# Verified Equivalence Checking for Vector Assembly in Fiat Cryptography

by

Andrew M. Spears

S.B. Computer Science and Engineering and Mathematics, Massachusetts Institute of Technology, 2025.

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026



Authored by: Andrew M. Spears
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Adam Chlipala
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Verified Equivalence Checking for Vector Assembly in Fiat Cryptography

by

Andrew M. Spears

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## ABSTRACT

Fiat Cryptography is a verified compiler that synthesizes finite-field arithmetic primitives. It includes an equivalence checker that verifies assembly programs against high-level functional specifications. The highest-performance cryptographic implementations rely on parallel computation through vectorized assembly instructions, but the equivalence checker supported only scalar x86-64 instructions prior to this work.

This thesis extends the equivalence checker to support the Advanced Vector Extensions (AVX2) x86-64 instruction set. This contribution involved a full refactor of register and memory infrastructure, symbolic-execution support for 19 AVX2 instructions, and a scalable hierarchical proof strategy that enables one-line correctness proofs for most instructions. We introduce specifications for batched field operations, enabling verification of programs that exploit data-level parallelism across field elements.

Finally, we demonstrate end-to-end equivalence checking on a suite of hand-written AVX2 assembly programs implementing field arithmetic for Curve25519, including both limb-parallel and batched programs. To our knowledge, this work presents the first formally verified equivalence checking of raw vectorized assembly against a functional specification.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

# Acknowledgments

Thank you to Adam Chlipala, the PLV lab group, Tim Spears, Lyndi Spears, Bronwyn Spears, Marco Andrade, Sasha Sherstnev, Brook Wang, and all other friends who made this work possible.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Cryptographic software depends on a small set of arithmetic primitives that are invoked thousands of times in basic protocols. These operations are mathematically simple (finite-field addition, multiplication, and reduction), but their implementations are performance-critical. For this reason, they are often hand-optimized in assembly to exploit machine-specific features, a process that is both error-prone and difficult to verify.

Fiat Cryptography [1][1] and CryptOpt [2] are recent projects that attempt to solve this problem by synthesizing low-level programs and guaranteeing correctness through formal verification. The resulting code outperforms standard compilers, and certain synthesized primitives are used in production by all major web browsers and many TLS implementations.

However, both Fiat Cryptography and CryptOpt are currently limited to scalar x86-64 instructions. Vectorized instructions perform operations in parallel and are increasingly relied upon in the highest-performance implementations, but these synthesis pipelines are incompatible with vectorized instruction sets, putting a hard ceiling on their performance capabilities.

Fiat Cryptography includes an equivalence checker that verifies assembly programs against high-level specifications. This thesis extends the equivalence checker to support AVX2 vector instructions. Our contributions include:

1. Refactoring existing proof infrastructure and automation to be more modular, enabling scalable extension to new instruction families

2. Refactoring the register and memory infrastructure to support 128-bit and 256-bit operands

3. Implementing symbolic execution for 19 AVX2 instructions, with modular abstractions designed to scale to additional instructions

4. Developing a hierarchical proof strategy with correspondence lemmas and proof automation for relating symbolic and concrete semantics

5. Adding rewrite rules that enable the equivalence checker to simplify lane-gathering and decomposition patterns

[1]Source code for Fiat Cryptography and the work described in this thesis is available at https://github.com/mit-plv/fiat-crypto.

6. Demonstrating end-to-end equivalence checking on a suite of vectorized field-arithmetic programs

# Chapter 2

# Background

## 2.1 Cryptography and Limb Representation

Elliptic-curve cryptography works with finite fields, generally integers modulo a large prime. The field elements involved are typically on the order of hundreds of bits, far too large to fit in a single machine word. Instead, they are represented using arrays of 64-bit integers called *limbs*.

One commonly used field is integers modulo $2^{255} - 19$, whose largest element occupies 255 bits. These 255 bits could mathematically fit in 4 64-bit limbs, but in practice, representation schemes typically use 5 or more limbs. For example, the *unsaturated Solinas* [1] representation uses 5 64-bit limbs with headroom in each limb so that many operations can be applied independently to each limb before carrying between them.

These engineering choices have two important consequences:

- Addition and subtraction can be parallelized across limbs (although multiplication and carry propagation cannot).
- Implementing and optimizing even basic field-arithmetic operations can be extremely complex, since each operation involves many registers and ALU instructions.

## 2.2 AVX2 and SIMD Instructions

SIMD (Single Instruction, Multiple Data) is a class of parallel processing in which a single instruction operates on multiple data elements simultaneously. On x86-64, Intel's AVX2 (Advanced Vector Extensions 2) instruction set provides SIMD capabilities through vector registers (split into *lanes* that are used to hold multiple values) and vector instructions.

AVX2 introduces two tiers of vector registers:

- **XMM registers** (`xmm0`–`xmm15`): 128 bits wide, holding two 64-bit lanes or four 32-bit lanes.
- **YMM registers** (`ymm0`–`ymm15`): 256 bits wide, holding four 64-bit lanes or eight 32-bit lanes. Each YMM register's lower 128 bits alias the corresponding XMM register.

Most AVX2 arithmetic instructions operate *lane-wise*, meaning they apply the same operation independently to each lane of the input registers.

For cryptography, SIMD parallelism can be utilized in two ways:

- *Limb-parallel* programs use vector registers to process multiple limbs of a single field element simultaneously — for example, packing limbs 0 and 1 into one XMM register and adding them in a single `vpaddq`. These programs can be checked against existing specifications in Fiat Cryptography, since they are optimizations of scalar field operations.
- *Batched* programs instead process the same limb across multiple independent field elements — for example, packing limb 0 of four different elements into one YMM register, so a single `vpaddq` performs four independent additions at that limb position. These specifications did not previously exist in Fiat Cryptography.

The AVX2 instructions used in this work include data movement (`vmovdqu`, `vmovq`, `vpbroadcastq`), lane-wise arithmetic (`vpaddq`, `vpsubq`, `vpmuludq`), bitwise operations (`vpandq`, `vpxorq`), shifts (`vpsllq`, `vpsrlq`), and lane manipulation (`vpblendd`, `vpunpcklqdq`, `vpextrq`, `vextracti128`).

## 2.3 Rocq Theorem Prover

Most software is checked for correctness by testing on known input-output pairs. Formal verification instead establishes correctness through machine-checked mathematical proofs that software matches a precise logical specification.

Fiat Cryptography and our additions are implemented in Rocq (formerly Coq), an interactive proof assistant enabling functional code along with compile-time-verified proofs of logical propositions about runtime values.

Throughout this thesis we show Rocq code snippets: `Definition` and `Fixpoint` introduce functions, `Lemma` states a theorem, and `Proof ... Qed` provides its proof.

## 2.4 Fiat Cryptography

Fiat Cryptography [1] is a verified compiler that synthesizes correct-by-construction implementations of finite-field arithmetic. At a high level, its synthesis pipeline works as follows:

1. A mathematical specification of each field operation (e.g., modular addition) is defined by hand as a Rocq function.
2. The specification is *reified* — converted from a Rocq function into an explicit syntax tree in PHOAS (Parametric Higher-Order Abstract Syntax) [3] that can be inspected and transformed programmatically.

3. The PHOAS expression is passed through `BoundsPipeline`, which performs partial evaluation, bounds analysis, and arithmetic optimizations, producing an optimized PHOAS expression.

4. The optimized expression is stringified into a C program (or a program in another target language).

Each stage is accompanied by a machine-checked proof that it preserves the semantics of the original specification.

## 2.5 CryptOpt

CryptOpt [2] extends the approach of a formally verified synthesis pipeline with two significant features: a stochastic optimizer and an equivalence checker.

Starting from the optimized PHOAS expression, CryptOpt generates an initial assembly implementation, then repeatedly mutates it by reordering instructions, substituting equivalent instruction templates, and reallocating registers. Mutations that improve execution time on the target CPU are kept, while those that decrease performance are discarded. The result is a highly performant but not certifiably correct program.

## 2.6 The Equivalence Checker

The equivalence checker exists to verify the untrusted assembly programs output by CryptOpt[1]. At a high level, it simulates executing a program, representing each newly computed value as an expression over inputs and relating them in a data structure called an *e-graph*[2]. We call this process *symbolic execution* and explain it in detail in chapter 3.

An equivalence checker between assembly programs and high-level functional programs would be straightforward if both kinds of programs were self-contained mappings from inputs to outputs. In reality, x86-64 assembly is inherently stateful — each instruction reads and writes register and memory values, so the behavior of each instruction depends on prior ones.

The equivalence checker simulates state with a simple mapping called *symbolic state* that maps each register and memory address to a previously created logical variable in the e-graph.

When each instruction is symbolically executed, the equivalence checker runs a set of *rewrite rules* that normalize each new expression through algebraic simplification before inserting them into the e-graph. These rewrites are essential for recognizing equivalence between two programs. Without this step, checking equivalence would be computationally expensive or undecidable.

By the end of symbolic execution, the logical variables associated with output registers and memory locations represent the program's outputs as expressions over the inputs, and equivalence between the program and specification reduces to checking that outputs map to the same logical variables across the two programs.

[1] Or another optimization process, including handwritten code. The equivalence checker resides in the Fiat-Crypto repository.

[2] A simplified version of an e-graph without node merging.

To execute an assembly program symbolically, the equivalence checker needs a reference for how to execute each instruction, but also assurance that this symbolic execution is faithful to actual x86-64 semantics. It achieves both by having two separate descriptions of each instruction: the semantic description, which encodes exactly what the instruction does on the machine; and the symbolic description, which describes how the instruction affects the DAG and symbolic state. A layer in between then proves equivalence of the two descriptions.

This split achieves three goals:

1. The trusted specification of each instruction (concrete semantics) is simple enough for a human to audit against the ISA manual.

2. The symbolic layer is free to use efficient abstractions (decomposing instructions to a set of atomic operations) tailored to the DAG representation.

3. The proof layer between the two ensures soundness without trusting that the symbolic description is faithful to x86 behavior.

## 2.7 Related Work

Several projects address verified cryptographic implementations at different levels of abstraction.

**Jasmin** [4] is a language and compiler for high-assurance cryptography, verified in Rocq. Programmers write in a C-like language with assembly-level control over instruction selection and register allocation, and the compiler produces x86-64 assembly with formally verified passes. Unlike our work, Jasmin verifies its own compiler's output rather than externally produced assembly, and it currently targets only scalar x86-64 instructions.

**HACL*** [5] is a verified cryptography library written in F*. It provides pure C implementations of cryptographic primitives that are formally verified for memory safety, functional correctness, and secret independence. The resulting code is portable but not optimized for platform-specific assembly.

**Vale and EverCrypt.** Vale [6] is a tool for writing and verifying assembly code in F*. Cryptographic routines are written directly in Vale's DSL, and F* proves similar properties as in HACL*. EverCrypt [7] wraps Vale and HACL* behind a multiplexing API that selects the best implementation at runtime; it is deployed in Firefox, the Linux kernel, and other production systems. However, Vale requires cryptographic code to be written in its DSL, whereas our approach verifies assembly written in standard Intel syntax.

**HACL×N** [8] is the closest prior work on verified SIMD cryptography. It extends HACL* with generic SIMD code in F* parameterized over vector width, targeting ARM Neon, AVX, AVX2, and AVX-512. However, HACL×N operates at a higher level of abstraction: SIMD parallelism is expressed through F*'s type system with explicit vector types, avoiding reasoning about x86 semantics entirely. Assembly programs can be compiled from this point but they will not be provably correct.

To our knowledge, this thesis presents the first formally verified checking of raw AVX2 assembly against a functional specification. Prior work either avoids assembly-level reasoning by using a DSL or does not handle SIMD instructions.

# Chapter 3

# System Architecture

## 3.1 Symbolic Execution

We mentioned in section 2.6 that the equivalence checker uses a simplified version of an e-graph to simulate executing a program. Specifically, it uses a hash-consing DAG, where each node stores an operator and a list of child indices. To handle state, a *symbolic state* maps each register and memory address to a DAG node.

When a new instruction is symbolically executed, the equivalence checker performs the following steps:

1. Look up the DAG nodes for each source operand in the symbolic state.
2. Build an expression tree representing the instruction's effect, with children pointing to the operand nodes.
3. Run rewrite rules to normalize the expression (e.g., constant folding, canonicalizing commutative operand order, algebraic simplifications).
4. Insert the normalized expression into the DAG via hash-consing: if a structurally identical node already exists, return its index; otherwise, create a new node.
5. Update the symbolic state so the destination register points to the resulting DAG index.

This design means two expressions are proven equal exactly when they resolve to the same DAG index, without needing the added complexity of SAT solving or union-find.

In Rocq, these stateful operations are threaded through a symbolic state monad `M`, where the bind notation ← sequences steps that read or update the state. For example, the symbolic execution of `mov dst, src` is:

```
1   | mov, [dst; src] =>
2     v <- GetOperand src;
3     SetOperand dst v
```

`GetOperand` looks up the source's current DAG index and sets $v$ to the resulting value, while threading the state through to the next operation; `SetOperand` edits the destination register and returns the resulting state.

## 3.2 Correspondence

A sound equivalence checker must be faithful to x86-64 semantics when performing symbolic execution. Correspondence between the semantic and symbolic layers is captured by a relation $R\ s\ m$, which asserts that the symbolic state $s$ (the DAG and register/memory mappings into it) is a faithful abstraction of the machine state $m$ (simulated concrete register and memory values).

Specifically, if a DAG index $i$ is mapped to a register or memory slot in the symbolic state, then evaluating the DAG expression rooted at $i$ must produce the concrete value stored in that register or slot in $m$.

Each correspondence lemma then takes the form: given $R\ s\ m$ before an instruction and a successful symbolic execution producing state $s'$, conclude that:

- There exists a machine state $m'$ that is the result of concretely executing the same instruction on $m$.
- $R\ s'\ m'$ holds (the simulation relation is preserved).
- $s \prec s'$: the DAG only grew, so all previously valid indices still evaluate to the same values.

For example, this lemma proves correspondence is preserved by setting a register value:

```
1 Lemma R_SetReg {opts : symbolic_options_computed_opt} {descr:description}
2   s s' m (HR : R s m) r i _tt
3   (H : Symbolic.SetReg r i s = Success (_tt, s')) (* updated symbolic state *)
4   v (Hv : eval s i v) (* index i in state s evaluates to value v *)
5   : let m' := (update_reg_with m (fun rs => set_reg rs r v)) in (* updated machine state *)
6    R s' m' /\ s :< s'.
```

# Chapter 4

# Infrastructure for Wide Operands

Fiat Cryptography was originally designed around scalar x86-64 registers and therefore had no notion of registers or memory accesses larger than 64 bits. AVX includes 128-bit `xmm` and 256-bit `ymm` registers and instructions that load and store 128 or 256 bits at a time. Before implementing any vector-instruction semantics, we needed to refactor the register and memory infrastructure to accommodate these wider operands.

## 4.1 Register Types

This refactoring process began with the register type. Scalar and vector registers are fundamentally different in terms of how they interact with the rest of the codebase, in a few important ways:

- Only scalar registers can form memory-address operands.
- Only scalar registers can participate in flag-setting arithmetic and carry chains.
- Only scalar registers can appear in calling conventions.
- Scalar registers' alias hierarchy is four levels deep (`al`/`ax`/`eax`/`rax`).
- Vector registers carry opaque packed-lane data.
- Vector registers alias only two levels deep (`xmm`/`ymm`).

We made the two register kinds distinguishable by type, with a unifying type `REG = SReg SREG | VReg VREG`.

```
1 (* Unified register type: scalar or vector *)
2 Inductive REG :=
3 | SReg (r : SREG)
4 | VReg (v : VREG)
5 .
```

Downstream functions that need to distinguish between the two can now do so directly by type. For example, a memory location needs to accept only scalar registers for the base address:

```
Record MEM := {
  mem_bits_access_size : option AccessSize ;
  mem_base_reg : option SREG ;
  mem_scale_reg : option (Z * SREG) ;
  mem_base_label : option string ;
  mem_offset : option Z ;
  rip_relative : rip_relative_kind
}.
```

## 4.2 Register Read/Writes

Many existing functions for interacting with register values in Fiat Cryptography were specialized to handling 64-bit values. For example, the SetReg function uses two branches for 64-bit writes and smaller register writes because an 8- or 32-bit write needs to combine the new value with the existing value, whereas a 64-bit write can overwrite the old value, meaning the old value can be completely forgotten (or be unspecified to begin with), simplifying the computation graph in many cases.

```
(* Old *)
Definition SetReg {opts : symbolic_options_computed_opt} {descr:description}
  r (v : idx) : M unit :=
  let '(rn, lo, sz) := index_and_shift_and_bitcount_of_reg r in
  if N.eqb sz 64
  then v <- App (slice 0 64, [v]);
       SetReg64 rn v (* works even if old value is unspecified *)
  else old <- GetReg64 rn;
       v <- App ((set_slice lo sz), [old; v]);
       SetReg64 rn v.
```

However, when writing to a 128-bit-wide xmm register, even a 64-bit write needs to set a slice of the old value, so this function should check more generally if the write matches the largest size of this register's aliasing hierarchy:

```
(* New *)
Definition SetReg {opts : symbolic_options_computed_opt} {descr:description}
  r (v : idx) : M unit :=
  let '(_, lo, sz) := index_and_shift_and_bitcount_of_reg r in
  (* check size against widest register in hierarchy *)
  if (N.eqb lo 0) && (N.eqb sz (widest_reg_size_of r))
  then SetRegOverwrite r v (* works even if old value is unspecified *)
  else SetRegSlice r v.
```

This kind of refactor requires new lemma structure as well. In this case, we introduced two new lemmas:

```
(* Overwrite case: r is the widest register in its hierarchy. *)
Lemma R_SetRegOverwrite {opts : symbolic_options_computed_opt} {descr:description}
  s m (HR : R s m) r i _tt s'
  (Hoffset : reg_offset r = 0%N)
  (Hwidest : reg_size r = widest_reg_size_of r)
  (H : Symbolic.SetRegOverwrite r i s = Success (_tt, s'))
  v (Hv : eval s i v)
  : let m' := (update_reg_with m (fun rs => set_reg rs r v)) in
    R s' m' /\ s :< s'.
```

```
(* Slice case: r is narrower than the widest register in its hierarchy.
   No precondition on offset/size: SetRegSlice always works correctly. *)
Lemma R_SetRegSlice {opts : symbolic_options_computed_opt} {descr:description}
  s m (HR : R s m) r i _tt s'
  (H : Symbolic.SetRegSlice r i s = Success (_tt, s'))
  v (Hv : eval s i v)
  : let m' := (update_reg_with m (fun rs => set_reg rs r v)) in
    R s' m' /\ s :< s'.
```

## 4.3 Memory Model

Like registers, memory operations were originally built around 64-bit values. Memory in Fiat Cryptography is represented in two ways. In semantics, memory state is a mapping from 64-bit addresses to bytes:

```
(* semantic *)
Definition mem_state := (SortedListWord.map (Naive.word 64) Byte.byte).
```

This map represents what is actually stored in the machine. On the other hand, we have a representation that is referenced when memory operations are added to the symbolic execution graph. This representation only needs to store values in the abstract to keep track of how each store operation influences later load operations, so it uses a mapping between symbolic-execution nodes:

```
(* symbolic *)
Definition mem_state := list (idx * idx).
```

where both keys and values are implicitly 64-bits. This consistency is important because there is no overflow from one memory write to the next address — writing to address `a` cannot influence address `a+8` in this model, so the values need to be a consistent size. One could have used bytes like the semantic memory model, but symbolic execution is far simpler when

everything uses 64 bits, the most common size. Thus, two operations exist for this purpose:

```
Definition Load64 (a : idx) : M idx :=
  some_or (load a) (error.load a).
```

```
Definition Store64 (a v : idx) : M unit :=
  ms <- some_or (store a v) (error.store a v);
  fun s => Success (tt, update_mem_with s (fun _ => ms)).
```

Then a single `Load` function handles the sizes of values involved:

```
(* Old *)
Definition Load {opts : symbolic_options_computed_opt} {descr:description}
  {s : OperationSize} {sa : AddressSize} (a : MEM) : M idx :=
  if negb (orb (Syntax.operand_size a s =? 8 )( Syntax.operand_size a s =? 64))%N
  then err (error.unsupported_memory_access_size (Syntax.operand_size a s)) else
  addr <- Address a;
  v <- Load64 addr;
  App ((slice 0 (Syntax.operand_size a s)), [v]).
```

Storing is handled similarly.

## 4.4 Multi-Word Loads and Stores

Some aspects of the memory model should stay fixed. Memory addresses should always be 64 bits, and it makes sense to keep these data structures. But we often want to store or load chunks of more than 64 bits when working with AVX registers. For example, `vmovdqu ymm0, [rbx]` loads 32 bytes of memory into the `ymm0` register.

To address this issue, we need to change the existing Load and Store functions to allow larger memory reads without altering the 64-bit value memory model. The simplest approach is to break larger operations into 64-bit chunks:

```
(* Load (n * 64) bits starting at addr, as a single idx.
   Produces n sequential Load64s at addr, addr+8, ..., addr+8*(n-1),
   combined via set_slice into one (n*64)-bit value. *)
Fixpoint Load_of_idx {opts : symbolic_options_computed_opt} {descr:description}
  {sa : AddressSize} (n : nat) (addr : idx) : M idx :=
  match n with
  | O      => App (const 0, nil)
  | S n'   => prev   <- Load_of_idx n' addr;
              offset <- App (const (8 * Z.of_nat n'), nil);
              addr_k <- App (add sa, [addr; offset]);
              chunk  <- Load64 addr_k;
              App (set_slice (64 * N.of_nat n') 64, [prev; chunk])
  end.
```

```
(* New *)
Definition Load {opts : symbolic_options_computed_opt} {descr:description}
  {s : OperationSize} {sa : AddressSize} (a : MEM) : M idx :=
  let sz := Syntax.operand_size a s in
  addr <- Address a;
  if ((sz =? 8) || (sz =? 64))%N%bool then
    v <- Load64 addr;
    App ((slice 0 sz), [v])
  else if (sz =? 128)%N then
    Load128_of_idx addr (* wrapper around Load_of_idx *)
  else if (sz =? 256)%N then
    Load256_of_idx addr (* wrapper around Load_of_idx *)
  else err (error.unsupported_memory_access_size sz).
```

This approach retains the simplicity of 64-bit loads (notice the first branch calls Load64 directly). We keep the memory model unchanged, but larger memory operations pay the price in extra operations: address arithmetic, slicing, and loading multiple chunks.

To prove correctness of this new structure, we need to prove that the `Load_of_idx` function (affects symbolic state) is equivalent to loading the appropriate number of 8-byte chunks with `get_mem` (affects machine state). We use the following lemma:

```
(* General (n*64)-bit load. Load128_R / Load256_R are corollaries below. *)
Lemma LoadN_R {opts : symbolic_options_computed_opt} {descr:description}
  {sa : AddressSize}
  (Hsa : sa = 64%N)
  n s m (HR : R s m) (addr : idx)
  va (Ha : eval s addr va)
  i s' (H : Load_of_idx n addr s = Success (i, s'))
  : R s' m /\ s :< s' /\
    exists v, eval s' i v /\
              get_mem m va (8 * n) = Some v /\
              v = Z.land v (Z.ones (64 * Z.of_nat n)).
```

which admits a proof by induction on $n$.

# Chapter 5

# Symbolic Execution for Vector Instructions

## 5.1 Two Approaches

When an instruction is symbolically executed, we need a way of encoding/interpreting what computation is actually being done. There are two principled ways to encode the execution of any (nontrivial) vector instruction, each of which abstracts details to one place or another. The first way is to include a single atomic operation in the execution graph, and the second is to break the instruction down into a composition of simpler operations as soon as it is symbolically executed.

We initially pursued the atomic-op approach, described in section 5.2. After running into scalability problems with the rewrite rules required to make it work, we abandoned it in favor of the decomposition approach described in section 5.3.

## 5.2 Initial Attempt: Atomic Vector Ops

This approach keeps symbolic execution simple by emitting a single atomic `op` for each vector instruction and abstracting details away. We need to define these new atomic operations, for example `vadd` and `vsub`:

```
1 Variant op := ... | vadd (lane_width num_lanes : N) | vsub (lane_width num_lanes : N).
```

Symbolic execution of a vector instruction becomes trivial by getting the source values,

applying the corresponding operation, and writing to the destination:

```
Definition SymexNormalInstruction {opts : symbolic_options_computed_opt} {descr:description}
    (instr : NormalInstruction) : M unit :=
    match instr.(Syntax.op), instr.(args) with
    ...
    | vpaddq, [dst; src1; src2] => (* packed add of quadwords *)
        SymbolicVector.SymexVectorBinOp dst src1 src2 (add 64) 4 64
    ...
```

This design has two important implications. First, we must extend `interp_op` with a case for `vadd`, defining its concrete semantics in terms of machine state. Second (more importantly), the burden of equivalence proofs shifts onto the rewrite system.

We were able to make this approach work for simple programs but only by defining many ad-hoc, complex rewrite rules, which we explain in section 5.4. Based on this initial attempt, we decided that this approach was less principled, more complicated, and less scalable than the alternative.

## 5.3 Adopted Approach: Lane Decomposition in Symbolic Execution

We therefore adopted the second approach: breaking vector instructions into their component operations immediately in symbolic execution. Most vector instructions follow the same pattern: slice lane values from both source registers (which we model as simply larger scalar registers), perform a binary operation, write to lanes of the destination register. For example, the `vpaddq` instruction can be expressed as a composition of simpler operations: `slice`, `add`, and `set_slice` to combine results back into a vector.

This pattern can be captured by a few helper functions and conveniently applies identically on the semantics side, both diagrammed in Figure 5.1. Helper functions handle slicing lanes, while the `vector_binop_aux` applies a binary operation to each lane and recursively builds up the result value.

```
Fixpoint vector_binop_aux {opts : symbolic_options_computed_opt} {descr : description}
    (v1 v2 : idx) (lane_op : op) (lane_idx : nat) (num_remaining : nat)
    (lane_width : Z) (acc : idx) : M idx :=
    match num_remaining with
    | O => ret acc
    | S n =>
        lane_val <- make_lane v1 v2 lane_op lane_idx lane_width;
        new_acc <- App (
            set_slice (N.of_nat lane_idx * Z.to_N lane_width) (Z.to_N lane_width),
            [acc; lane_val]);
        vector_binop_aux v1 v2 lane_op (S lane_idx) n lane_width new_acc
    end.
```

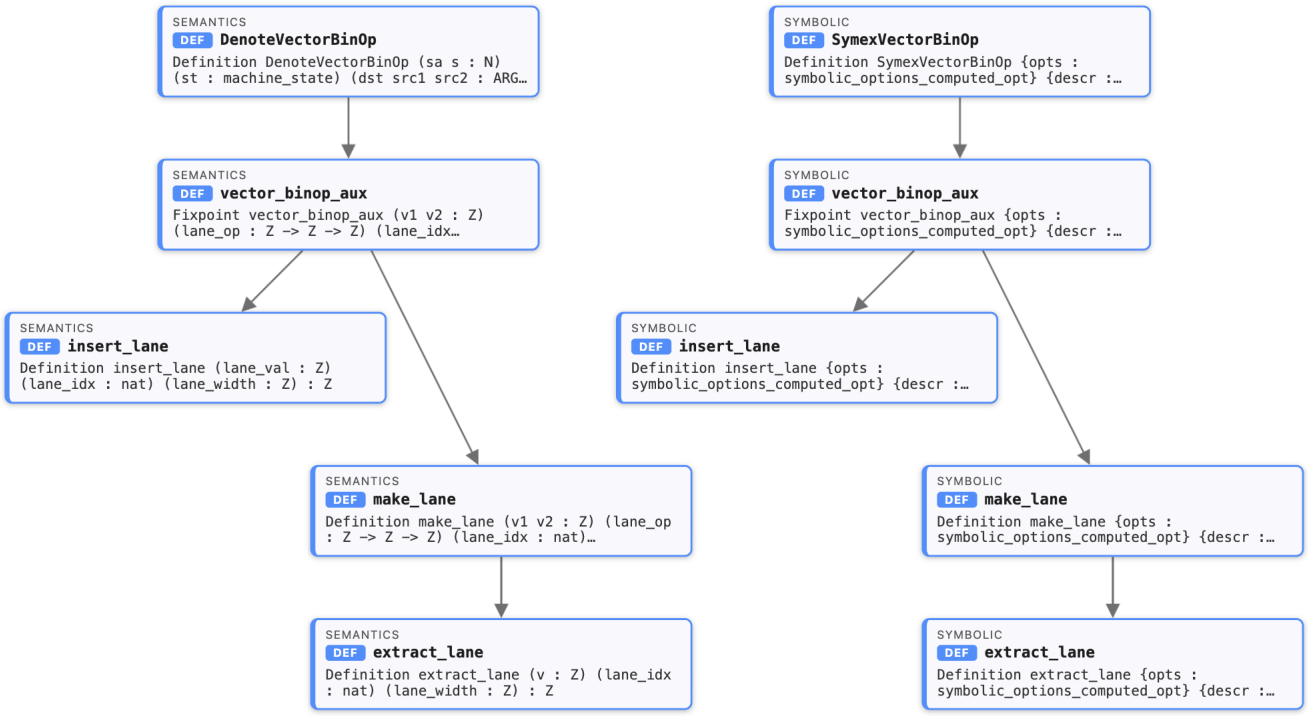


Figure 5.1: Dependency graph of functions in Symbolic.v and Semantics.v to implement vector symbolic execution and semantics.

This foundation makes the symbolic execution of most instructions simple to implement. For example, `vpaddq` can be implemented as follows:

```
Definition SymexNormalInstruction {opts : symbolic_options_computed_opt} {descr:description}
    (instr : NormalInstruction) : M unit :=
    match instr.(Syntax.op), instr.(args) with
    ...
    | vpaddq, [dst; src1; src2] => (* packed add of quadwords *)
        let num_lanes := (op_size / 64)%N in
        v1 <- GetOperand src1;
        v2 <- GetOperand src2;
        result <- App ((vadd 64 num_lanes), [v1; v2]);
        SetOperand dst result
    ...
```

This pattern generalizes beyond binary operations. Instructions like `vpbroadcastq` (replicate one lane to all lanes), `vpblendd` (select lanes from two sources by bitmask), and `vpmuludq` (multiply lower 32 bits of each 64-bit lane) all admit similar decompositions: slice the relevant parts of each source, apply a per-lane operation, and write results back via `set_slice` in some order. The differences lie only in how many sources are read, what operation is applied, and whether extra slicing is needed within each lane. We describe the

proof strategy for these variations in chapter 6.

## 5.4 Rewrite Rules

For the equivalence checker to succeed, a set of rewrite rules must reduce semantically equivalent computations to the same form. We first consider the rewrite rules needed for the atomic-vector-operation approach from section 5.2. If one program uses `vpaddq` (emitting a single `vadd 64 4` node) and an equivalent program performs four scalar 64-bit additions, the rewrite system must contain rules that bridge these two representations, most likely by expanding `vadd` into per-lane scalar additions, or by recognizing the per-lane pattern and folding it into a `vadd`.

Because rewrite rules pattern-match on specific compositions of operations in the DAG, the system must account for unexpected equivalent programs that use our new vector ops at arbitrary depths. This task becomes especially difficult when vector ops and slice operations compose: for example, consider the composition `slice(vadd(set_slice(YMM, v1), set_slice(YMM, v2)))`, which packs two scalars into YMM registers, adds the YMMs lane-wise, then extracts a single lane. Recognizing that this expression is equivalent to a plain scalar addition `v1 + v2` requires the rewrite system to look three layers deep past the outermost slice.

We initially tried this approach with rules about slicing into the result of vector operations, for example:

```
(* Decompose a slice of a vector add into a scalar add of slices. *)
Definition slice_vadd (d : dag) :=
  fun e => match e with
    ExprApp (slice lo lw, [ExprApp (vadd lw' nl, [v1; v2])]) =>
      if N.eqb lw lw' && N.eqb (lo mod lw)%N 0%N && N.leb (lo + lw) (lw' * nl) && N.ltb 0 lw
      then ExprApp (add lw, [coalescing_slice lo lw v1; coalescing_slice lo lw v2])
      else e | _ => e end%bool%N.
```

Notice the use of the `coalescing_slice` helper function:

```
(* If v = slice lo2 s2 [e'] and lo+lw <= s2, produce slice (lo2+lo) lw [e'] instead. *)
Definition coalescing_slice (lo lw : N) (v : expr) : expr :=
  match v with
  | ExprApp (slice lo2 s2, [e']) =>
      if N.leb (lo + lw) s2 then ExprApp (slice (lo2 + lo) lw, [e'])
      else ExprApp (slice lo lw, [v])
  | _ => ExprApp (slice lo lw, [v])
  end%N.
```

which is needed to handle nested slices coming out of the `slice_vadd` rule.

In fact, we ended up adding this rule as well based on empirical testing:

```
(* Recursively normalize (slice lo sz) over nested slice/set_slice towers,
peeling three kinds of layers at once:
- nested slice lo2 s2: combine offsets to (lo2+lo, sz, e')
- disjoint set_slice lo2 s2: descend into base
- containing set_slice lo2 s2: descend into val with adjusted lo
Needed for gather patterns (vmovq -> vpunpcklqdq -> vinserti128) that
interleave slice and set_slice layers *)
Fixpoint slice_tower_normalize (lo sz : N) (inner : expr) (fuel : nat) {struct fuel} : expr :=
  match fuel with
  | O => ExprApp (slice lo sz, [inner])
  | S fuel' =>
    match inner with
    | ExprApp (slice lo2 s2, [e']) =>
        if N.leb (lo + sz) s2
        then slice_tower_normalize (lo2 + lo) sz e' fuel'
        else ExprApp (slice lo sz, [inner])
    | ExprApp (set_slice lo2 s2, [base; val]) =>
        if (N.leb (lo + sz) lo2 || N.leb (lo2 + s2) lo)%bool
        then slice_tower_normalize lo sz base fuel'
        else if (N.leb lo2 lo && N.leb (lo + sz) (lo2 + s2))%bool
        then slice_tower_normalize (lo - lo2) sz val fuel'
        else ExprApp (slice lo sz, [inner])
    | _ => ExprApp (slice lo sz, [inner])
    end
  end%N.
```

We were able to verify simple vector programs with these rules, but each new instruction pattern required additional ad-hoc rules, confirming that this approach would not scale[1].

Since we did not adopt the atomic-vector-op approach, the `slice_vadd` rule was abandoned. The `coalescing_slice` and `slice_tower_normalize` rules were kept, since they are sound simplifications that can only help.

The lane-decomposition approach that we ended up using still requires rewrite rules. Writing a lane value into a vector register produces a `set_slice` node, and reading it back produces a `slice` node wrapping the previous write. Across multiple instructions, these layers compound, and the equivalence checker must simplify them away so that both programs' output indices in the DAG are identical.

The crucial difference from the atomic-op approach is that these rules only need to handle the immediate `slice`/`set_slice` layers introduced by each instruction, rather than recursing into the DAG to unfold earlier vector operations. We added three rewrite rules:

`slice_slice` collapses nested slices when the inner slice is wider. This rule fires when a

[1] Another approach is to convert to full e-graphs with node merging, which would remove the need for these rules entirely.

lane read follows a register read that already sliced from a wider register.

```
Definition slice_slice (d : dag) :=
  fun e => match e with
    ExprApp (slice lo1 s1, [ExprApp (slice lo2 s2, [e'])]) =>
      if N.leb (lo1 + s1) s2
      then ExprApp (slice (lo2 + lo1) s1, [e'])
      else e | _ => e end%N.
```

`slice_set_slice` extracts a value from a write when the slice range falls within the write range. Crucially, this rule recognizes that reading a lane of a vector register immediately after writing to that lane should return the written value.

```
Definition slice_set_slice (d : dag) :=
  fun e => match e with
    ExprApp (slice lo1 s1, [ExprApp (set_slice lo2 s2, [_; e'])]) =>
      if N.leb lo2 lo1 && N.leb (lo1 + s1) (lo2 + s2)
      then ExprApp (slice (lo1 - lo2) s1, [e'])
      else e | _ => e end%bool%N.
```

`slice_set_slice_disjoint` skips a write when the ranges do not overlap. Every write to a 4-lane YMM register produces a chain of three nested `set_slice` nodes (each writing a specific lane), but when reading lane 0, the later writes to lanes 1, 2, and 3 are irrelevant. This rule simplifies this situation using a recursive helper function `peel_disjoint_set_slices`.

```
Definition slice_set_slice_disjoint (d : dag) :=
  fun e => match e with
    ExprApp (slice lo1 s1, [ExprApp (set_slice lo2 s2, [base; _])]) =>
      if N.leb (lo1 + s1) lo2 || N.leb (lo2 + s2) lo1
      then peel_disjoint_set_slices lo1 s1 base 8
      else e | _ => e end%bool%N.
```

```
(* Recursively peel disjoint set_slice layers. Needed for YMM (4-lane) operations
   where set_slice chains are 3 deep and a single peel isn't enough. *)
Fixpoint peel_disjoint_set_slices
  (lo1 s1 : N) (inner : expr) (fuel : nat) {struct fuel} : expr :=
  match fuel with
  | O => ExprApp (slice lo1 s1, [inner])
  | S fuel' =>
    match inner with
    | ExprApp (set_slice lo2 s2, [base; _]) =>
      if (N.leb (lo1 + s1) lo2 || N.leb (lo2 + s2) lo1)%bool
      then peel_disjoint_set_slices lo1 s1 base fuel'
      else ExprApp (slice lo1 s1, [inner])
    | _ => ExprApp (slice lo1 s1, [inner])
    end
  end%N.
```

**Pass ordering.** Rewrite rules are registered in a pipeline and run in sequence after each instruction. The ordering matters: `slice_set_slice` may expose a new `slice`/`set_slice` pattern that `slice_set_slice_disjoint` can simplify, and vice versa. For this reason, both rules appear multiple times in the rewrite pipeline to handle deeply nested patterns that require several rounds of simplification. This ordering was determined empirically by verifying test programs.

**Correctness.** Each rewrite rule must be proven to preserve DAG semantics: if the original expression evaluates to $v$, the rewritten expression must also evaluate to $v$. These proofs are mostly short and mechanical for rules that only inspect one layer of the DAG. For rules like `slice_set_slice_disjoint` that peel off layers recursively, we use inductive proofs.

## 5.5 Batched Field Operations

We implemented several batched field-arithmetic operations by hand in assembly, along with corresponding batched specifications that explicitly describe four independent field operations on concatenated inputs. For example, the batched-addition specification applies the scalar

`addmod` independently to four consecutive field elements:

```
Definition batched_addmod (a b : list Z) : list Z :=
  addmod limbwidth_num limbwidth_den n
    (firstn n a) (firstn n b) ++
  addmod limbwidth_num limbwidth_den n
    (firstn n (skipn n a)) (firstn n (skipn n b)) ++
  addmod limbwidth_num limbwidth_den n
    (firstn n (skipn (n+n) a)) (firstn n (skipn (n+n) b)) ++
  addmod limbwidth_num limbwidth_den n
    (firstn n (skipn (n+n+n) a)) (firstn n (skipn (n+n+n) b)).
```

This definition is straightforward: it slices the input lists into four groups of `n` limbs each, applies the scalar operation to each group independently, and concatenates the results. The same pattern applies to `batched_submod`, `batched_oppmod`, and `batched_carrymod`. These batched specifications are reified into PHOAS, optimized through the standard pipeline, and then compared against the assembly by the equivalence checker — exactly the same path as scalar operations.

Currently these batched specifications are written by hand for a fixed batch size of 4 (matching the four 64-bit lanes in a YMM register). However, there is no principled reason that batched specifications cannot be automatically derived from scalar versions. A Rocq function could generate a batched specification from any scalar specification by slicing inputs into groups and concatenating outputs, but we did not attempt such a function.

Batched assembly programs are harder to generalize. Addition and subtraction map trivially to `vpaddq` and `vpsubq`, and carry propagation can be expressed with shifts and masks (`vpsrlq`, `vpandq`). However, multiplication is fundamentally different: scalar multiplication produces a 128-bit result from two 64-bit inputs, meaning four independent multiplications simply cannot be performed in parallel without overflowing registers. For this reason, `vpmuludq` only multiplies the lower 32 bits of each lane. Full 64-bit SIMD multiplication requires a Karatsuba-like decomposition with multiple `vpmuludq`, shifts, and additions, producing an assembly program that bears little resemblance to the scalar version. Automated assembly generation would therefore require operation-specific lowering strategies, not a simple batching transform. This approach is possible in theory but extremely nontrivial.

# Chapter 6

# Proofs

Having implemented vector instructions in symbolic execution, we also need to prove correspondence between symbolic and semantic layers, which is fairly straightforward due to design choices in representing execution. Our proof strategy throughout is to isolate reasoning about each function to a single lemma, avoiding leakage of complexity between abstraction layers.

## 6.1 Binops

Consider the two implementations of the `vpaddq` instruction:

```
1 (* Symbolic execution of vpaddq, in Symbolic.v *)
2 SymbolicVector.SymexVectorBinOp dst src1 src2 (add 64) 64
```

```
1 (* Concrete semantics of vpaddq, in Semantics.v *)
2 let num_lanes := (N.to_nat (N.div s 64)) in
3 SemanticVector.DenoteVectorBinOp sa s st dst src1 src2
4    (fun a b => Z.land (a + b) (Z.ones 64)) num_lanes 64
```

The form of these two implementations already exemplifies why the abstractions in Figure 5.1 are so powerful. Rather than having to reason about correspondence of addition specifically, we need only prove that `SymexVectorBinOp` and `DenoteVectorBinOp` have equivalent effects on the symbolic and machine states, respectively.

Because these two functions decompose with the same recursive structure, we can simply prove correspondence between horizontal layers in Figure 5.1, as diagrammed in Figure 6.1[1].

[1] Generated from source files using a custom-built VSCode extension.

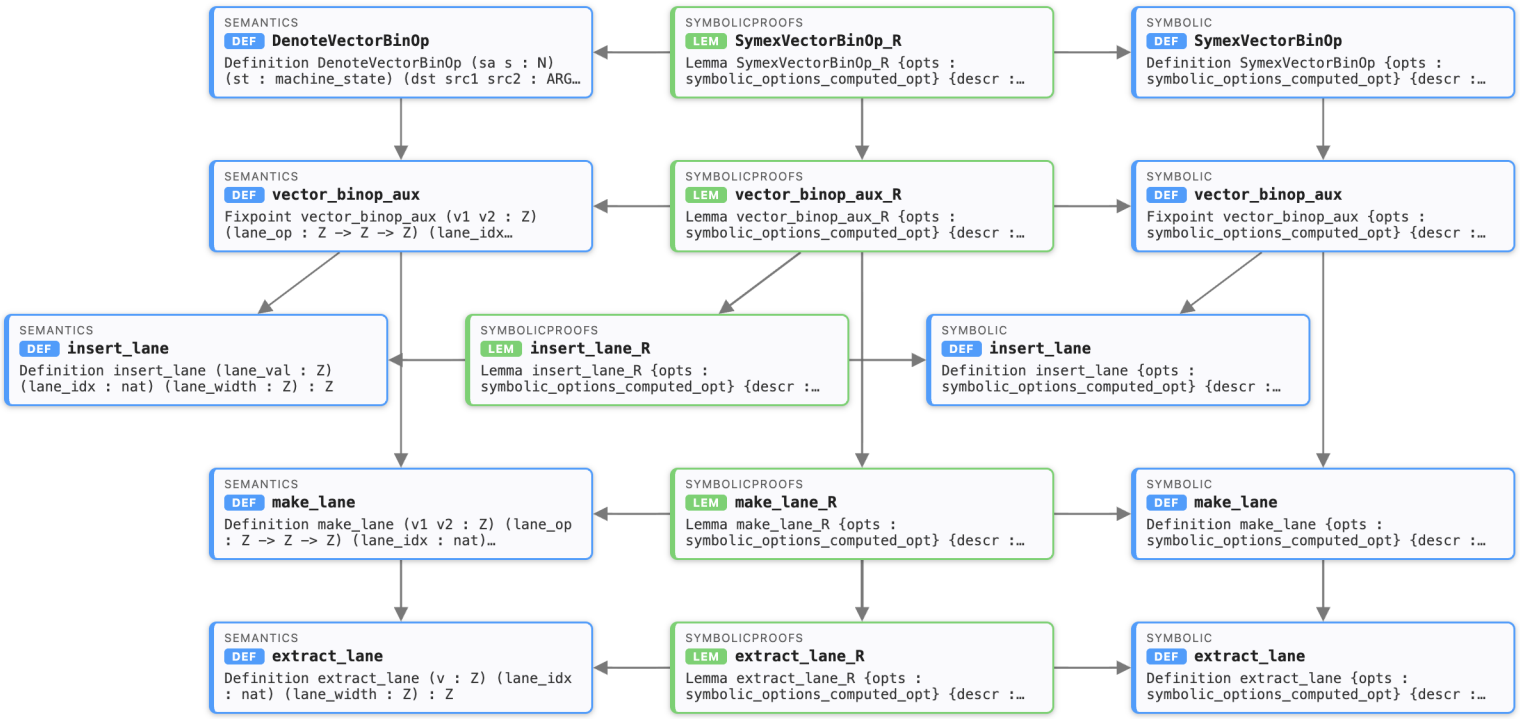


Figure 6.1: Dependency graph of correspondence lemmas relating the symbolic and concrete semantics of lane-wise binary vector operations. Each lemma relates one layer of abstraction from each side.

As one illustrative example, `make_lane_R` relates the two `make_lane` implementations:

```
(* make_lane computes one lane of a binop: lane_op v1[i] v2[i] *)
Lemma make_lane_R {opts : symbolic_options_computed_opt} {descr : description}
  s m s' (HR : R s m) res
  (i1 i2 : idx) (v1 v2 : Z)
  (Hv1 : eval s i1 v1) (Hv2 : eval s i2 v2)
  (lane_op : op) (* symbolic lane-wise operation *)
  (binop : Z -> Z -> Z) (* semantic lane-wise operation *)
  (Hop : interprets_as_binop lane_op binop) (* relating the two operations *)
  (lane_idx : nat) (lane_width : N) (Hlw : (lane_width > 0)%N)
  (H : SymbolicVector.make_lane i1 i2 lane_op lane_idx lane_width s = Success (res, s'))
  : eval s' res (SemanticVector.make_lane v1 v2 binop lane_idx (Z.of_N lane_width))
  /\ R s' m /\ s :< s'.
Proof using Type.
  unfold SymbolicVector.make_lane, SemanticVector.make_lane in *. repeat step_symex.
  eapply extract_lane_R in HSl1; try eassumption.
  eapply extract_lane_R in HSl2; try eassumption.
   step_symex. solve_R_subgoals. eauto.
Qed.
```

Notice that this proof unfolds exactly one layer of definitions and references the correspondence lemma for `extract_lane`. Also note the precondition `interprets_as_binop` in the statement of `make_lane_R`. This proposition states that the `lane_op` in the DAG actually represents the same operation as the semantic `binop`.

```
Definition interprets_as_binop (o : op) (f : Z -> Z -> Z) : Prop :=
forall a b, interp_op o [a; b] = Some (f a b).
```

This definition admits very simple (but subtly different) proofs for each binary instruction we implemented.

```
  Lemma interprets_as_add s :
      interprets_as_binop (add s) (fun a b => Z.land (a + b) (Z.ones (Z.of_N s))).
      Proof. intros a b. cbn [interp_op fold_right]. rewrite Z.add_0_r. reflexivity. Qed.

  Lemma interprets_as_sub s :
      interprets_as_binop (sub s) (fun a b => Z.land (a - b) (Z.ones (Z.of_N s))).
      Proof. intros a b. cbn [interp_op fold_right]. reflexivity. Qed.
  ...
```

Along with the main structural lemmas, these facts give us all the building blocks needed to prove correspondence for all binary instructions. In fact, these proofs all collapse to a single line:

```
(* all binops *)
all: eapply SymexVectorBinOp_R; try (eassumption || lia); auto with interprets_as.
```

which applies the top-level correspondence lemma `SymexVectorBinop_R`, then clears the preconditions using a hint database of all the `interprets_as_[op]` lemmas.

We stated and proved analogous lemmas for unary operations (a single source and destination, independent operation over lanes).

## 6.2 Non-binops

Not all instructions operate independently and identically on each lane of source and destination values. `vpbroadcastq` broadcasts one lane to other lanes of a register; `vpblendd` picks lanes from two source registers based on a bitmask; `vpmuludq` multiplies the lower 32 bits of each lane, operating independently across lanes but requiring an extra slicing step that does not fit neatly into our previous abstraction.

While these instructions do not fit the same binop pattern, they can all be decomposed into repeated lane-wise steps. Mirroring this structure, we used recursive functions to build up each lane of the output for all such instructions, and we continued the principle of proving correspondence after each step.

As an illustrative example, the `blend` fixpoint does one step of multiplexing between

source registers for a single lane, then recursively blends the remaining lanes:

```
(* Blend: for each lane, pick from v1 or v2 based on immediate mask bit *)
Fixpoint blend_aux {opts : symbolic_options_computed_opt} {descr : description}
  (v1 v2 : idx) (mask : Z) (lane_idx num_remaining : nat)
  (lane_width : N) (acc : idx) : M idx :=
  match num_remaining with
  | O => ret acc
  | S n =>
    let src := if Z.testbit mask (Z.of_nat lane_idx) then v2 else v1 in
    lane_val <- App (slice (N.of_nat lane_idx * lane_width) lane_width, [src]);
    new_acc <- App (set_slice (N.of_nat lane_idx * lane_width) lane_width, [acc; lane_val]);
    blend_aux v1 v2 mask (S lane_idx) n lane_width new_acc
  end.
```

We prove a correspondence lemma for `blend_aux` by induction on the number of remaining lanes, similar to `vector_binop_aux_R`. Each inductive step relates one lane's slice-and-insert to its semantic counterpart.

```
Lemma blend_aux_R {opts : symbolic_options_computed_opt} {descr : description}
      s m (HR : R s m)
      (i1 i2 : idx) (v1 v2 : Z)
      (Hv1 : eval s i1 v1) (Hv2 : eval s i2 v2) (mask : Z)
      (lane_idx num_remaining : nat) (lane_width : N) (Hlw : (lane_width > 0)%N)
      (acc : idx) (acc_val : Z) (Hacc : eval s acc acc_val)
      (Hacc_hi : Z.shiftr acc_val (Z.of_nat lane_idx * Z.of_N lane_width) = 0) res s'
      (H : SymbolicVector.blend_aux i1 i2 mask lane_idx num_remaining lane_width acc s
          = Success (res, s'))
  : R s' m /\ s :< s' /\
    eval s' res
      (Z.lor acc_val
        (SemanticVector.blend_aux v1 v2 mask lane_idx num_remaining
          (Z.of_N lane_width))).
```

For this and similar lemmas relating one step of symbolic execution, it makes sense to define a tactic like `step_blend` that applies the lemma and automatically clears preconditions. With this tactic, the final proof of correspondence for `vpblendd` reduces to a single line:

```
(* vpblendd *)
{ step_blend. step_symex. cleanup. exists m0. solve_R_subgoals. }
```

Analogous recursive lemmas, step tactics, and one-line proofs were used for `vpbroadcastq`, `vpmuludq`, `vpunpcklqdq`, and `vzeroupper`.

A few instructions do not decompose into repeated lane-wise steps at all. `vpextrq`, `vextracti128`, and `vinserti128` each extract or insert a single fixed-width chunk at a constant offset, so their symbolic execution is just a single `slice` or `set_slice` operation.

For example, `vpextrq` extracts a single 64-bit lane from a 128-bit XMM register at an immediate-specified index:

```
| vpextrq, [dst; src; imm] =>
  v <- GetOperand (s:=128%N) src;
  imm_val <- GetOperand imm;
  lane <- RevealConst imm_val;
  result <- App (slice (Z.to_N lane * 64)%N 64, [v]);
  SetOperand (s:=64%N) dst result
```

The resulting correspondence goals are pure bitvector equalities, which Rocq's `bitblast` tactic discharges automatically:

```
(* vpextrq, vextracti128, vinserti128 *)
all: normalize_NZ; rewrite Z2N.id in *;
     try solve [eapply DenoteOperand_nonneg; eassumption];
     bitblast.Z.bitblast.
```

## 6.3 Proof Automation

Working on such a large project necessitates proofs and tactics that are modular, intuitive, and robust to small changes in definitions or other tactics. Existing code provided some proof automation through the `step_symex` tactic, which steps through symbolic execution in the hypotheses to deduce relevant information, but many existing proofs were still extremely verbose and difficult to refactor.

Our approach to proof automation focused on tactics that were:

- **Idempotent**: repeated application cannot change the goal state beyond the first.
- **Pattern-targeted**: each tactic matches on a specific goal shape and is inert otherwise.
- **Unidirectional**: transformations only move toward solved goals, so tactics cannot undo each other's progress.

One example is `normalize_NZ`, which canonicalizes integer-arithmetic expressions through many rewrites (commutativity and associativity, identity elements, etc.), making equality proofs between two such expressions trivial. This tactic has the added benefit of standardizing the statements of integer arithmetic lemmas, so applying them requires no extra transformation.

Another illustrative example is the `solve_subsumed` tactic, designed to solve a frequent pattern in every correspondence proof:

$$\frac{s \prec s_0 \quad s_0 \prec s_1 \quad \cdots \quad s_n \prec s'}{s \prec s'}$$

This goal is theoretically simple because $\prec$ is transitive and reflexive, but not simple enough for Rocq to independently solve. One solution is an all-purpose search like `eauto`,

which works but is extremely opaque and can have downstream effects if it is called in the wrong goal state by accident. A tactic that surgically matches on the goal and applies the appropriate transformations avoids both issues:

```
Ltac solve_subsumed ::=
    match goal with
    | |- _ :< _ =>
      solve [ eassumption
            | apply subsumed_refl
            | repeat (eapply subsumed_trans; [eassumption|]); eapply subsumed_refl ]
    end.
```

These principles made proofs for vector instructions extremely modular and interpretable, and we attempted to refactor many existing proofs similarly. This refactor was possible for many helper lemmas but not for the main correspondence proofs of scalar instructions, which were far too sensitive to any changes and would require far more dedicated effort. As a practical middle ground, the two instruction sets share many lower-level lemmas but use completely separate, self-contained sets of tactics at the higher level.

# Chapter 7

# Results

Prior to this work, Fiat Cryptography's equivalence checker could only verify scalar x86-64 assembly. With the extensions described in the preceding chapters, it now handles 19 AVX2 vector instructions:

- **Arithmetic:** `vpaddq`, `vpsubq`, `vpaddd`, `vpsubd`, `vpmuludq`, `vpandq`, `vpxorq`
- **Shifts:** `vpsrlq`, `vpsllq`
- **Data movement:** `vmovdqu`, `vmovq`, `vpbroadcastq`, `vpblendd`
- **Shuffles:** `vpunpcklqdq`, `vpunpckhqdq`, `vpextrq`, `vextracti128`, `vinserti128`
- **Other:** `vzeroupper`

We evaluate our implementation on a suite of hand-written AVX2 assembly programs that implement field-arithmetic operations for Curve25519. Each test verifies that the assembly program computes the same function as the corresponding Fiat Cryptography specification, using the equivalence checker.

Our test suite, listed in Table 7.1, covers both limb-parallel and batched SIMD programs, as described in section 2.2, along with two 4×-scalar baselines used as comparison points in our runtime benchmarks. Checking time appears to scale roughly linearly with instruction count and is at most a few seconds even for the largest test, which is inconsequential in practice since each program is only checked once.

Two representative programs are shown in Figure 7.1 and Figure 7.2.

## 7.1 Runtime Performance

While correctness is the primary goal of this work, we also evaluate the performance of our handwritten arithmetic primitives to demonstrate the performance gain from even simple vector optimizations. For each vectorized assembly program in Table 7.1, we benchmark against a scalar baseline performing the same batched operation using four sequential calls to the corresponding function synthesized by Fiat Cryptography. The scalar baselines are compiled once with `gcc -O2`, and the resulting assembly is frozen to ensure reproducibility.

| Test name | Description | Instructions | Checking Time (s) |
|---|---|---|---|
| avx-xmm-add | 128-bit limb-parallel add | 14 | 0.230 |
| avx-ymm-add | 256-bit limb-parallel add | 10 | 0.208 |
| avx-xmm-sub | 128-bit limb-parallel sub | 34 | 0.258 |
| avx-ymm-sub | 256-bit limb-parallel sub | 18 | 0.234 |
| batch-avx-add | batched add | 18 | 0.317 |
| batch-avx-sub | batched sub | 33 | 0.421 |
| batch-avx-opp | batched opp | 23 | 0.342 |
| batch-avx-carry | batched carry | 131 | 0.602 |
| batch-avx-carry-add | batched carry composed with add | 191 | 0.755 |
| batch-avx-carry-sub | batched carry composed with sub | 202 | 0.876 |
| batch-avx-carry-opp | batched carry composed with opp | 142 | 0.665 |
| batch-scalar-carry | 4× scalar carry baseline | 161 | 0.602 |
| batch-carry-mul | 4× scalar carry_mul baseline | 705 | 6.854 |

Table 7.1: Equivalence-checking test suite. Each AVX2 assembly program is checked against the corresponding Fiat Cryptography specification for Curve25519 with 64-bit limbs. The two `batch-scalar-*` tests are non-vectorized 4×-unrolled scalar baselines used as reference points for the runtime measurements in section 7.1.

Both scalar and vectorized sides are precompiled assembly linked against the same benchmark harness. We run on an AMD EPYC 7502P pinned to a single core via `taskset`, reporting the mean and standard deviation over 10 runs of $10^7$ iterations each. Results are reported in Table 7.2.

The results group into three regimes. Batched operations on contiguous data show speedups between 2.2× and 5.5× over the scalar baseline, roughly the theoretical ceiling.

Limb-parallel vector operations get no meaningful speedup. Field elements have five 64-bit limbs, but XMM and YMM registers store two and four limbs respectively, so the vectorized program still handles a tail in scalar instructions and pays overhead at the boundary. Limb-parallel optimization might be more effective in representations with different limb counts.

Batched carry operations are slower than scalar in two cases and only modestly faster in others. Batching carry propagation across elements is difficult because it requires transforming the data layout, which incurs a fixed overhead. This fixed cost dominates for short programs like these, but would be less significant for longer chains of batched operations.

## 7.2 Implementation Effort

The extensions described in this thesis add roughly 3,650 lines across seven files of the Fiat-Crypto repository. Table 7.3 breaks down the changes by file. The majority of the new code constitutes proofs of correspondence in `SymbolicProofs.v` (described in chapter 6).

```
SECTION .text
    GLOBAL fiat_25519_add

fiat_25519_add:
    ; limbs 0-1: load 128 bits (two 64-bit limbs), add, store
    vmovdqu xmm0, [rsi]
    vmovdqu xmm1, [rdx]
    vpaddq  xmm0, xmm0, xmm1
    vmovdqu [rdi], xmm0
    ; limbs 2-3: same pattern at offset +16
    vmovdqu xmm0, [rsi + 16]
    vmovdqu xmm1, [rdx + 16]
    vpaddq  xmm0, xmm0, xmm1
    vmovdqu [rdi + 16], xmm0
    ; limb 4: single 64-bit limb via vmovq
    vmovq   xmm0, [rsi + 32]
    vmovq   xmm1, [rdx + 32]
    vpaddq  xmm0, xmm0, xmm1
    vmovq   [rdi + 32], xmm0
    ret
```

Figure 7.1: XMM vectorized field addition for Curve25519. Adds five 64-bit limbs using 128-bit `vpaddq` instructions, processing two limbs per iteration with a single-limb tail.

| **Test** | **Scalar (ns)** | **AVX2 (ns)** | **Speedup** |
|---|---|---|---|
| *Batched, contiguous YMM loads (no layout transposition)* | | | |
| batch-avx-add | $13.63 \pm 0.05$ | $6.31 \pm 0.02$ | $2.16\times$ |
| batch-avx-sub | $34.14 \pm 0.09$ | $8.62 \pm 0.03$ | $3.96\times$ |
| batch-avx-opp | $37.38 \pm 0.18$ | $6.81 \pm 0.03$ | $5.49\times$ |
| *Limb-parallel (single element)* | | | |
| avx-xmm-add | $2.62 \pm 0.03$ | $3.15 \pm 0.02$ | $0.83\times$ |
| avx-ymm-add | $2.65 \pm 0.02$ | $4.70 \pm 0.03$ | $0.56\times$ |
| avx-xmm-sub | $6.52 \pm 0.12$ | $5.48 \pm 0.04$ | $1.19\times$ |
| avx-ymm-sub | $6.68 \pm 0.06$ | $5.99 \pm 0.03$ | $1.11\times$ |
| *Batched carry (gather/scatter required)* | | | |
| batch-avx-carry | $22.31 \pm 0.16$ | $27.72 \pm 0.03$ | $0.80\times$ |
| batch-avx-carry-add | $33.03 \pm 0.10$ | $40.17 \pm 0.18$ | $0.82\times$ |
| batch-avx-carry-sub | $51.03 \pm 0.19$ | $42.20 \pm 0.12$ | $1.21\times$ |
| batch-avx-carry-opp | $43.06 \pm 0.42$ | $29.62 \pm 0.14$ | $1.45\times$ |

Table 7.2: Runtime of verified AVX2 implementations against scalar Fiat Cryptography baselines on an AMD EPYC 7502P. The scalar baseline for each batched test is four sequential calls to the corresponding scalar primitive. Each cell reports the mean $\pm$ standard deviation over 10 runs of $10^7$ iterations.

```
SECTION .text
    GLOBAL fiat_25519_batch_add

fiat_25519_batch_add:
    ; group 0: limb 0 of all 4 elements
    vmovdqu ymm0, [rsi]
    vpaddq  ymm0, ymm0, [rdx]
    vmovdqu [rdi], ymm0
    ; group 1: limb 1 of all 4 elements
    vmovdqu ymm1, [rsi + 32]
    vpaddq  ymm1, ymm1, [rdx + 32]
    vmovdqu [rdi + 32], ymm1
    ; group 2: limb 2 of all 4 elements
    vmovdqu ymm2, [rsi + 64]
    vpaddq  ymm2, ymm2, [rdx + 64]
    vmovdqu [rdi + 64], ymm2
    ; group 3: limb 3 of all 4 elements
    vmovdqu ymm3, [rsi + 96]
    vpaddq  ymm3, ymm3, [rdx + 96]
    vmovdqu [rdi + 96], ymm3
    ; group 4: limb 4 of all 4 elements
    vmovdqu ymm4, [rsi + 128]
    vpaddq  ymm4, ymm4, [rdx + 128]
    vmovdqu [rdi + 128], ymm4
    vzeroupper
    ret
```

Figure 7.2: Batched YMM field addition for Curve25519. Each 256-bit `vpaddq` adds the same limb position across four independent field elements stored in structure-of-arrays layout.

| File | Lines added | Purpose |
|---|---|---|
| `SymbolicProofs.v` | 2324 | Correspondence proofs |
| `Symbolic.v` | 727 | Symbolic-execution rules, rewrite rules |
| `Semantics.v` | 247 | Concrete (denotational) semantics |
| `Syntax.v` | 235 | Opcode constructors, operand size inference |
| `Equivalence.v` | 76 | 128/256-bit DAG-node support |
| `Parse.v` | 36 | Memory-operand parsing (vector indices) |
| `Equality.v` | 5 | `MEM`-type equality update |
| **Total** | **3,650** | |

Table 7.3: Lines of Rocq added per file.

# Chapter 8

# Discussion and Future Work

This thesis extended Fiat Cryptography's equivalence checker from scalar-only x86-64 to a subset of AVX2 enabling SIMD field arithmetic. Our additions include refactored register and memory infrastructure for 128-bit and 256-bit operands, symbolic execution and concrete semantics for 19 AVX2 instructions, verified correspondence proofs for all implemented instructions, rewrite rules for simplifying vector-induced DAG patterns, and a test suite of verified vectorized-field-arithmetic programs.

Two limitations are worth noting. First, the rewrite system is best-effort: rules are syntactic pattern matches on DAG structure, determined empirically. Some equivalent programs may not reduce to the same DAG form if they use instruction sequences the rules do not anticipate. This limitation could be addressed by adopting a full e-graph with union-find, but the current approach has been sufficient in practice. Second, this work only verifies externally hand-written assembly. There is currently no way to synthesize vectorized programs automatically through Fiat Cryptography or CryptOpt.

## 8.1 Future Work

**Automated batched specifications.** We wrote batched versions of cryptographic primitives in both PHOAS and assembly. The PHOAS specifications could be fully automated, as described in section 5.5.

**CryptOpt integration.** The most impactful next step would be integrating AVX2 support into CryptOpt's stochastic optimizer, enabling it to generate and verify optimized vector assembly automatically. Stochastic optimization has the potential to discover extremely unintuitive implementations using vector instructions.

**New synthesis/optimization pipelines.** Following the same principle but a completely new approach, an LLM could serve a similar role to CryptOpt by generating untrusted assembly code, relying on the equivalence checker to verify correctness.

**Performance benchmarking.** Our test programs are hand-written to exercise the equivalence checker, not to maximize throughput. Comparing the performance of verified vectorized

implementations against scalar baselines and unverified SIMD code would quantify the practical benefit of this infrastructure.

**AVX-512.** Our work generalizes naturally to 512-bit ZMM registers. Our refactored memory model, lane decomposition, and rewrite rules are all parameterized by width, and adding AVX-512 support would be straightforward. We chose to focus on AVX2 because it is more widely supported in modern hardware.

# Appendix A

# Dependency Graphs

We show dependency graphs for `GetOperand` and `SetOperand`, two of the top-level functions used in symbolic execution, as well as for their correspondence lemmas, `GetOperand_R` and `R_SetOperand`. Arrows indicate that one definition calls another, or that one lemma is used in the proof of another. These graphs exclude many helper functions and lemmas that are not essential to the high-level structure.

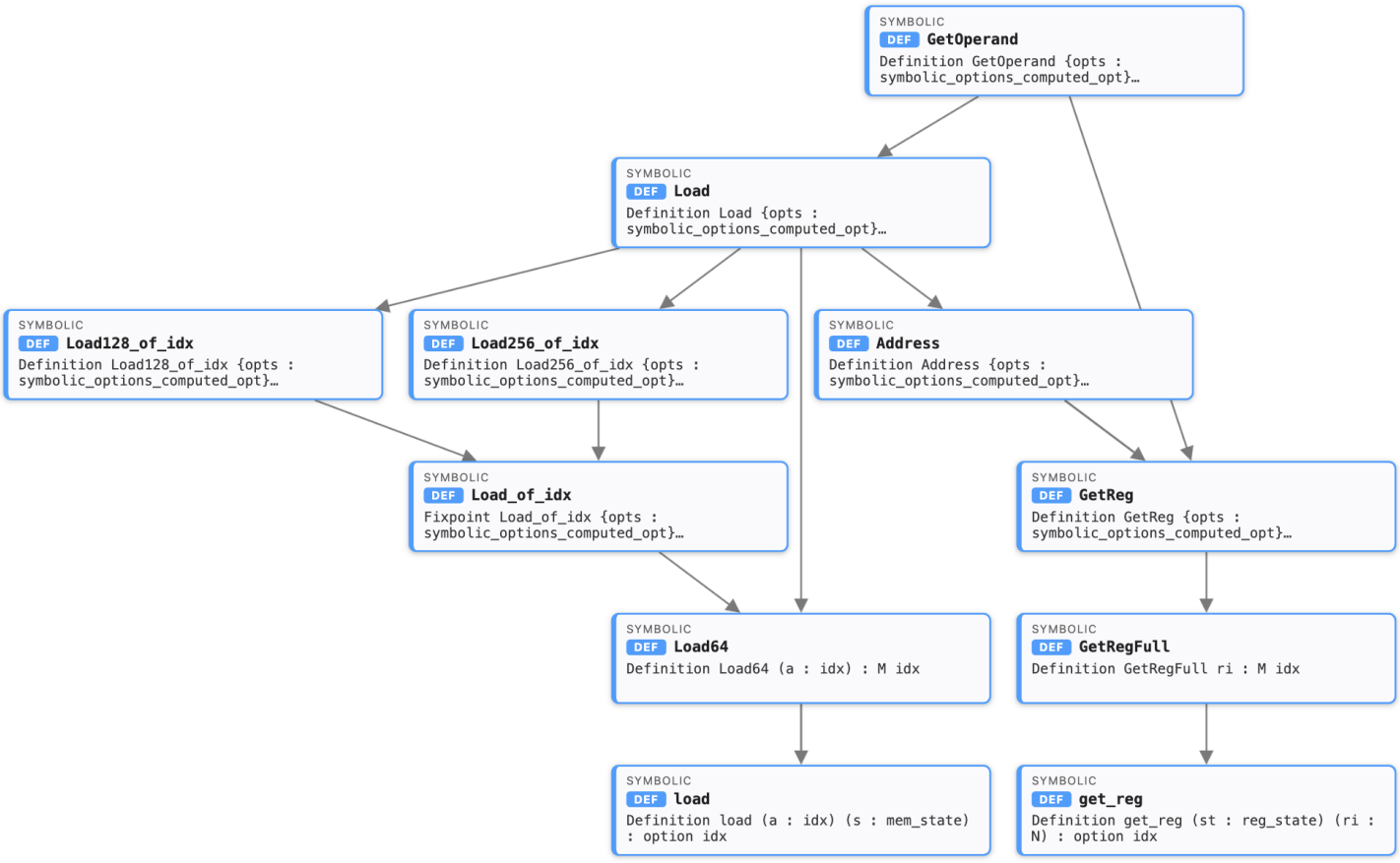


Figure A.1: Dependency graph of `GetOperand`, which resolves a source operand to a DAG index by dispatching on operand type (register, memory, immediate).

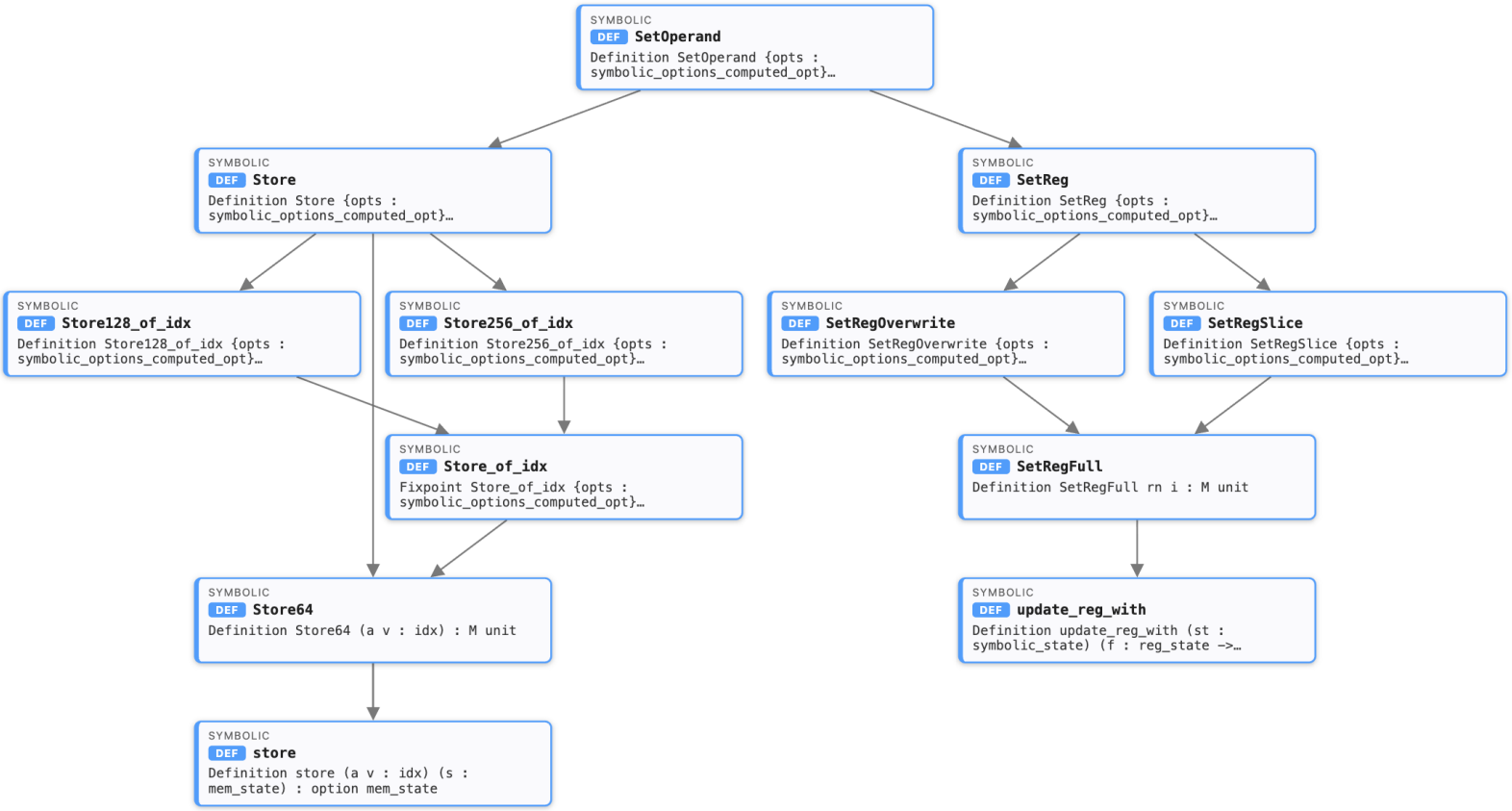


Figure A.2: Dependency graph of `SetOperand`, which writes a DAG index to a destination operand, handling register aliasing and memory stores.

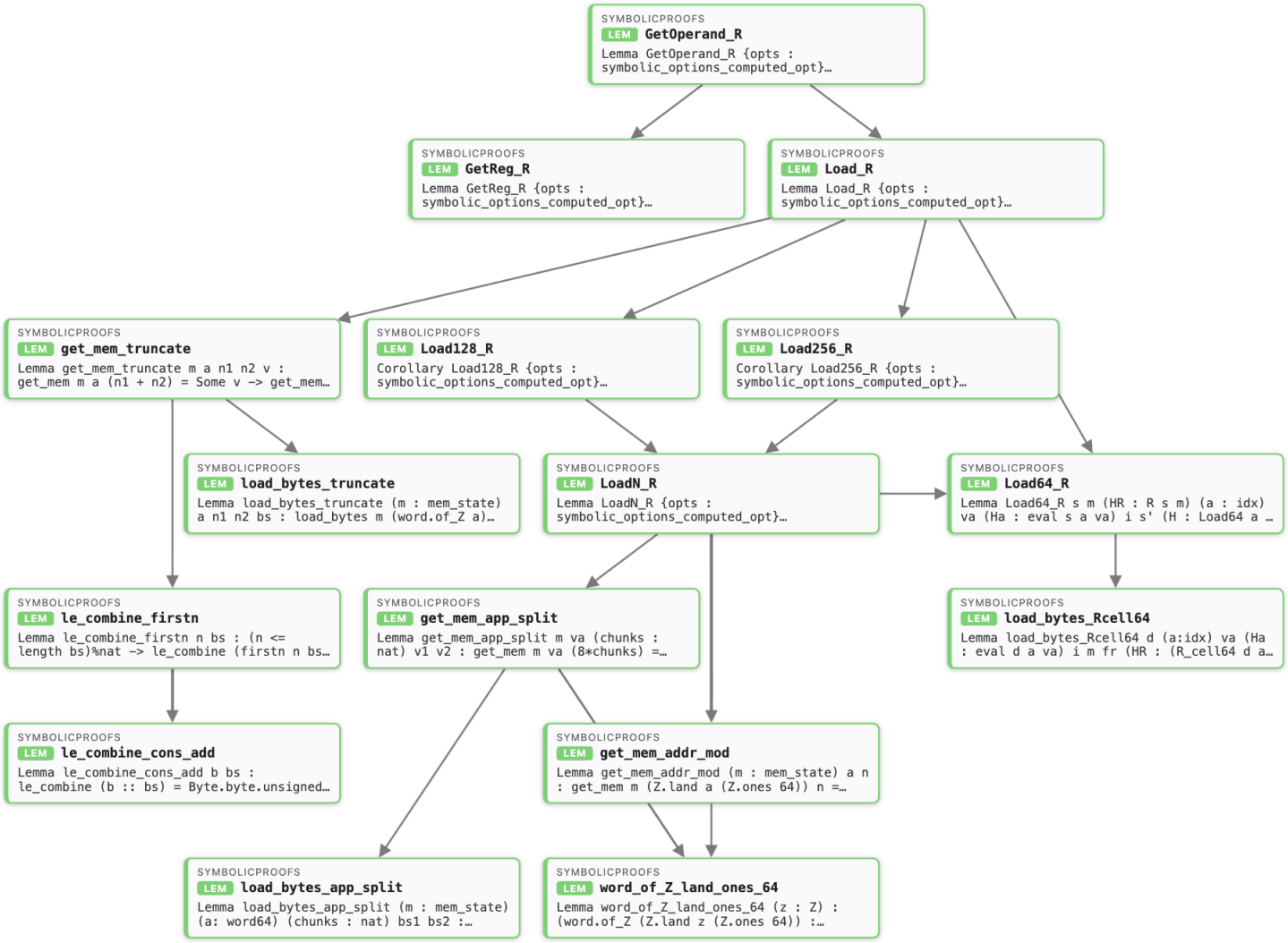


Figure A.3: Dependency graph of `GetOperand_R`, the correspondence lemma relating symbolic `GetOperand` to concrete operand evaluation.

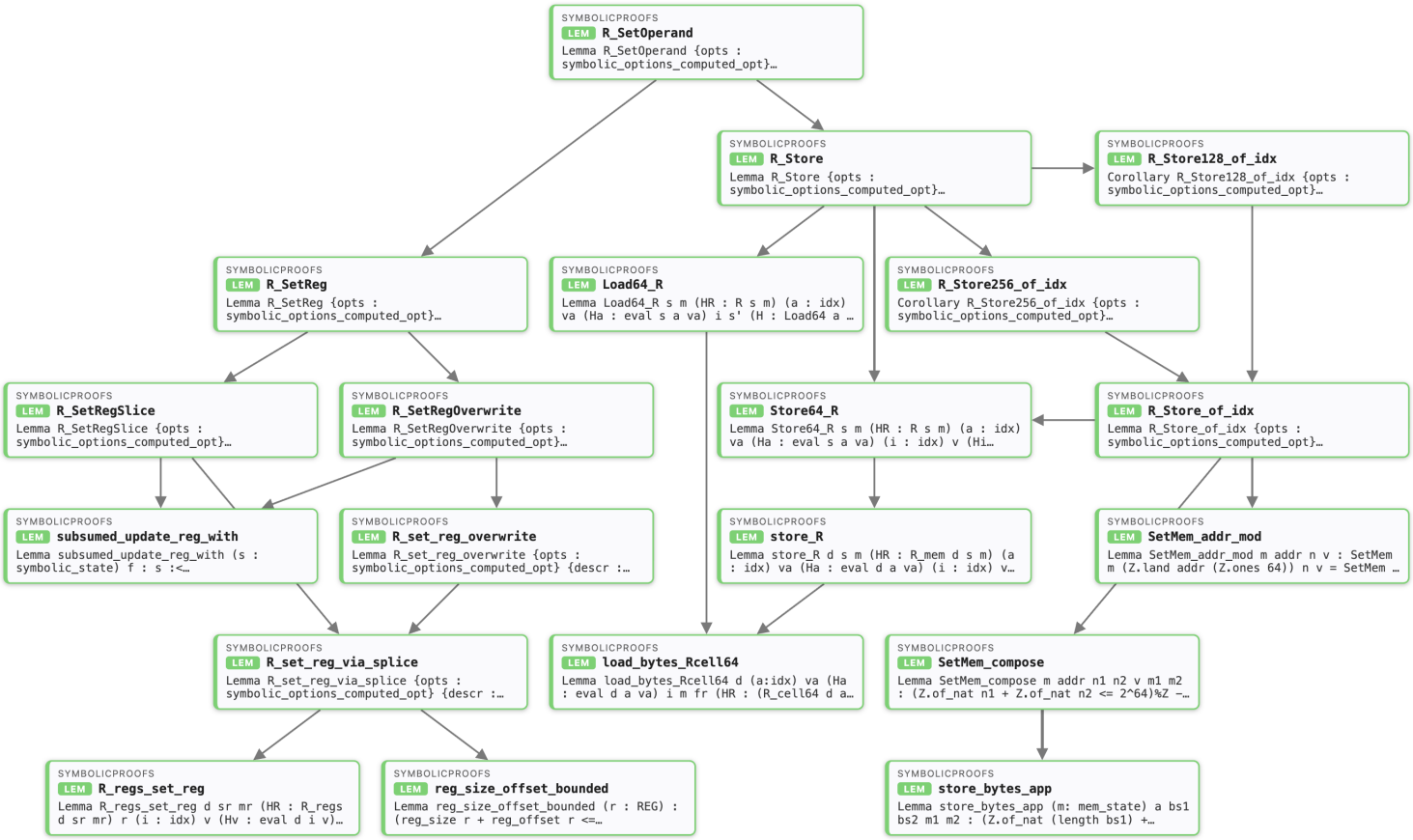


Figure A.4: Dependency graph of `R_SetOperand`, the correspondence lemma relating symbolic `SetOperand` to concrete writing of registers or memory addresses.

# References


[1] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. "Simple High-Level Code for Cryptographic Arithmetic — With Proofs, Without Compromises." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2019. DOI: 10.1109/SP.2019.00005.

[2] J. Kuepper, A. Erbsen, J. Gross, O. Conoly, C. Sun, S. Tian, D. Wu, A. Chlipala, C. Chuengsatiansup, D. Genkin, M. Wagner, and Y. Yarom. "CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives." *Proceedings of the ACM on Programming Languages*, **7**(PLDI), 2023. DOI: 10.1145/3591272. arXiv: 2211.10665.

[3] A. Chlipala. "Parametric Higher-Order Abstract Syntax for Mechanized Semantics." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2008. DOI: 10.1145/1411204.1411226.

[4] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. "Jasmin: High-Assurance and High-Speed Cryptography." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134078.

[5] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. "HACL*: A Verified Modern Cryptographic Library." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017. DOI: 10.1145/3133956.3134043.

[6] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. "Vale: Verifying High-Performance Cryptographic Assembly Code." In: *Proceedings of the 26th USENIX Security Symposium*. 2017.

[7] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Béguelin. "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2020. DOI: 10.1109/SP40000.2020.00114.

[8] M. Polubelova, K. Bhargavan, J. Protzenko, B. Beurdouche, A. Fromherz, N. Kulatova, and S. Zanella-Béguelin. "HACL×N: Verified Generic SIMD Crypto (for all your favourite platforms)." In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020. DOI: 10.1145/3372297.3423352.