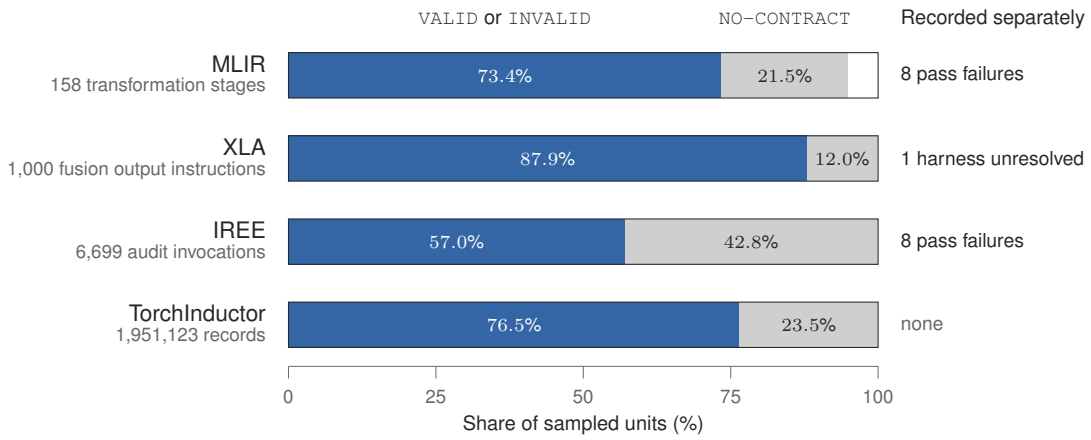


Figure 4. Applicability within each corpus, using the sampling units shown. `VALID` and `INVALID` indicate resolved checks; `NO-CONTRACT` indicates that no witness was constructed. Unfilled portions represent compiler failures or unresolved harness outcomes. Sampling differences prevent direct comparison across compilers.

Figure 5 separates the additional time incurred while recording this information from the cost of checking the resulting witnesses afterward. Recording increases the time to the first result by 1.8–50.1% across the tested configurations. The largest increase occurs in the loop-splitting workload, which takes 5.51 s longer than the baseline, while checking its witnesses separately at process exit takes 103.4 ms.

For MLIR and IREE, we compare complete audit-tool runs with the production-pass tool, including startup and parsing in both measurements. We collect 20 paired trials for each of eight configurations per compiler. Across the pooled audit-to-reference time ratios, the median increases are 15.5% for MLIR and 6.3% for IREE. Median audit times for individual configurations range from 8.9 to 15.7 ms for MLIR and from 10.5 to 44.6 ms for IREE, with the largest IREE relative cost reaching 3.98 times its reference time.

For XLA, we measure offline processing of five saved HLO inputs, including witness construction and checker invocation but excluding the compilation that produced them. The per-input median processing times range from 7.79 to 68.34 ms, with a median of 8.42 ms across the five inputs.

6 Discussion and Limitations

CIVIC checks indexing over the encoded execution domain without executing generated code, allowing developers to inspect captured transformations when the target accelerator is unavailable. We tested this capability with an XeGPU audit and two XLA nested-slice audits in a sandbox without GPU device access, with each run constructing and checking witnesses from retained source or compiler artifacts. The runs reproduced the XeGPU violations and the XLA verdicts at the recorded parameter values, while tracing showed that the audit processes opened no GPU devices or GPU runtime libraries.

The guarantees concern selected indexing obligations and depend on faithful frontend construction. They exclude numerical correctness, synchronization, and reduction semantics. Mask completeness additionally assumes that every in-bounds access is required, an assumption the MLIR frontend does not verify. A violation therefore supports a compiler-bug claim only after we establish that the omitted access was required. XLA ownership checks apply to the recorded device assignment and do not establish correctness for other tensor partitions.

LLMs can help compiler engineers extend CIVIC's frontends by drafting semantic encodings and tests, as in our development workflow, while the shared checker evaluates the resulting witnesses. These encodings require the independent semantic checks and adversarial testing described in Section 5.3, because incorrect mappings or domains can conceal bugs or produce unreachable counterexamples even when checking is exact. An industrial compiler team also runs CIVIC automatically in weekly audits of two internal production ML compilers, checking its frontends against current compiler revisions. CIVIC has reported 42 and 12 findings on these compilers, respectively, which we report separately from the open-source evaluation.

7 Related Work

Translation validation. Alive2 (Lopes et al., 2021) checks LLVM refinement using SMT with bounded loop reasoning, while MLIR-TV (Bang et al., 2022) encodes supported MLIR operations in SMT and HEC (Yin et al., 2025) uses equality saturation. Clément & Cohen (2022) combine

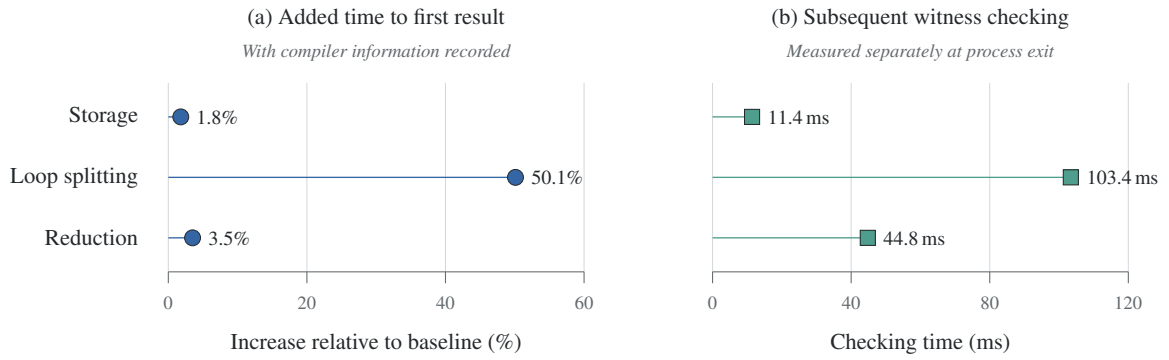


Figure 5. CIVIC's effect on the time to the first compiled result (left) and the subsequent witness-checking time (right) for three TorchInductor workloads. Each point is the median over 15 retained trials. In (a) the percentage is that median increase divided by the program's median baseline time to first result, not a median of per-trial ratios; the appendix reports the spread and the differing convention used for the native audit tools.

affine and SMT reasoning with compiler-propagated annotations to check access coverage and value correctness in Halide. CIVIC's frontends independently derive expected mappings from source semantics and recover generated accesses, expressing selected indexing obligations through a common contract that supports checking across compiler representations.

CIVIC's MLIR audits construct witnesses by running a production pass, whereas MLIR-TV checks a supplied source/output pair. On the convolution-to-im2col test, MLIR-TV validated its stored target but could not encode the current pass output checked by CIVIC. CIVIC also declined dynamic-subview tests; the appendix explains why no tested pair received conclusive verdicts from both. Halide's validator requires annotated Halide IR, so using it on our MLIR and XLA outputs would require translating them into that form and validating the translation.

Polyhedral and layout reasoning. Verdoolaege et al. (2012) check equivalence of static affine programs through dependence graphs, while Bhaskaracharya et al. (2025) model CuTe and Triton layouts as integer relations. CIVIC uses integer relations to connect source requirements with transformed accesses, distinguishing bounds, coverage, write uniqueness, and address correspondence.

Verified compilation and scheduling. Courant & Leroy (2021) and Liu et al. (2024) verify polyhedral code generation and tensor-language lowering, respectively. Loopy (Namjoshi & Singhania, 2016) and Exo (Ikarashi et al., 2022) check loop and scheduling transformations. CIVIC checks selected properties of outputs from existing production passes, conditional on frontend fidelity.

Accelerator and GPU verification. Melchert et al. (2025) and BOLT (Gaonkar et al., 2026) validate compiler mappings to accelerators. GPUVerify (Betts et al., 2012) checks races and barrier divergence, Volta (Driscoll et al., 2026) checks equivalence for a supported class of GPU kernels, and RESOLVE (Gargary et al., 2026) combines testing, reduction, and proof. CIVIC focuses on indexing at compiler boundaries, leaving synchronization and numerical equivalence outside its contract.

Tensor rewrite verification. TensorRight (Arora et al., 2025) and TensorEgg (Kothari & Baronia, 2026) verify tensor rewrite rules across supported parameter spaces. CIVIC checks the indexing of a particular transformation instance over its encoded domain.

Compiler testing. Tzer (Liu et al., 2022) mutates IR and pass sequences, NNSmith (Liu et al., 2023) generates programs for differential testing, and WhiteFox (Yang et al., 2024) generates optimization-triggering programs using LLMs. CIVIC checks selected indexing obligations without executing the generated computation, while independent confirmation determines whether a violation reflects a compiler defect.

8 Conclusion

CIVIC expresses selected indexing requirements through a common witness contract across four compiler integrations. Compiler-specific frontends connect source semantics to transformed indexing, allowing one shared checker to evaluate these requirements without executing the generated computation. The evaluation reports 38 confirmed compiler bugs and agreement with semantic references across 300 configurations, supporting static indexing checks combined with frontend validation.

REFERENCES

Arora, J., Lu, S., Jain, D., Xu, T., Houshmand, F., Phothilimthana, P. M., Lesani, M., Narayanan, P., Murthy, K. S., Bodík, R., Sabne, A., and Mendis, C. TensorRight: Automated verification of tensor graph rewrites. *Proceedings of the ACM on Programming Languages*, 9(POPL): 832–863, 2025.

Bang, S., Nam, S., Chun, I., Jhoo, H. Y., and Lee, J. SMT-based translation validation for machine learning compiler. In *Computer Aided Verification (CAV)*, 2022.

Betts, A., Chong, N., Donaldson, A. F., Qadeer, S., and Thomson, P. GPUVerify: A verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012. URL https://nchong.github.io/papers/oopsla12.pdf.

Bhaskaracharya, S. G., Acharya, A., Hagedorn, B., and Grover, V. Modeling layout abstractions using integer set relations, 2025. URL https://arxiv.org/abs/2511.10374.

Clément, B. and Cohen, A. End-to-end translation validation for the Halide language. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022.

Courant, N. and Leroy, X. Verified code generation for the polyhedral model. *Proceedings of the ACM on Programming Languages*, 5(POPL), 2021. doi: 10.1145/3434321.

Driscoll, B., Dubey, K., Wei, A., Kayal, N., Sharma, R., and Aiken, A. Equivalence checking of ML GPU kernels. *Proceedings of the ACM on Programming Languages*, 10 (OOPSLA2), 2026. doi: 10.1145/3839470.

Gaonkar, A., He, M., Li, Y., Huang, B.-Y., Cheung, A., Canumalla, V., Smith, G. H., Tatlock, Z., Fedyukovich, G., Malik, S., and Gupta, A. Verification of compiler-to-accelerator mappings for machine learning accelerators, 2026. URL https://arxiv.org/abs/2609.30651.

Gargary, A. V., Martínez, G., Burckhardt, S., Ebner, G., Jangda, A., Musuvathi, M., and Sorensen, T. RESOLVE: Language-agnostic validation of GPU kernels through testing, reduction, and proof, 2026. URL https://arxiv.org/abs/2610.05683.

Ikarashi, Y., Bernstein, G. L., Reinking, A., Genc, H., and Ragan-Kelley, J. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2022. doi: 10.1145/3519939.3523446.

Kothari, A. and Baronia, D. TensorEgg: Formally verifying rewrite rules and applying them to tensor programs. In *Proceedings of the IEEE/ACM 14th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pp. 138–142, 2026. doi: 10.1145/3793656.3793689.

Liu, A., Bernstein, G., Chlipala, A., and Ragan-Kelley, J. A verified compiler for a functional tensor language. *Proceedings of the ACM on Programming Languages*, 8 (PLDI), 2024. doi: 10.1145/3656390.

Liu, J., Wei, Y., Yang, S., Deng, Y., and Zhang, L. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–26, 2022.

Liu, J., Lin, J., Ruffy, F., Tan, C., Li, J., Panda, A., and Zhang, L. NNSmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 530–543, 2023.

Lopes, N. P., Lee, J., Hur, C.-K., Liu, Z., and Regehr, J. Alive2: Bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2021.

Melchert, J., Terrill, C., Perez-Lopez, Á. R., Barrett, C., and Raina, P. Automated translation validation of a compiler for statically scheduled accelerators. In *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 198–208. TU Wien Academic Press, 2025. doi: 10.34727/2025/isbn.978-3-85448-084-6_26.

Namjoshi, K. S. and Singhania, N. Loopy: Programmable and formally verified loop transformations. In *Static Analysis Symposium (SAS)*, 2016. doi: 10.1007/978-3-662-53413-7_19.

NVIDIA Corporation. *Parallel Thread Execution ISA Version 9.0*. NVIDIA Corporation, 2025. URL https://docs.nvidia.com/cuda/archive/13.0.0/parallel-thread-execution/index.html. CUDA Toolkit 13.0 documentation archive.

Pitchanathan, A., Grover, K. S. S., Weber, M., and Grosser, T. Bringing presburger arithmetic to MLIR with FPL. In *12th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2022. URL https://www.impact-workshop.org/impact2022/papers/paper9.pdf.

Verdoolaege, S., Janssens, G., and Bruynooghe, M. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems*, 34(3), 2012. doi: 10.1145/2362389.2362390.

Yang, C., Deng, Y., Lu, R., Yao, J., Liu, J., Jabbarvand, R., and Zhang, L. WhiteFox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2): 709–735, 2024.

Yin, J., Song, Z., Agostini, N. B., Tumeo, A., and Yu, C. HEC: Equivalence verification checking for code transformation via equality saturation. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 1181–1196, Boston, MA, 2025. USENIX Association. ISBN 978-1-939133-48-9.

AI USAGE DISCLOSURE

We used generative AI tools to assist with implementation, experimental design and analysis, literature search, and manuscript preparation. GPT 5.6 Sol agents assisted with frontend development and experimental code, while Claude Fable 5 agents assisted with review and adversarial testing. We also used ChatGPT to discuss design choices, identify related work, plan experiments, and draft and revise text throughout the manuscript. Code assistance included compiler-specific frontends, experimental harnesses, tests, and reference implementations, together with debugging and interpretation of experimental failures.

APPENDIX

This appendix provides the experimental details supporting the evaluation of CIVIC. It describes frontend capabilities and restrictions, validation procedures, comparisons across checking modes and with execution and MLIR-TV, the audits run without GPU access, applicability corpora, measurement protocols, the comparison of checking backends, and upstream reporting status, together with the detailed results referenced in the main text. Tables numbered A1 onward belong to this appendix.

A FRONTEND CAPABILITIES AND RESTRICTIONS

Table A1 lists each evaluated frontend or check group with the representation it supports and the restrictions on its evaluated paths. Capability the checker has but no evaluated frontend uses is excluded from the table. In particular, the checker accepts parameters as separate symbols, but no evaluated frontend combines that representation with coverage or write uniqueness, so no reported result rests on it.

B FRONTEND VALIDATION

Configuration selection. Before collecting results, we fix each family's candidate population in a manifest and select 100 configurations by ordering the `sha256` hashes of their configuration keys. Table A2 records the parameter sets, population sizes, and manifests for these experiments. The SCF grid contains 180 unique $(lb, ub, step)$ triples, of which two were used during development and are excluded, leaving 178 candidates. Five additional development configurations have lower bounds outside the grid and therefore never enter the candidate population. In both XLA families, the frontend and the independent reference process the same captured compiler artifact, with the reference computing its results without calling the frontend.

Comparison is exhaustive over the enumerated domain rather than sampled: SCF enumerates every induction value and the XLA families enumerate every point of the printed domain. Configurations whose iteration count would exceed the harness's recording capacity of 64 entries are excluded when the population is generated, rather than truncated during the run, so no retained configuration is compared over a partial domain.

Harness corrections. We corrected two harness errors before producing the reported results. In the SCF experiment, the execution and audit arms initially used different representations of the loop bounds, which caused the compiler to take different transformation paths. We reran both arms using the same program form and pass options. In the reduce-window experiment, the capture parser failed to recognize maps without a symbol list when the window extent was one; correcting the parser recovered those maps. Neither correction changed the frontend implementation or the selected parameter configurations, and all 100 configurations in each affected family were rerun.

We compare 22 matrix-instruction mapping cases and six `ldmatrix` address mappings against PTX ISA 9.0. For each supported case, we enumerate all 32 lanes and every register position defined by the operand type. Twenty matrix-instruction cases and all six `ldmatrix` mappings agree with the specification throughout these domains. The evaluated MLIR interface rejects the BF16 A- and B-operand cases before returning a mapping, so we classify those two cases as unsupported rather than as mismatches.

The 22 cases are `m16n8k16` for f16, bf16, and f32 accumulators; `m16n8k4` and `m16n8k8` for tf32; the f64 form; and the i8 and i4 integer forms.

Where mutations are applied. The mutation experiments alter different stages of witness construction. MLIR mutations change the frontend's expected derivation, IREE and TorchInductor mutations change extracted values or observed expressions, and XLA mutations change witness fields directly. Each experiment therefore exercises the checking path downstream of its mutation site. We evaluate extraction separately by modifying captured compiler output while leaving the expected mapping unchanged. Mutations alter domain bounds, arithmetic signs, operand order, and offsets; the MLIR experiments require a rebuild before the tests are rerun.

What counts as a detected change. The harnesses do not share a kill criterion, so their rates are not comparable and we report no combined total. `mutate_lanes.py` compares the tuple of full per-witness diagnostic lines; `gate0_mutations.py` compares verdict and counterexamples separately; `audit_xla_mutations.py` compares the verdict against its baseline mode; and `stage5b_mutations.py` marks a mutation killed when the run yields any `INVALID`. The last criterion counts an outcome rather than a transition, so it is sound only if the unmutated run did not already produce `INVALID`.

Why logical-bounds companions were added. The initial experiments revealed that several frontends recorded the source tensor's logical bounds but requested only address correspondence. Expected and observed addresses can agree even when both refer to an element outside those bounds, so changing the source domain left the selected check unaffected. We added companion logical-bounds witnesses for transformations that require every enabled access to remain within the source domain, without changing the shared checker. The paragraphs below give each campaign's rate

Frontend or check group	Supported representation and checks	Restrictions
MLIR (77 lanes)	Indices and access conditions traced through arithmetic to loop induction variables and thread identifiers; for scalar state, exit values encoded as a one-point address claim. Address correspondence, bounds, mask completeness, coverage, and write uniqueness.	Narrowing casts; `divsi` or `remsi` with a dividend not provably non-negative; ambiguous association.
IREE (18 lanes)	Indexing expressions recovered from the transformed clone, re-found by markers. Address correspondence with in-bounds companions.	Three implementations can add a helper call before running the pass, and the materialize-encoding implementation can also supply target information. The evaluated paths require static layouts and support only the indexing forms listed for each implementation.
XLA (12 implementations)	Indexing expressions printed by the compiler as an `IndexingMap`. Address correspondence with in-bounds companions, and mask completeness on the SPMD owner-mask lane. Runtime starts are retained as constrained variables over printed ranges.	Arithmetic that can leave s32 over the admitted domain; width-changing bitcasts; unparseable maps.
Inductor algebra	Two symbolic forms of an indexing expression, compared for equality under the frontend's arithmetic interpretation over a declared box.	Equality under that interpretation only. A `VALID` result is not a claim about the emitted Triton or C++.
Inductor layout	The compiler-produced layout against the source layout, with bounds checks.	Requires a provably non-negative `ModularIndexing` base and intermediates within the kernel's index width.
Inductor index expressions	The compiler's returned expression against its input, with bounds checks.	As for layouts.
Inductor predicates	A structural or textual test, with its result encoded through address equality.	The address comparison records the test's outcome; it is not an independent address recovery.
Inductor extents	Source and generated extents over real loop ranges from the compiler: bounds, coverage, and write uniqueness.	The accompanying address query compares identity-shaped expressions, which are equal by construction.
Inductor printing	Printed index expressions for floor division and remainder in `index_to_str`, against copied expected addresses.	Copied-address equality holds automatically; the substantive check is the emitted spelling.

Table A1. Frontend capabilities and restrictions on the evaluated paths. Individual witnesses select subsets of the listed checks; coverage and write uniqueness use concrete parameter values.

Family	Parameter sets	Unique candidates	Excl. dev.	Selected	Manifest
SCF peeling	$lb \in \{-9,-1,0,1,7\}$; $step \in \{1,2,3,4,8\}$; $ub - lb \in \{-step, -1, 0, 1, step-1, step, step+1, 2\,step+1\}$	180	2	100	`scf-peeling-manifest.json` `49a313ec`
XLA slice/ update-slice	operand $\in \{(9,5),(13,5),(8,4),(16,6)\}$; update $\in \{(3,5),(5,5),(2,4),(4,6),(1,5)\}$; per-dimension starts covering negative, zero, last valid, and beyond-bound values	600	1	100	`dus-manifest.json` `753fa822`
XLA reduce-window	input $\in \{(10,8),(12,6),(7,5),(16,4)\}$; window $\in \{1,\dots,5\}$; stride $\in \{1,\dots,4\}$; pad $\in \{0,1,2\}$	239	0	100	`rw-manifest.json` `2fed04bb`

Table A2. Fresh-configuration populations. Candidates are unique parameter tuples after deduplication; *Excl. dev.* counts configurations used during development that fall inside the population and are removed. Selection keys are `scf-peeling:lb:ub:step`, `dus:operand:update:start`, and `rw:(input,window,stride,pad)`. The manifest column gives the frozen file and the first eight hexadecimal digits of its `sha256`.

before and after that change, against its own denominator and criterion.

XLA. For XLA, all 37 baseline runs return `VALID`. After adding logical-bounds checks, 84 of 88 mutations change the verdict to `INVALID`, giving a verdict-change rate of 95.5%, up from 60.2% over the same mutations before the change. Although the harness also counts changes to `NO-CONTRACT` or `UNKNOWN`, neither outcome occurs in this campaign.

TorchInductor. TorchInductor uses a different criterion, counting mutations whose runs produce `INVALID`. The recorded summary reports 119 such outcomes among 131 mutations, or 90.8%, up from 86.3% before the change. The per-run records for this campaign were not retained, so its baseline outcomes cannot be rechecked. A separate retained campaign records valid baselines for all twelve tested obligations, with no `INVALID` in any baseline run, but does not establish the baselines of the 131-mutation campaign.

Additional mutation results. In IREE, 146 mutations change the verdict from `VALID` to `INVALID`, while two cause witness construction to return `NO-CONTRACT`. Of these two, one changes a previously `VALID` outcome and the other changes a previously `INVALID` outcome.

In MLIR, 59 surviving mutations affect resolver branches that the tested passes never invoke, so they leave witness construction unchanged on this corpus. For example, replacing subtraction with addition in an arithmetic resolver has no effect when the pass supplies its access relation through composed affine maps, bypassing that resolver.

Structural validation of captured artifacts. A separate batch tests witness construction on program structures used neither in development nor in the fresh-configuration study above. Six XLA templates supply 24 configurations and four TorchInductor source-view forms supply 16. Each configuration is evaluated in three versions: the original compiler output, an equivalent version computing the same indices by a different expression, and an incorrect version carrying a single-operator indexing error. The XLA versions alter the printed index expression after capture. The TorchInductor versions alter the object returned by `ExpandView.create` through a test-only wrapper installed before the frontend's hook, so the frontend captures the altered object through its own extraction path; neither the expected derivation nor the witness is edited. Three MLIR SCF templates supply a further 12 configurations evaluated as originals only, because the audit pass clone-runs the production pass and builds the witness in one invocation, leaving no capture boundary at which generated IR can be altered without a test-only hook. Table A3 gives the outcomes.

Beyond verdict agreement we record the following. For XLA, the witness's execution domain, expected mapping, and observed mapping each agree with the independent reference in all 72 variants, and all 24 incorrect variants yield a counterexample at an execution point the enumerated domain reaches. For TorchInductor, the intended compiler hook ran and a witness was constructed in all 48 variants with no declines; the altered object reached the capture point in all 32 modified variants; and the counterexample agrees with the altered object's mapping in all 16 incorrect variants. For SCF, the eight declines are correct outcomes rather than failures: three configurations have ranges the pass divides exactly, leaving no partial tail to peel, and five have empty or reversed ranges. The single `INVALID` is a reversed range with a positive step, whose source loop runs zero iterations while the audit emits two witnesses for a live generated loop. That is the empty-source-range defect class already represented in the bug totals, reproduced here in a structure of two sibling loops with distinct store sites rather than in the single-loop template.

Population	Original	Equivalent	Incorrect
XLA, six templates and 24 configurations	24 `VALID`	24 `VALID`	24 `INVALID`
TorchInductor, four view forms and 16 configurations	16 `VALID`	16 `VALID`	16 `INVALID`
SCF, three templates and 12 configurations	3 `VALID`, 1 `INVALID`, 8 `NO-CONTRACT`	Not evaluated	Not evaluated

Table A3. Additional frontend-validation results. XLA and TorchInductor variants modify captured compiler artifacts before witness construction, with independent references confirming equivalence or an indexing error. The SCF experiment evaluates original compiler outputs only. Injected faults are validation controls and do not contribute to the compiler-bug totals.

Revisions to this batch. Three changes to the test inputs and the fault-injection procedure are recorded, with the initial outcomes retained in the artifact. Two XLA templates were changed from tuple-rooted to chained forms *before* the population was frozen and before any outcome existed, because the capture interface emits maps only for a root instruction's operands; the frozen manifest records the chained configurations. The three SCF templates were rewritten *after* their first run, which declined all 12 configurations because arithmetic-wrapped loop bounds leave the write index not statically evaluable; the parameter configurations are unchanged but the programs are not, so those 12 rows are a revised population. The TorchInductor fault operator was corrected *after* its first run: `ExpandView.create` returns a reinterpreted view for one of the four forms, which the frontend reads through its layout strides rather than its reindexer, so that form's four incorrect variants were initially unaltered controls whose `VALID` outcomes were correct for an unaltered object rather than missed detections. Three further corrections were to the harness and left the tested programs unchanged: an operand selector that read a scalar start rather than the sliced operand, a reachability check that parsed integers out of a temporary filename, and a compilation cache that served later compiles from cache so the intended hook never ran.

Earlier captured-output controls. Two smaller controls preceded this batch and are retained as supporting evidence. In an MLIR test, adding an offset to a generated index changes CIVIC's verdict from `VALID` to `INVALID`. A separate change narrows an index to an integer type that may not preserve its value, causing the frontend to report `NO-CONTRACT` because it cannot represent the modified arithmetic faithfully.

Finite-width arithmetic in the XLA frontend. The XLA frontend declines a region when it cannot establish that every arithmetic intermediate remains within the signed 32-bit range. This check covers both address expressions and expressions used in branch conditions, because overflow in a condition can change which piece of the mapping executes. The frontend admits conditions in stages, using bounds already established before each condition, so a condition cannot justify the arithmetic used to compute itself.

The regression tests cover a sum whose runtime start ranges over all signed 32-bit values, which the frontend declines, and a restricted range for which the same sum remains representable and is checked. They also verify that a condition cannot justify its own arithmetic and that an empty region requires no arithmetic evaluation. Table A4 names the two controls; they pass at revision `5b51578180`.

Control	Test
Declines what can wrap; checks what cannot	`test_s32_arithmetic_gate_declines_only_what_can_wrap`
No self-justifying condition	`test_s32_gate_does_not_justify_arithmetic_with_its_own_condition`

Table A4. Regression controls for finite-width arithmetic, in `frontends/xla/test/test_xla_witness.py`.

Frontend failures found during development. Three development failures led to repairs or restrictions, and we report them separately from the fresh-configuration results. First, the XLA frontend encoded 32-bit arithmetic as unbounded; the finite-width restriction above replaces that encoding. Second, the original `audit-loop-coalescing` frontend failed to recover the generated loop and reused the source action count, producing an empty execution domain and a false `VALID`. Its revised construction declines the affected cases, including some genuinely empty transformations, so the change prevents the false `VALID` without restoring detection in that configuration (Appendix C gives the consequence for the execution comparison). Third, an XLA witness admitted values inconsistent with a runtime variable's definition, producing an unreachable counterexample; constraining the variable to its clamped definition removed that report.

IREE historical-bug replays. Section 5.3 reports seven IREE experiments that remove a known fix and rerun the same audited input: six change from `VALID` to `INVALID` and one yields `NO-CONTRACT`. The independent control for each is listed below. Each control is an observation on the audited input, made without the witness or the checker. Five recipes audit programs, and for those we re-run the audited production pass on that input at both builds and compare the generated IR.

- **iree-23142**: at the pin the pass rebuilds the pre-encoding coordinates before comparing them; the revert compares `linalg.index 0` against `linalg.index 1` in tiled coordinates.
- **iree-24266**: the revert folds the `insert_slice` into the store, dropping the load of the destination and writing only rows 1–2 of a 4×1 result.
- **iree-24349**: after the unit dimension is dropped the pin emits `index 1` and `index 2`; the revert leaves `index 2` and `index 3`.
- **iree-gather**: the `uge` subview-bound guard before `amdgpu.gather_to_lds` is present at the pin and absent at the revert.
- **iree-inner-tiled**: the revert's own verifier rejects the audited input with `inner_tiled op shape does not match iteration bounds`.
- **iree-data-tile-24437**: this lane audits a layout attribute rather than a program, so there is no pass output to compare. Asking each build what undistributed tile type that same attribute requires gives `vector<2x16x1xf32>` at the pin and `vector<32x1xf32>` at the revert, differing by exactly the swizzle permutation the fix restores.
- **iree-24217**: the revert does not transform the trigger at all, so the production pass no longer emits the form the frontend recognizes and witness construction fails. The `NO-CONTRACT` is the correct outcome and is not evidence about the defect either way.

Two limits apply throughout. The observations are of generated IR or required types, not of execution. And detection here is of a defect reintroduced by reverting a known fix, with frontends written knowing it, so these experiments do not measure prospective discovery.

C Obligation and Execution Comparisons

Checking modes compared. Table A5 runs one captured instance under different selected obligations, for the 135-of-150 coverage case of Section 5.2 and for two MLIR bugs.

Case	Mode	Verdict	Violated obligation	Restricted mode and verdict
In-bounds wrong address (MLIR split reduction)	`address`	`INVALID`	address correspondence	`in_bounds`: `VALID`
Out-of-bounds access (MLIR affine super-vectorization)	`physical_bounds`	`INVALID`	physical bounds	`address`: `VALID`
135-of-150 omission (TorchInductor loop splitting)	`full`	`INVALID`	coverage	`in_bounds`, `address`, `physical_bounds`: each `VALID`

Table A5. Within each row, the same captured instance is checked under the listed modes; unlisted modes were not evaluated. For the out-of-bounds case the claimed address equals the expected address by construction, so its `VALID` address result does not demonstrate independent address agreement. The in-bounds wrong-address case uses an independently recovered expression.

Obligation-selection control. The artifact also retains a valid reduction as a scope control, not listed in Table A5. Write uniqueness does not apply to an accumulation, so the frontend does not select it there; the control records an obligation that is inapplicable, which differs from one that is selected and holds.

Execution comparison. Three confirmed bugs have execution drivers, exercising vector-to-SCF lowering with a negative start, SCF front peeling on a dynamic zero-trip loop, and OpenACC lowering with a negative step. On all three, directed execution and CIVIC both expose the defect. Random execution also detects the two cases that have random drivers; the OpenACC case has none. Input policies were fixed beforehand, without consulting witnesses or counterexamples. The vector-to-SCF signal arises from an out-of-bounds read and is reported as a memory-safety outcome for every arm. The comparison establishes no detection-yield advantage for CIVIC on these cases, and the provisional timings support no performance conclusion.

The execution comparison evaluates a selected audit configuration for each bug, whose outcome can differ from that of another CIVIC configuration. For loop coalescing, the selected `audit-loop-coalescing` configuration is the one whose original construction produced a false `VALID` and whose revised construction declines the affected cases (Appendix B). A separate configuration, `audit-signed-trip-products`, detects the compiler bug through logical-bounds violations in witnesses representing the coalesced actions.

For `WhileMoveIfDown`, the comparison's selected address check does not cover the faulty value forwarding, whereas `audit-scf-value-reconstruction` derives the expected exit state from the source recurrence and compares it with the generated exit state through a one-point address claim. The two quantities it compares, 20 and 2, are encoded exit states rather than addresses recovered from a generated memory access. That configuration returns `INVALID`, with independent execution subsequently confirming the defect. We retain the original configuration-specific outcomes in the comparison while counting each compiler bug once in the detection results. A sixth selected case, `async-parallel-for`, could not be run: lowering its `async.execute` form through the async-to-LLVM pipeline fails inside LLVM code generation, and we did not establish the cause. It therefore contributes no execution comparison.

D Comparison with MLIR-TV

We examine the applicability of CIVIC and MLIR-TV using five test cases from CIVIC's existing findings and additional tests from MLIR-TV's own suite. The five cases belong to one upstream issue, llvm-project #220505, in which transformations change a structured operation's iteration domain without remapping `linalg.index`. MLIR-TV checks refinement between supplied source and output programs, whereas CIVIC runs a production transformation and checks selected indexing obligations through witnesses constructed from that execution. The experiments therefore examine which inputs each integration supports and whether it reaches a verdict on faulty outputs and correct controls. Because we selected the bug cases from CIVIC's existing findings, these results do not establish the tools' relative bug-detection rates.

Baseline setup and health. We use MLIR-TV HEAD `897c870`, whose source is identical to `2e53307` of 13 November 2024, built in its own CI image `strikef/mlir-tv-ci-base:2024-11-22` (LLVM `9894cd5febbb`, Z3 4.8.13, cvc5 1.2.0). An earlier attempt to build it against LLVM revisions outside this image failed, so the pinned image is used throughout. At this pin MLIR-TV passes its Opts suite (12/12) and all its Litmus suites except one expected-invalid memref case, `copy-fill-bad`, which times out at both 30 s and 600 s with Z3 and with cvc5; its correct counterpart validates in 121 ms. MLIR-TV runs isolated under bubblewrap, takes each pair as source and output files, and is stopped externally at 30 s. Its own exit codes are recorded: unsupported input, target parse failure,

solver timeout, and the external limit are distinct outcomes. MLIR-TV encodes no `scf` or `vector` operations, so the shared ground is loop-free `linalg`, `tensor`, and `memref` transformations.

Pairs and ground truth. For each of five bug tests we form a faulty pair and a control pair. The faulty source is the retained bug test, byte-identical. The control keeps the same `linalg.generic` and audit configuration but places the anchored `linalg.index` on a dimension whose loop position the rewrite preserves, so it is a correct instance of the same pass, not a fixed version of the faulty input; no fixed build is available. Outputs come from running the upstream rewrite at the witness-opt pin (LLVM `57b5017e0102`) through its transform op (`structured.split_reduction`, `structured.decompose`, or `structured.pack`), with parameters equal to the source's own audit configuration; nothing is hand-written. CIVIC runs `-audit-linalg-partial-reduction` or `-audit-linalg-index-rewrites` followed by `-verify-index-witness` on the source; the audit clone-runs the same upstream function. Expected behaviour is established by execution: source and output are lowered for the CPU at the pin and run with `mlir-runner` on eight seeded random inputs. Every faulty output differs from its source on 8 of 8 inputs, and every control output matches its source on 8 of 8.

Results. CIVIC's verdicts agree with the execution checks on all ten source/output pairs, returning `INVALID` for the five faulty cases and `VALID` for their controls (Table A6). MLIR-TV reaches no conclusive verdict on these pairs within the 30 s budget, with four runs timing out, four rejecting unsupported inputs, and two unable to parse the generated target. The timeouts occur on both split-reduction cases and their correct controls, and persist in separate diagnostic runs that raise the per-query SMT limit to 600 s.

Convolution-to-im2col across compiler versions. MLIR-TV's `tests/opts/conv2d-to-img2col/nhwc_filter` test transforms a two-dimensional convolution into im2col followed by matrix multiplication, using a $1 \times 16 \times 16 \times 4$ input and a $3 \times 3 \times 4 \times 16$ filter with `f32` elements. Both tools accept the source unchanged, but they check outputs from different compiler versions. MLIR-TV validates its stored target, produced by IREE's 2022 `-iree-flow-convert-conv2d-to-img2col` pass, in 5.6 s. CIVIC's `-audit-linalg-convolution` audit with `mode=im2col` runs the current `linalg::rewriteInIm2Col` implementation at the evaluated LLVM revision and returns `VALID` for two witnesses covering logical bounds on the packed input and address correspondence of the im2col gather.

The current rewrite expresses the gather through a single indexing map containing floor division and remainder, which MLIR-TV rejects with "unsupported affine Expr." CIVIC, in turn, constructs witnesses from the rewrite it runs and cannot check MLIR-TV's stored target as an external input. Executing the source and both targets on eight seeded inputs produces identical results, but the integration restrictions prevent us from obtaining conclusive verdicts from both tools on the same source/output pair.

MLIR-TV's `fold-memref-subview-op` tests (`const_tensor`, `var_tensor`, `rank_reduction`) cover the memref alias fold that CIVIC audits with `-audit-memref-fold`. CIVIC declines all three as `NO-CONTRACT`, because its audit supports only static subviews and these have dynamic offsets, sizes, or strides.

E Auditing Without GPU Access

Section 6 reports an XeGPU audit and two XLA nested-slice audits run without GPU device access. Each audit starts from retained source or compiler artifacts and performs witness construction and checking; none replays a serialized witness. This experiment supersedes an earlier single XeGPU run on a host where device nodes remained visible.

Sandbox. The audits run under bubblewrap with the host root mounted read-only and a fresh minimal `/dev` containing no `/dev/nvidia*` and no `/dev/dri`. `/proc` and `/tmp` are private and every namespace is unshared, so there is no network; `CUDA_VISIBLE_DEVICES` is empty and `JAX_PLATFORMS=cpu`. Inside the sandbox, opening `/dev/nvidia0` fails and `nvidia-smi` sees no GPU. The host has an NVIDIA GPU, so device independence is shown by the sandbox and the trace rather than by the absence of hardware.

Audits. Table A7 reports three audits that reproduce their recorded verdicts. The XeGPU audit (B1) runs `witness-opt` with `-audit-xegpu-sg-to-lane-values` and `-verify-index-witness`, while the XLA audits (B2 and B3) construct witnesses from HLO captured before and after the algebraic simplifier. At the recorded caller inputs, B2 returns `INVALID` and B3 returns `VALID`; their symbolic-region results cover different parameter domains and are reported separately, following Section 3. B2's symbolic analysis checks 210 of 256 clamp regions, yielding 200 `VALID` and 10 `INVALID` results, and declines the remaining 46 because their 32-bit arithmetic can wrap.

B3 also produces an `INVALID` symbolic-region witness at $(3, -2^{31}, 15)$, a sample admitted by the encoded leaf ranges but excluded by the program. The retained HLO constrains

Bug test	CIVIC: faulty / control	MLIR-TV: faulty / control	Selected CIVIC obligation	Baseline restriction
Split reduction, extent 4 `linalg-split-reduction-index-remap`	`INVALID` / `VALID`	timeout / timeout	address corresp.	No verdict within 30 s on either pair
Split reduction, extent 8 `...-nondegenerate`	`INVALID` / `VALID`	timeout / timeout	address corresp.	As above
Convolution decomposition `linalg-decompose-convolution-index-remap`	`INVALID` / `VALID`	unsupported / unsupported	address corresp.	Source rejected: a convolution-shaped `linalg.generic` is outside its encoded forms
Convolution decomposition, dropped dimension `...-dropped-index-remap`	`INVALID` / `VALID`	unsupported / unsupported	address corresp.	As above
Pack index remap `linalg-pack-index-remap`	`INVALID` / `VALID`	target unparsable / target unparsable	address corresp.	Output contains `linalg.pack`/`linalg.unpack`, absent at the baseline's LLVM; renaming them to `tensor.pack`/`tensor.unpack` (diagnostic only) still gives "Unknown op"

Table A6. CIVIC and pinned MLIR-TV on five MLIR bug tests, each with its faulty production output and a correct control instance of the same pass, under a 30 s limit per validation. Execution confirms that every faulty output differs from its source and every control matches. The five tests are rewrite sites of one open upstream issue, llvm-project #220505, not five upstream issues. Timeouts, unsupported inputs, and parse failures are distinct outcomes and are not verdicts.

Audit	Input artifacts (`sha256` prefix)	Audited target	Selected obligation	Recorded verdict	Repro.	Wall
B1 XeGPU	`xegpu-packed-lane-reduction-provenance.mlir` (`8369cb81`)	XeGPU subgroup-to-lane value distribution, run on a clone	address corresp. (16 witnesses); logical bounds (16)	16 `INVALID` [address corresp.] at points (0) and (1), claimed 1 vs. layout 0 and 0 vs. 1; 16 `VALID`	identical	0.06 s
B2 XLA nested dynamic slice	`xa/repro/before_algsimp.txt` (`a329df09`), `after_algsimp.txt` (`430f5ce8`)	AlgebraicSimplifier nested dynamic-slice fold	address corresp.	`INVALID` at $(3, 5, 12)$: claimed 17 vs. layout 13	identical	0.47 s
B3 XLA nested dynamic slice, start in range	`xa/inrange/before_algsimp.txt` (`8afee87b`), `after_algsimp.txt` (`17f19a59`)	Same fold; the program clamps the start into range	address corresp.	`VALID` at the caller input	`VALID`	0.47 s

Table A7. Audits reproduced in a sandbox without GPU device nodes. Verdicts for B2 and B3 are at the caller input; symbolic-region results are reported separately in the text. Wall times are single observed durations and support no performance claim.

the outer start to $\min(5, \max(0, \mathit{start})) \in [0, 5]$ and the inner start to $\mathrm{clamp}(0, \cdot, 14) \in [0, 14]$, so neither sampled start is reachable. The earlier domain repair described in Appendix B constrained a different variable; these symbolic-region checks still use full signed 32-bit leaf ranges and flag counterexamples for validation against their producers. B2 exhibits the same limitation, and we exclude these unreachable counterexamples from the compiler-bug count.

Trace attribution. The whole run was traced with `strace -f` for `execve`, `open`, `openat`, and `openat2`, and every open was attributed to the program that issued it. The checked names were `/dev/nvidia*`, `/dev/dri`, `libcuda`, `libnvidia*`, `libcudart`, `libze_loader`, `libOpenCL`, and `iris_dri`. The audit processes, `witness-opt` and the Python interpreter running the XLA frontend, opened no GPU-named files. The only GPU-named opens in the trace come from the sandbox probes (the `/dev/nvidia0` open test and `nvidia-smi`, whose device opens fail with `ENOENT`) and from the JAX recheck below, whose CUDA plugin opens the `libcuda` and `libcudart` library files but no device node, and runs on the CPU.

Independent rechecks. Every emitted witness was decided again by Z3 through the harness of Appendix H, and each counterexample was replayed by that harness's concrete evaluator, which uses the checker's arithmetic but none

of its decision code. All 16 B1 address-correspondence counterexamples recheck and all 16 B1 logical-bounds witnesses are `VALID`; B2's reported witness and its region witness both recheck; B3's caller-input witness is `VALID`, and its region-witness counterexample is a genuine violation of that witness but unreachable in the program, as above. For B2, the retained differential recipe `xla_nested_dynamic_slice_roll.py`, run on the CPU inside the sandbox with jaxlib 0.11.2, gives $[50, 60, 70, 80]$ under `jit`, whereas op-by-op execution and NumPy give the expected $[10, 20, 30, 40]$, confirming the defect by execution. B1 has no execution recheck, because its target is Intel XeVM and no such device or runtime is involved.

The XeGPU compiler defect is confirmed separately from this audit, by its upstream fix: the audited compiler is at pin `57b5017e0102`, and the fix is `04659071744a` (llvm-project #223077).

Host dependencies. `witness-opt` (`sha256` prefix `533a195c`, LLVM `57b5017e0102`) links only `libz`, `libstdc++`, `libm`, `libgcc_s`, and `libc`. B2 and B3 need only the Python 3.12.3 standard library together with the XLA frontend scripts and the shared witness package; they read the retained HLO dumps and need no JAX, XLA, or CUDA. The recheck tools are not needed by the audits.

F Applicability Corpora and Outcomes

Table A8 gives the counts behind Figure 4, and Table A13 those behind Figure 5. The qualifications stated in those captions apply unchanged.

IREE audit invocations. The 1,317 dispatch sources give 4,247 lane/source pairs, since one source can be applicable to more than one audit lane. Each pair contributes a baseline invocation, which places no restriction on where within the source the audit applies. Five lanes additionally invoke the audit once per occurrence of a lane-specific token in the input text, and one lane once per candidate found by matching a layout-producing operation to its consumer; together these give 6,699 invocations. A textual occurrence count can exceed the number of matching operations, so some invocations have no operation to audit and record `NO-CONTRACT`.

TorchInductor corpus composition. The 22,161 compilation cases comprise a pinned release, 19 nightly configurations, and 12,000 seeded programs. Only the seeded campaign retains records at the granularity Table 3 reports, so that decomposition covers its 1,205,377 records rather than all 1,951,123.

What these rates do not measure. The applicability rates report construction and checking outcomes for the recorded audit opportunities. They do not measure whether a frontend selected every obligation relevant to the transformation it audits.

TorchInductor's `records[]` keeps up to three entries per (lane, hook, verdict) per case, so sampled reason counts are not proportional to the population. We classify recorded reasons as unsupported semantics, extraction failures, or correspondence failures. Exact totals exist per (lane, hook), so we attribute a hook's whole total to a category when every reason sampled from it agrees. This is an inference from the sample to the hook, not an exhaustive classification of every record: an unsampled record in an attributed hook could belong to another category. On that basis 255,568 of the 313,486 seeded `NO-CONTRACT` records are attributed, leaving 57,918 in six hooks whose sampled reasons span categories. The most frequent recorded limitations are an index variable with no static range, a symbolic shape, a non-constant divisor, and a layout the frontend cannot read from the IR node.

Denominator sensitivity. Two categories may contribute records without an additional indexing check: hooks with no indexing structure to audit, and hooks whose check is already represented by another witness. Their exact hook totals bound these contributions by 22,094 and 10,906 records, respectively. Removing the maximum contribution from both categories would raise the seeded campaign's resolved fraction from 74.0% to 76.1%. This calculation measures sensitivity to the counting convention; neither percentage estimates the probability of constructing a witness conditional on a transformation occurring.

Other frontends. MLIR's 34 outcomes resolve to eight encoding limitations and 26 carrying `hunt.py`'s fallback label, which is returned for any reason its specific patterns miss and so cannot separate an absent operation from one in an unsupported form. IREE's `gate2-corpus.json` stores truncated diagnostics. The detailed analysis of `NO-CONTRACT` reasons covers the 19 cases for which a full fail-closed reason was retained, and the XLA summaries record counts rather than per-root reasons.

G Measurement Protocols and Detailed Results

Table A9 gives the per-frontend setup behind the integration costs.

Measurement setup. All integration-cost experiments use the same host: an AMD Ryzen 7 7800X3D (8 cores, 16 threads) running Linux 6.17.0-35-generic, x86_64, glibc

Frontend	Evaluation unit	Total	Resolved	NO-CONTRACT	UNK.	Pass fail.	Unres.
MLIR	Transformation stages	158	116	34	0	8	0
XLA	Fusion output instructions	1,000	879	120	0	0	1[†]
IREE	Audit invocations	6,699	3,821	2,870	0	8	0
TorchInductor	Instrumentation rec.	1,951,123	1,493,342	457,781	0	0	0

Table A8. Applicability counts behind Figure 4. Resolved counts `VALID` and `INVALID` outcomes. `UNK.` counts only checker-issued outcomes, where the checker received a witness and could not decide an obligation exactly; there are none. Recorded separately: production-pass failures, which prevent checking the intended transformation output, and the one XLA case that is unresolved at harness level, matching neither a definite timeout nor a known checker path. Eight seeded TorchInductor compilation failures use a different counting unit and are not included here.

Frontend	Audited revision	Trials	Percentile	Components separated
MLIR	LLVM `57b5017e0102`[†]	20 pairs/config	Nearest sample	None; whole-tool time
IREE	IREE `5fbfea5ac9`[†]	20 pairs/config	Nearest sample	None; whole-tool time
TorchInductor	PyTorch 2.14.0+cu130; Python 3.12.14	15 pairs/program	Linear interp.	Capture; deferred check
XLA	XLA `645c721deeb0`	15 trials/fixture	Linear interp.	Construction; checking

Table A9. Integration-cost setup. *Trials* is the retained measurement count and *Percentile* the within-configuration convention; aggregation across configurations is stated in the text. [†]Not an upstream hash; see the paragraph on audited revisions.

2.39. The NVIDIA RTX PRO 4000 Blackwell SFF Edition GPU is exercised only by the TorchInductor experiment; the other three are CPU-only compiler work. Each experiment ran and discarded one warm-up pair beforehand, or one warm-up trial for XLA. Paired experiments interleave the two arms within each pair. The native runs disable compiler threading, run one job at a time, and discard output; TorchInductor uses a fresh compilation cache per trial.

Audited revisions are not all upstream. Two of the four builds carry one commit from this work on top of a named upstream base, so neither hash resolves in the upstream repository. The MLIR build uses LLVM `57b5017e0102`, which is upstream `31a1c03fe67c` plus one SCF commit clamping normalized loop sizes where they are combined. The IREE build uses `5fbfea5ac9`, which is upstream `a5ca3e2bea` plus one LLVMGPU commit sizing the `mma.sync` LHS fragment by element width. The XLA revision `645c721deeb0` is upstream.

Aggregation. Within each configuration, the native harnesses compute quantile f by sorting the n retained samples and selecting the observed value at zero-based index $\mathrm{round}((n-1)f)$, while TorchInductor and XLA use linear interpolation. For the MLIR and IREE summary ratios, we pool the audit-to-pass ratios from 20 paired trials in each of eight configurations, giving 160 ratios per frontend. We then apply the same sample-selection rule at $f = 0.5$, so the reported median is an observed paired ratio from the pooled trials rather than a median of configuration-level medians.

Excluded configuration. A configuration contributes a ratio only when no trial of its reference run reports a pass failure or a timeout. The IREE `reconcile-translation-info` configuration fails this test. The pass creates a workgroup-identifier operation from IREE's HAL dialect without declaring that dialect among its dependencies, so running it in isolation, as a timing reference does, aborts inside the pass while creating the operation. The audit does not encounter this because its own pass declares the dialect, as would any pipeline that loads it beforehand. The recorded 1.08 s reference duration is therefore dominated by crash handling rather than pass work.

Per-configuration dispersion. Table A10 gives both arms for every native configuration with its spread over the retained trials, so a ratio can be read against what it is a ratio of and against how much the trials varied.

TorchInductor and XLA dispersion. Tables A11 and A12 give the retained spread behind the point estimates.

XLA construction and checking. XLA is the one frontend whose measurements separate the two components. Construction takes under 1 ms for the two-witness XB and XD fixtures, 7.30 ms for XC's 128 witnesses, and 9.68 ms for XA's 512 witnesses. Checker invocation costs 2.2× construction for XC and 6–60× for the other fixtures. The five fixtures audit nested dynamic slices, the GPU dynamic-slice-copy offset path, a fused update-and-roll indexing map, and runtime start expressions.

Obligation latency. The per-obligation timings in the main text come from `timeObligation` in the checker, which wraps a single obligation's decision function in `std::chrono::steady_clock` and emits the elapsed microseconds as a remark. It excludes witness construction, parsing, and process startup. The harness records durations in integer microseconds. Address correspondence has a

Frontend	Configuration	Reference (ms)		Audit (ms)		Median paired ratio
		median	$p95$	median	$p95$	median
MLIR	`sparse-reinterpret-map`	8.75	9.11	8.93	9.95	1.043×
MLIR	`async-parallel-for`	9.37	10.01	9.82	10.23	1.043×
MLIR	`xegpu-wg-to-sg-distribute`	8.47	8.89	9.11	9.50	1.087×
MLIR	`xegpu-sg-to-lane-distribute`	8.41	8.99	9.40	9.84	1.115×
MLIR	`xegpu-blocking`	8.54	9.39	10.03	10.47	1.188×
MLIR	`scf-for-loop-peeling`	8.74	14.74	10.58	20.91	1.236×
MLIR	`affine-loop-fusion`	12.32	12.62	15.53	16.11	1.262×
MLIR	`xegpu-propagate-layout`	11.17	11.64	15.72	16.62	1.410×
IREE	`lane9-decompose-im2col`	11.86	12.34	12.37	13.08	1.036×
IREE	`lane7-fold-affinemin`	10.29	10.94	10.49	10.91	1.040×
IREE	`lane6-gpu-distribute-forall`	10.75	11.31	11.18	11.94	1.041×
IREE	`lane1-gpu-pack-partial-reductions`	11.40	12.02	11.87	12.50	1.046×
IREE	`lane8-collapse-dimensions`	10.77	11.22	11.18	11.92	1.049×
IREE	`lane10-decompose-winograd`	11.57	11.80	12.17	12.58	1.058×
IREE	`lane2-materialize-encoding-index`	14.72	15.44	17.41	17.71	1.187×
IREE	`lane3-vector-distribute`	11.18	11.87	44.60	45.20	3.982×

Table A10. Both arms per native configuration, over the 20 retained paired trials each. The ratio column is the configuration's median paired ratio; the manuscript's headline pools all 160 raw per-trial ratios instead of taking a median of these medians. `reconcile-translation-info` is excluded by the successful-reference criterion and appears in neither column.

	Capture		Deferred	
Program	median	$p95$	median	$p95$
Storage bounds	17.8	27.2	11.36	11.86
Loop splitting	5,512.8	5,658.2	103.39	104.94
Reduction	44.0	50.1	44.84	45.41

Table A11. TorchInductor dispersion in milliseconds over 15 retained pairs, for the programs `i3-storage`, `i5-loop-split`, and `i6-reduction` in row order.

Fixture	median	$p95$
Nested dynamic slice `xa-nested-dynamic-slice-repro`	68.34	71.84
GPU slice-copy offsets `xb-gpu-dus-copy-repro`	8.42	9.88
Fused update-and-roll `xc-dus-dynamic-roll-fusion`	23.70	24.95
Loop-carried start `xd-loop-carried-start`	8.00	8.72
SPMD shard offsets `xd-spmd-shard-offsets`	7.79	8.48

Table A12. XLA total processing in milliseconds over 15 retained trials, with each fixture's artifact identifier beneath its name.

recorded median of 1 μs, which is close to the measurement granularity and should not be interpreted as a precise sub-microsecond timing estimate.

Across the MLIR census, the 95th-percentile latencies are 2 μs for address correspondence over 50 samples, 75 μs for physical bounds over 98 samples, and 253 μs for logical bounds over 381 samples. Write uniqueness and coverage each have 98 samples, with 95th-percentile latencies of 16.5 and 182.7 ms, respectively; coverage has a median of 3.0 ms. Mask completeness has only three samples, whose 95th percentile is 88 μs, so we omit it from the main text because this sample is insufficient to characterize its latency distribution. Appendix H reports a separate backend comparison using a different measurement harness on 696 replayed census queries and 217 queries from retained cases, giving its reported percentiles a different population from the measurements here.

Program	Baseline time to first result	Capture increase	%	Deferred check
`i3-storage`	974.6	17.8	1.8%	11.4
`i5-loop-split`	11,002.6	5,512.8	50.1%	103.4
`i6-reduction`	1,249.6	44.0	3.5%	44.8

Table A13. TorchInductor integration costs in milliseconds, behind Figure 5. Each run pairs an instrumented and a baseline compilation in fresh subprocesses; the capture increase is the median of the 15 paired differences, not the difference between the two arms' medians. The percentage is that median paired difference divided by the baseline median in the same row. Deferred checking is a separate interval, timed at process exit.

Versions. The campaign manifest pins the checker binary `witness-opt` and 380 semantic sources by `sha256`, and records the host and the dirty working paths at freeze time, so the build used for these populations is identified by content rather than by revision alone.

H CHECKING BACKENDS COMPARED

Section 5.5 states that, on identical witness queries, Presburger has lower decision times for bounds and address correspondence, while Z3 is faster for coverage and write uniqueness. This section gives the protocol and results. The comparison times decision procedures on witnesses that frontends have already constructed; it changes no frontend and no verdict reported in the main text.

Protocol. Each query is one selected obligation of one parsed witness, and every backend receives the same parsed witness; translation happens after witness construction. The Presburger backend is a verbatim copy of the checker's obligation code, whose source remains unchanged in the checker. Its construction time is the checker's preparation step and its decision time is the obligation function, the region timed in Section 5.5. Z3 5.3.1 and cvc5 1.4.2 receive the violation query over mathematical integers, with floor division by a positive constant encoded exactly and the remainder encoded as $e - c\lfloor e/c \rfloor$. Queries declare `QF_LIA`, or `LIA` for coverage, which retains its quantifier alternation; the SMT side receives no enumeration, projection, or simplification, and both solvers use default settings. Mask completeness is encoded as the checker decides it, as an execution point $x \in E$ with $F(x) \in S$ and $\neg P(x)$; the logical-bounds direction is a separate obligation in mask mode.

The witness file is parsed once. Each trial forks a fresh child that performs setup, construction, and decision and times each phase; the parent stops the child at a 30 s budget, identically for all backends. Solver libraries are loaded in the parent before forking, one worker runs per physical core of a single host (AMD Ryzen 7 7800X3D, 8 cores and 16 threads, 124 GB, with each core's second thread left idle), and each child has a 16 GB address-space limit. Each query runs seven trials, and we take the per-query median. Timeouts and solver `UNKNOWN` outcomes would be counted separately and never converted into runtimes; none occurred. Parsing and process start-up are excluded for every backend. Creating a solver context is reported separately as setup, with medians of 6.4 ms for Z3, 0.7 ms for cvc5, and none for Presburger, because a long-running checker would pay it once.

Translation check. Before any timing, every query ran once per backend, and wherever the domain was free of symbols and had at most $2 \cdot 10^6$ points, an independent evaluator using the checker's expression semantics enumerated it. Table A14 gives the results. Each counterexample is replayed against its witness; for coverage, the recheck also confirms that no preimage exists. Three scaling rechecks are partial: they are coverage counterexamples at extent 65,536, whose preimage search ($4.3 \cdot 10^9$ points) exceeds the limit, so source-domain membership is confirmed and all backends agree, but the absence of a preimage is not independently rechecked. This check found two harness defects, which were fixed before timing, with the first run retained: a quantifier over zero variables, arising for coverage on a zero-dimensional execution domain, was rejected by both solvers, and the coverage recheck had skipped witnesses with symbols.

Corpus	Queries	Disagree	Brute force	Counterexamples rechecked
Census and retained	913	0	886/886 enumerable	77/77 per backend
Scaling, originals and controls	350	0	314/314 enumerable	147/150 per backend; 3 partial

Table A14. Translation check. All three backends decided every query within the budget.

Corpora. The census corpus replays the 200-case checking-cost census through its own pipeline, stopped before verification, with upstream `mlir-opt` at the witness-opt pin `57b5017e0102`. Its 114 witness-bearing cases carry 381 witnesses and 696 obligation queries; every replayed checker verdict matches the archive, and 696 of the 728 latency samples behind Section 5.5 are reproduced. The other 32 come from two `continuous-tiling` regression-test chunks whose archived syntax the pinned parser rejects. The retained corpus contains 116 distinct modules with 135 witnesses and 217 queries: 13 XLA fixtures, 8 IREE verifier tests, 31 MLIR bug tests replayed through their audit commands, 63 TorchInductor captures, and one further retained module. Its 77 violation queries are the only `INVALID` queries in Experiment 1. The scaling corpus has 32 programs from the census generators, pre-registered before any scaling timing, in four families: domain extent (8 to 65,536), coordinate count (rank 1 to 6, one or two tiling levels), piecewise tails (0 to 4 non-divisible dimensions at rank 4), and division depth (1 to 7 nested floor-division or remainder operations). Each scaling witness also runs as a synthetic erroneous counterpart, its last map result plus one, applied after parsing; these are controls, not compiler bugs.

Experiment 1: identical witnesses. Across the 913 queries in Table A15, Presburger has lower decision times for logical bounds, physical bounds, and address correspondence, with geometric-mean advantages of 4–13× over the SMT backends. Z3 is faster for coverage and write uniqueness, while cvc5's performance depends on the latency statistic. For coverage, cvc5 has a higher median than

		Median decision (ms)			95th-percentile decision (ms)			Ratio to Presb.	
Obligation	Queries	Presburger	Z3	cvc5	Presburger	Z3	cvc5	Z3	cvc5
Logical bounds	393	0.086	0.90	0.50	0.31	1.18	0.86	6.5×	4.1×
Physical bounds	127	0.075	1.03	0.58	0.12	1.35	1.02	13.2×	7.1×
Address correspondence	152	0.016	0.19	0.17	0.12	1.42	0.59	12.1×	8.0×
Mask completeness	3	0.17	0.18	0.14	0.18	0.18	0.14	1.2×	0.9×
Write uniqueness	116	1.72	1.29	1.72	18.0	1.73	3.21	0.76×	0.91×
Coverage	122	1.87	0.76	3.61	75.0	2.58	7.94	0.32×	1.02×
All	913	0.095	0.90	0.56	10.6	1.52	4.43	4.0×	3.4×

Table A15. Decision time by obligation on identical queries (Experiment 1), excluding setup, construction, parsing, and start-up. Each query's time is the median of seven trials; columns give the median and 95th percentile across queries. The ratio is the geometric mean over queries of SMT time divided by Presburger time, so values above one favour Presburger. All backends return the same verdict on every query (836 `VALID`, 77 `INVALID`).

Presburger (3.61 versus 1.87 ms) but a substantially lower 95th percentile (7.9 versus 75.0 ms); for write uniqueness, their medians are equal at 1.72 ms. Coverage and write uniqueness account for Presburger's longest decision times, while the three mask-completeness queries provide too few observations for a meaningful comparison.

Summing costs across all 913 queries without solver setup gives 2.79 s for Presburger, 1.90 s for Z3, and 1.87 s for cvc5. Including setup increases the SMT totals to 7.78 and 2.51 s, respectively, showing how context creation changes the aggregate comparison. We also sum costs over each witness's selected obligations to characterize the backend work required for a complete witness. Without setup, Presburger's median, 95th-percentile, and maximum costs are 0.24, 24.1, and 201.8 ms, respectively. The corresponding values are 2.25, 10.3, and 11.9 ms for Z3, and 1.49, 13.9, and 28.6 ms for cvc5. Presburger thus has the lowest median per-witness cost in this corpus, while both SMT backends have lower costs at the 95th percentile and maximum.

Experiment 2: controlled scaling. The scaling experiments measure backend processing of already constructed witnesses, excluding solver setup and frontend witness construction. Across domain extents from 8 to 65,536, all three backends show little variation in runtime, with Presburger taking approximately 4.9 ms per program.

Increasing the number of execution coordinates has a much larger effect on Presburger coverage. Its decision time rises through 0.6, 2.3, 18.7, 193, 2,219, and 25,360 ms at ranks one through six, respectively, reaching approximately 85% of the 30 s budget at rank six. Z3 remains near 1 ms, while cvc5 reaches 8.4 ms at the highest rank. Presburger's write-uniqueness time also increases, from 0.27 to 48 ms, compared with 2 to 4 ms for the SMT backends. Adding another tiling level changes the measured cost only slightly, while varying the number of non-divisible dimensions at rank four leaves Presburger coverage near 190 ms. Within these families, coverage cost is therefore more sensitive to coordinate count than to the tested tail predicates.

Increasing division depth produces a different pattern when proving address correspondence. Across the tested depths, Presburger's decision time grows from 0.01 to 27 ms, compared with 0.08 to 136 ms for cvc5 and 0.16 to 1,296 ms for Z3. On the erroneous counterparts, however, both SMT backends find a counterexample in approximately 1 ms at every depth, while Presburger takes 28 ms at the greatest depth.

Limitations. Fork-per-trial execution charges each query a fresh solver context; a persistent-context deployment would sit closer to the totals without setup. Each solver is used in one version with its default configuration and no tactic tuning. The scaling families come from two generators, tiled `linalg.copy` and `memref` collapse folding, and other witness families may scale differently. No decision reached the budget, so the results characterize latency only.

I Upstream Reporting Status

Table A16 records the filed-issue population. It counts separately from the deduplicated bugs of the main text: one bug may produce several reports, and a report may cover a manifestation consolidated under another root cause.

Compiler	Filed	Linked fix	Not reproducing
LLVM/MLIR/NVPTX	26	9	0
XLA	7	2	1
IREE	3	2	0
Total	36	13	1

Table A16. Upstream status checked on 18 October 2026. Thirteen reports are linked to identified merged fixes. One additional XLA report no longer reproduces on the tested prerelease build, but its resolution has not been attributed to a commit. Issue closure is recorded separately from fix status: four of the linked fixes are merged while their issues remain open. The artifact records the mapping between manifestations, root causes, and filed issues. Previously reported TorchInductor bugs are counted separately.