Symbolic Crosschecking of Floating-Point and SIMD Code

Peter Collingbourne   Cristian Cadar   Paul H. J. Kelly

Department of Computing
Imperial College London
{peter.collingbourne03, c.cadar, p.kelly}@imperial.ac.uk

Abstract

We present an effective technique for crosschecking an IEEE 754 floating-point program and its SIMD-vectorized version, implemented in KLEE-FP, an extension to the KLEE symbolic execution tool that supports symbolic reasoning on the equivalence between floating-point values.

The key insight behind our approach is that floating-point values are only reliably equal if they are essentially built by the same operations. As a result, our technique works by lowering the Intel Streaming SIMD Extension (SSE) instruction set to primitive integer and floating-point operations, and then using an algorithm based on symbolic expression matching augmented with canonicalization rules.

Under symbolic execution, we have to verify equivalence along every feasible control-flow path. We reduce the branching factor of this process by aggressively merging conditionals, if-converting branches into `select` operations via an aggressive phi-node folding transformation.

We applied KLEE-FP to OpenCV, a popular open source computer vision library. KLEE-FP was able to successfully crosscheck 51 SIMD/SSE implementations against their corresponding scalar versions, proving the *bounded* equivalence of 41 of them (i.e., on images up to a certain size), and finding inconsistencies in the other 10.

Categories and Subject Descriptors   C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Single-instruction-stream, multiple-data-stream processors (SIMD); D.2.4 [*Software/Program Verification*]: Reliability; D.2.5 [*Testing and Debugging*]: Symbolic execution

General Terms   Reliability, Verification

1. Introduction

Single Instruction Multiple Data (SIMD) computing is an increasingly popular means of improving the performance of programs by exploiting their data level parallelism. A number of traditionally scalar architectures have been extended with SIMD support, such as the Streaming SIMD Extensions (SSE), 3DNow! and Advanced Vector Extensions (AVX) for x86, NEON for ARM, or AltiVec for PowerPC. Furthermore, GPU programming languages, such as OpenCL and CUDA, are based on the SIMD execution model.

SIMD processors exploit data level parallelism by providing instruction sets that operate on one-dimensional arrays of data called vectors. While automatic vectorization is an active area of research [Eichenberger 2004, Larsen 2000, Naishlos 2003], the difficulty of reasoning about data dependencies and arithmetic precision means that optimizing scalar code to use SIMD instructions is still a mostly manual process. Unfortunately, manually translating scalar code into an equivalent SIMD version is a difficult task, because any programming error may cause the hand-optimized SIMD code to act differently from the purportedly equivalent scalar version. In this paper, we propose a novel automatic technique for verifying that the SIMD version of a piece of code is equivalent to its (original) scalar version.

Our technique is based on symbolic execution [King 1975], which provides a systematic way for exploring all feasible paths in a program for inputs up to a certain size. On each explored path, our technique works by building the symbolic expressions associated with the scalar and respectively the SIMD version of the code, and proving their equivalence.

While symbolic crosschecking has been successfully employed in the past (e.g., in the context of block cipher implementations [Smith 2008]), we need to address a series of challenges to apply it to the verification of SIMD vectorizations. First, we need to model the semantics of a real SIMD instruction set, which the current generation of symbolic execution tools do not handle. Second, and more importantly, SIMD code makes intensive use of floating point operations. Due to the complexity of floating point semantics [IEEE Task P754 2008], it is extremely difficult — if not infeasible — to build a constraint solver for floating point,

and as a result there are currently no such constraint solvers available. Thus, in this paper we take a different approach, in which we prove the equivalence of two symbolic floating point expressions by first applying a series of expression canonicalization rules, and then syntactically matching the two expressions. The key insight into why our approach works is that constructing two equivalent values from the same inputs in floating point can usually only be done reliably by performing the same operations.

This paper makes the following contributions:

1. We present a symbolic execution (SE) based technique for crosschecking SIMD vectorizations against their scalar implementations.

2. We implement our technique in a tool called KLEE-FP, an extension to the open source symbolic execution tool KLEE [klee.llvm.org].

3. We reason about floating-point values (which KLEE's constraint solver cannot handle), using expression matching augmented with canonicalization rules that express strict equivalences in floating-point and mixed FP-integer expressions. As far as we know, this is the first practical SE-based technique that can precisely handle IEEE 754 floating point arithmetic.

4. We evaluate our technique by applying KLEE-FP to OpenCV [Intel], a popular open source computer vision library. KLEE-FP was able to crosscheck a total of 51 SIMD/SSE implementations against their corresponding scalar versions, proving the bounded equivalence of 41 of them on images up to a certain size, and finding inconsistencies in the other 10 pairs.

To achieve this, the semantics for a substantial portion of the Intel SSE instruction set are implemented via translation to an intermediate representation. We improve the tractability of our technique by implementing an aggressive variant of if-conversion using phi-node folding [Chuang 2003, Lattner 2004], to replace control-flow forking with predicated `select` instructions, in order to reduce the number of paths explored by symbolic execution.

2. Overview

This section illustrates the main features of our technique by showing how it can be used to verify the equivalence between a scalar and an SIMD implementation of a simple routine. Our code example, shown in Figure 1, is based on one of the OpenCV benchmarks we evaluated (specifically `thresh(BINARY_INV, f32)`; see §5). The code defines a routine called `zlimit`, which takes as input a floating point array `src` of size `size`, and returns as output the array `dst` of the same size. Each element of `dst` is the greater of the corresponding elements of `src` and 0. The routine consists of both a scalar and an SIMD implementation; users choose

```
1   void zlimit (int simd, float *src, float *dst,
2                 size_t  size) {
3       if (simd) {
4           __m128 zero4 = _mm_set1_ps(0.f);
5           while ( size >= 4) {
6               __m128 srcv = _mm_loadu_ps(src);
7               __m128 cmpv = _mm_cmpgt_ps(srcv, zero4);
8               __m128 dstv = _mm_and_ps(cmpv, srcv);
9               _mm_storeu_ps(dst, dstv);
10              src += 4;  dst += 4;  size −= 4;
11          }
12      }
13      while ( size) {
14          *dst = *src > 0.f ? *src : 0.f;
15          src++; dst++; size−−;
16      }
17  }
18
19  int main(void) {
20      float  src [64], dstv [64], dsts [64];
21      uint32_t *dstvi = ( uint32_t *)dstv;
22      uint32_t *dstsi = ( uint32_t *)dsts;
23      unsigned i;
24      klee_make_symbolic(src, sizeof (src), "src");
25      zlimit (0, src, dsts, 64);
26      zlimit (1, src, dstv, 64);
27      for (i = 0; i < 64; ++i)
28          assert ( dstvi [i] == dstsi [i]);
29  }
```

Figure 1. Simple test benchmark.

between the two versions via the `simd` argument. The SIMD implementation makes use of Intel's SSE instruction set.

The first loop of the routine, at lines 5–11, contains the core of the SIMD implementation, and is a good illustration of how SIMD code is structured. Each iteration of the loop processes four elements of array `src` at a time. The variables `srcv`, `cmpv` and `dstv` are of type `__m128`, i.e., 128-bit vectors consisting of four `float`s each. The code first loads four values from `src` into `srcv` by using the SIMD instruction `_mm_loadu_ps()` (line 6). It then compares each element of `srcv` to the corresponding element of `zero4`, which was initialized on line 4 to a vector of four 0 values (line 7). The output vector `cmpv` contains the result of each comparison as a vector of four 32-bit bitmasks each consisting of all-ones (if the `srcv` element was > 0) or all-zeros (otherwise). Next it applies the `cmpv` bitmask to `srcv` by performing a bitwise AND of `cmpv` and `srcv` to produce `dstv`, a copy of `srcv` with values ≤ 0 replaced by 0 (line 8). Finally, it stores `dstv` into `dst` (line 9).

The second loop of the `zlimit` routine, at lines 13–16, is the scalar implementation, which is also used by the SIMD version to process the last few elements of `src` when the size is not an exact multiple of 4.
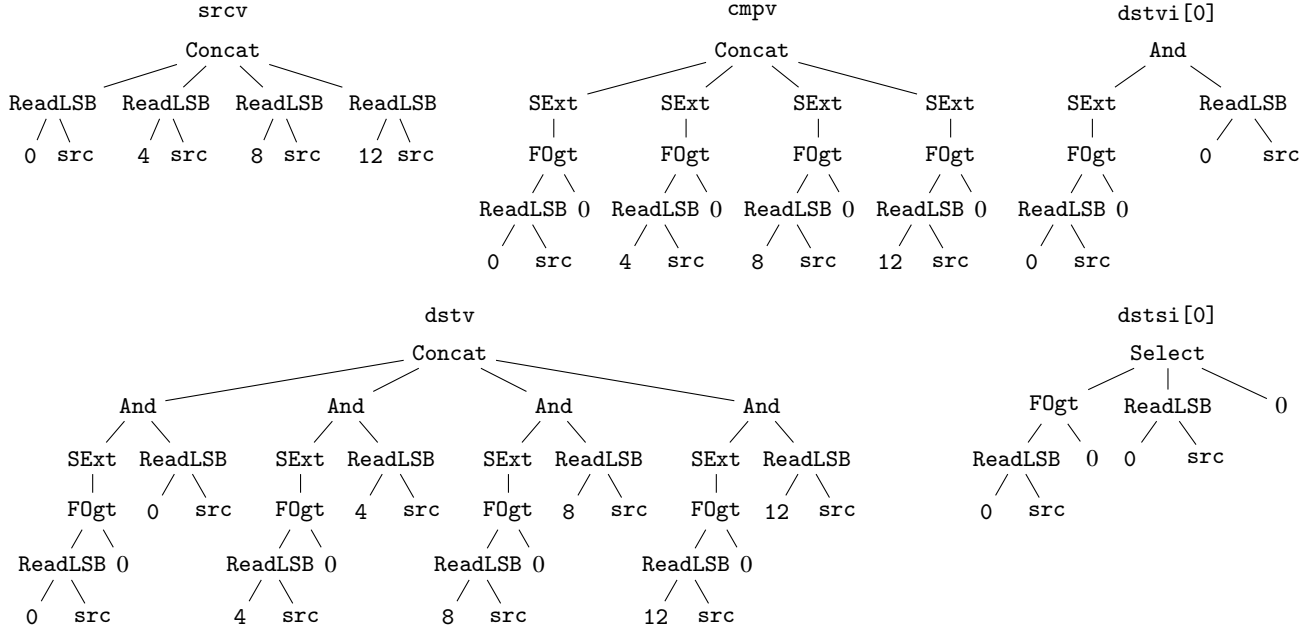
Figure 2. Symbolic expressions assigned to variables `srcv`, `cmpv`, `dstv` and to the array elements `dstvi[0]` and `dstsi[0]` of Figure 1. `src` represents the symbolic array `src`. The `ReadLSB` (Read Least Significant Byte first) node represents a 4-byte little-endian array read, `FOgt` floating point greater-than comparison, `SExt` sign extension, `Select` the equivalent of the C ternary operator and `Concat` bitwise concatenation.

The `main` function constitutes the test harness. In order to use KLEE-FP, developers have to identify the scalar and the SIMD versions of the code being checked, and the inputs and outputs to these routines. In our example, we have one input, namely the array `src`. Thus, the first step is to mark this array as *symbolic*, meaning that its elements could initially have any value (see §4.1 for more details). This is accomplished on line 24 by calling the function `klee_make_symbolic()` provided by KLEE, which takes three arguments: the address of the memory region to be made symbolic, its size in bytes, and a name used for debugging purposes only. Then, on line 25 we call the scalar version of the code and store the result in `dsts`, and on line 26 we call the SIMD version and store the result in `dstv`. Finally, on lines 27–28 each element of `dstv` is compared against the corresponding element of `dsts`. Note that we use bitcasting to integers via the pointers `dstvi` and `dstsi` for a bitwise comparison. As we will further discuss in Section 3, this is necessary because in the presence of `NaN` (*Not a Number*) values, the C floating point comparison operator `==` does not always return `true` if its floating-point operands are the same, as distinguished from a bitwise comparison.

To use KLEE-FP to run this benchmark, the user first compiles the code to LLVM bitcode [Lattner 2004], the low level representation on which KLEE and our extension KLEE-FP operate. The bitcode file can then be run directly by KLEE-FP.

Before KLEE-FP begins executing the input bitcode file, it first carries out a number of transformations. One of these is a lowering pass that replaces instruction-set specific SIMD operations with standard, instruction-set neutral instructions. Section 3.3 discusses this pass in more detail.

KLEE-FP interprets a program by evaluating the transformed bitcode instructions sequentially. During symbolic execution, values representing variables and intermediate expressions are manipulated. Both vector and scalar values are represented as bitvectors: concrete values by bitvector constants and symbolic ones by bitvector expressions. Vectors have bitwidth $s \times n$, where s is the bitwidth of the underlying scalar and n is the number of elements in the vector. Section 3 gives more details on our modeling approach.

For example, during the first iteration of the `zlimit` SIMD loop, the variables `srcv`, `cmpv` and `dstv` defined at lines 6–8 in Figure 1 are represented by the three expressions shown on the left hand side of Figure 2. Similarly, the results `dstvi[0]` and `dstsi[0]` are represented by the two expressions shown on the right side of Figure 2.

When KLEE-FP reaches an `assert` statement, it tries to prove that the associated expression is always `true`. For example, during the first iteration of the loop at lines 27–28 the expressions `dstvi[0]` and `dstsi[0]` are compared. To this end, KLEE-FP applies a series of expression rewrite rules, whose goal is to bring the expressions to a canonical normal form. As discussed in Section 4.4, one of our canonicalization rules transforms an expression tree of the

form $\text{And}(\text{SExt}(P), X)$ into $\text{Select}(P, X, 0)$, where P is an arbitrary boolean predicate and X an arbitrary expression. For our example, this rule transforms the expression corresponding to `dstvi[0]` shown in Figure 2 to be identical to expression `dstsi[0]`, shown in the same figure. Once both expressions are canonicalized, we attempt to prove their equivalence by (1) using a simple syntactical matching for the floating-point subtrees, and (2) using a constraint solver for the integer subtrees. As highlighted in the introduction, the reason we are able to prove the equivalence of floating-point expressions by bringing them to canonical form and then syntactically matching them is that constructing two equivalent values from the same inputs in floating point can usually only be done reliably in a limited number of ways. As a consequence, we found that in practice we only need a relatively small number of expression canonicalization rules in order to apply our technique to real code (see §4.4).

One concern not covered by this simple example, which has a single execution path, is the number of proofs that are needed: under symbolic execution, every feasible program path is explored, and we have to conduct the proof on every path. Thus, an important optimization is to reduce the number of paths explored by merging multiple ones together. This optimization is discussed in detail in Section 4.2.

3. Modeling Floating Point and SSE Operations

This section discusses our approach for modeling floating point and SSE operations in KLEE-FP. In Section 3.1 we start by presenting our floating point extension to KLEE. Then, in Section 3.2 we describe our modeling of SSE vector operations, and in Section 3.3 we present our lowering pass that translates SSE intrinsics into standard LLVM operations. Finally, in Section 3.4 we discuss the way we handle LLVM atomic intrinsics.

3.1 Floating Point Operations

In order to add support for floating point, we extended KLEE's constraint language to include floating point types and operations. Floating point operation semantics are derived from those presented by LLVM, whose floating point instructions include $+$, $-$, $\times$, $\div$, remainder, conversion to and from signed or unsigned integer values (`FPToSI`, `FPToUI`, `UIToFP`, `SIToFP`), conversion between floating point precisions (`FPExt`, `FPTrunc`) and the relational operators $<$, $=$, $>$, $\leq$, $\geq$ and $\neq$. Of particular importance for our crosschecking algorithm (§ 4.3) is the fact that relational operators can occur in both *ordered* and *unordered* form. Ordered and unordered operators differ in the way they treat `NaN` values: if any operand is a `NaN`, ordered comparisons always evaluate to `false` while unordered ones to `true`.

A comparison of two floating point values x and y must have one of four mutually exclusive outcomes: $x < y$, $x = y$, $x > y$ or x `UNO` y (either or both of x and y

Shorthand	FCmp operation	Meaning
$\text{FOeq}(X, Y)$	$\text{FCmp}(X, Y, \{=\})$	Ordered $=$
$\text{FOlt}(X, Y)$	$\text{FCmp}(X, Y, \{<\})$	Ordered $<$
$\text{FOle}(X, Y)$	$\text{FCmp}(X, Y, \{<, =\})$	Ordered $\leq$
$\text{FUno}(X, Y)$	$\text{FCmp}(X, Y, \{\text{UNO}\})$	Unordered test

Table 1. Floating point predicate shorthand semantics.

are `NaN`). We establish a set $\mathbf{O} = \{<, =, >, \text{UNO}\}$ of these outcomes. Then, any floating point relational operator may be represented by a subset of $\mathbf{O}$: for example, ordered $\leq$ (`FOle`) is represented by $\{<, =\}$.

In KLEE-FP, all floating point relational operators are represented using a generic `FCmp` expression. The first two operands to `FCmp` are the comparison operands, whereas the third operand is a subset of $\mathbf{O}$, known as the *outcome set* (represented internally using a vector of four bits, based on the floating point predicate representation used by LLVM [Lattner 2004]). In this paper we normally refer to predicate operations using shorthand names rather than using `FCmp`. Table 1 gives a few examples of mappings between shorthand names and associated `FCmp` operations. In Section 4.4 we show how outcome sets can be used to simplify expressions involving floating-point comparisons.

In future work, we may also wish to store the rounding mode of each non-relational operation. However, we have not yet found this necessary, because none of the code we have worked with changes the rounding mode.

3.2 SSE Vector Operations

Intel's Streaming SIMD Extension operates on a set of eight 128-bit vector registers, called *XMM* registers. Each of these registers can be used to pack together either four 32-bit single-precision floats, two 64-bit double-precision floats, or various combinations of integer values (e.g., four 32-bit ints, or eight 16-bit shorts).

Since the same register set is used to operate on different data types, it is possible to perform an operation of a certain type on the result of an operation of a different type: e.g., one could perform a single-precision computation on the result of a double-precision, or even integer, computation. As a consequence, in order to capture the precise semantics of SSE vector operations, it is important to model SSE registers at the bit-level. Fortunately, KLEE already models its constraints with bit-level accuracy [Cadar 2008] by using the *bitvector* data type provided by its underlying constraint solver, STP [Ganesh 2007]. Thus, we model each XMM register as a 128-bit STP bitvector that can be treated as storing different data types, depending on the instruction that uses the register.

At the LLVM intermediate language level, SSE vectors are represented by 128-bit typed arrays. There are only three generic operations that operate on these arrays: `insertelement`, `extractelement` and `shufflevector`.

#	LLVM intrinsic (llvm.x86.)	# Occurrences in OpenCV	Instruction	Function
1	sse.cmp.ps	19	CMPPS	Compare Packed Single-Precision Floating-Point Values
2	sse.max.ps	4	MAXPS	Return Maximum Packed Single-Precision Floating-Point Values
3	sse.min.ps	6	MINPS	Return Minimum Packed Single-Precision Floating-Point Values
4	sse2.pslli.w	5	PSLLW	Shift Packed Data Left Logical
5	sse2.psubus.b	17	PSUBUSB	Subtract Packed Unsigned Integers with Unsigned Saturation
6	sse2.psubus.w	11	PSUBUSW	

Table 2. Examples of SSE intrinsics supported by KLEE-FP. The entire list consists of 37 intrinsics.

All other SSE instructions are implemented as LLVM intrinsics, as discussed in the next section.

The `extractelement` operation takes as arguments a 128-bit wide array (e.g., an eight element array of 16-bit integers) and an offset into this array, and returns the element at that offset. For example,

%res = **extractelement** <8 x **i16**> %a, **i32** 3

extracts the fourth element of the array a (which contains eight 16-bit shorts) and stores it in `%res`. Similarly,

%res = **insertelement** <8 x **i16**> %a, **i16** 10, **i32** 2

returns in `%res` an array with all values equal to those of the array `%a` except for the third element which receives the value 10.

The `shufflevector` instruction takes two vectors of the same type and returns a permutation of elements from those two vectors. The permutation is specified using an immediate vector argument whose elements represent offsets into the vectors. For example,

%res = **shufflevector** <4 x **float**> %a, <4 x **float**> %b,
   <4 x **i32**> <**i32** 0, **i32** 1, **i32** 4, **i32** 5>

returns in `%res` a vector with its 2 lower order elements taken from the 2 lower order elements of `%a` and its 2 higher order elements from the 2 lower order elements of `%b`.

In our implementation, we model these three operations using the bitvector extraction and concatenation primitives provided by STP. The modeling is straightforward. For example, if A is the 128-bit bitvector representing the array a, $\text{Extract}^{16}(A, 48)$ is the bitvector expression encoding the `extractelement` operation above, where $\text{Extract}^W(BV, k)$ extracts a bitvector of size W starting at offset k of bitvector BV.

3.3 SSE Intrinsic Lowering

Not all SSE instructions are implemented in terms of vector operations; most of them are represented using LLVM intrinsics. To enable comparison with scalar code, we implemented a pass that translates them into standard LLVM instructions by making use of the `extractelement` and `insertelement` operations presented in Section 3.2.

We added support for 37 SSE intrinsics; Table 2 shows a few examples. These 37 intrinsics were sufficient to handle the OpenCV benchmarks on which we evaluated our technique (§ 5). An example of a call to an SSE-specific intrinsic is shown below:

%res = **call** <8 x **i16**> @llvm.x86.sse2.pslli.w (
        <8 x **i16**> %arg, **i32** 1)

This instruction shifts every element of `%arg` left by 1 yielding `%res`. The lowering pass transforms this call into the following sequence of instructions:

%1 = **extractelement** <8 x **i16**> %arg, **i32** 0
%2 = **shl i16** %1, 1
%3 = **insertelement** <8 x **i16**> **undef**, **i16** %2, **i32** 0
%4 = **extractelement** <8 x **i16**> %arg, **i32** 1
%5 = **shl i16** %4, 1
%6 = **insertelement** <8 x **i16**> %3, **i16** %5, **i32** 1
 ...
%22 = **extractelement** <8 x **i16**> %arg, **i32** 7
%23 = **shl i16** %22, 1
%res = **insertelement** <8 x **i16**> %21, **i16** %23, **i32** 7

These instructions carry out the same task as the intrinsic but are expressed in terms of the standard LLVM instructions `insertelement`, `extractelement` and `shl`.

3.4 Atomic Intrinsics

LLVM provides a number of intrinsics which are used to represent atomic operations. Since our OpenCV benchmarks use atomic operations, we needed to add support for them to KLEE-FP.

An example of such an LLVM atomic intrinsic is the following:

%res = **call i32** @llvm.atomic.load.add.i32.p0i32 (
        i32∗ %ptr, **i32** 1)

This operation atomically loads a 32-bit integer from the given memory pointer `%ptr`, increments it, stores the result to `%ptr` and returns the value originally loaded from `%ptr` in `%res`.

Since KLEE does not support threading or signals, KLEE-FP uses a very simple work-around for atomic operations: it simply lowers them to equivalent sequences of non-atomic instructions. For example, the atomic operation shown above is translated to:

%res = **load i32**∗ %ptr
%1 = **add i32** %res, 1
store i32 %1, **i32**∗ %ptr

Our atomic lowering pass handles all 13 atomic intrinsics supported by LLVM 2.7, and was subsequently contributed to the main LLVM branch to be used by similar tools.

4. Crosschecking Algorithm

Crosschecking an SIMD routine against its scalar equivalent using our technique involves four main stages. First, we write a test harness that invokes the scalar and SIMD versions of the code on the same symbolic input, and asserts that their results are equal. For example, the `main()` function in Figure 1 represents the test harness for our simple `zlimit` benchmark. Second, we use *symbolic execution* to explore all the feasible paths in the code under test (§4.1). To increase the applicability of symbolic execution, we apply an aggressive version of *phi node folding* to statically merge paths (§4.2), which reduces the number of paths we have to track by an exponential factor on some benchmarks. Then, on each explored path, we try to prove that the symbolic expressions corresponding to the scalar and SIMD variants are equivalent. To do so, we first canonicalize the expressions through a series of expression rewrite rules and analyses (§4.4), and then use expression matching and constraint solving to prove that the resulting expressions are equivalent (§4.3).

4.1 Symbolic Execution

KLEE-FP uses symbolic execution [King 1975] to explore all the feasible paths in a program up to a certain input size. Symbolic execution runs the program on a *symbolic* input, whose value is initially unconstrained. As the program runs, it tracks the constraints on each symbolic memory location. If code uses a symbolic expression in a conditional, it follows both outcomes of the branch (if both are possible), constraining the conditional expression to be `true` on the true path and `false` on the other. Each of the two paths is explored in the same way, forking execution whenever both sides of a conditional expression are possible.

There are two fundamental limitations of symbolic execution which are relevant to this work:

1. It does not handle symbolically-sized objects. Thus, for code that uses arbitrarily-sized data structures, we can only verify the *bounded* equivalence of SIMD and scalar versions, i.e. we can verify they are equivalent up to a certain input size.

2. The number of paths in a program is in general exponential in the number of branches encountered during execution, thus for some programs, symbolic execution may fail to explore all feasible paths in a practical amount of time even for small input sizes. To reduce the number of explored paths, we discuss in Section 4.2 an approach for statically merging paths via phi-node folding.

In our work, we use the symbolic execution tool KLEE, which is built on top of the LLVM compiler infrastructure.
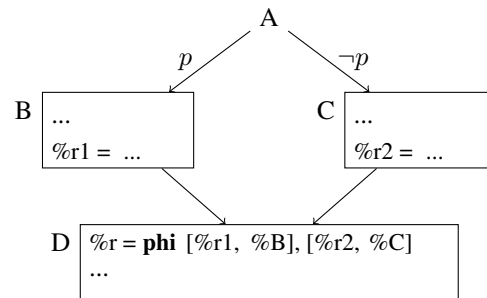

Figure 3. Diamond control flow pattern.

We found KLEE to be a good match for implementing our technique because it handles C/C++ code, tracks constraints with bit-level accuracy, and provides an easily extensible expression language [Cadar 2008].

4.2 Phi Node Folding

To reduce the number of explored paths, we apply a more aggressive variant of *phi-node folding* (also known as *if-conversion*) [Chuang 2003, Lattner 2004], which attempts to statically merge program paths.

Phi-node folding usually operates on the static single-assignment (SSA) form of a program [Alpern 1988] and targets branches with a control flow structure matching the diamond pattern shown in Figure 3, commonly associated with `if` statements and the C ternary operator. The beginning of block D contains one or more *phi* nodes, which select the correct register values (in our example, that of `%r`) depending on what block was previously executed.

We can reduce the amount of branching in a program by merging all four basic blocks in a diamond pattern into a single block. This is accomplished by unconditionally executing blocks B and C and using the branch predicate p to select the result via `select` instructions.

The traditional application of phi-node folding in compilers has both *safety* and *performance* restrictions. Because blocks B and C are executed unconditionally, it is only safe to perform the transformation if neither block contains an instruction that may throw an exception or cause any other side effects. Most arithmetic instructions satisfy these constraints. However, floating point instructions do not, because they may throw an exception if either operand is a `NaN`. Furthermore, the transformation is only performed when folding is cheap enough, in order to minimize the amount of unnecessary work done by the CPU.

Due to forking, the cost of not applying the optimization in a symbolic execution context is usually greater than that of applying it. Furthermore, since KLEE-FP's crosschecking algorithm (§4.3) and expression canonicalization rules (§4.4) do not interfere with the side effects associated with floating point expressions, is it is always safe to fold floating point instructions in KLEE-FP.

Thus, we have adapted phi-node folding to aggressively merge paths when we encounter the diamond pattern shown in Figure 3. Our implementation is built on top of LLVM's SimplifyCFG pass, and always merges paths regardless of the costs of blocks B and C.

4.3 Crosschecking of Floating Point Expressions

On every path explored via symbolic execution, KLEE-FP tries to prove that the symbolic floating-point expressions associated with the scalar and the SIMD implementations are equivalent.

Proving that two floating point expressions are equivalent involves two main steps. First, KLEE-FP applies a series of expression rewrite rules that aim at bringing each expression to a simple canonical form. These transformations include, among others, category analysis, identity reduction, folding of bitwise operations, and concat merging, and are discussed in detail in Section 4.4.

After these canonicalization rules are applied, KLEE-FP determines if the two normalized expressions are equivalent by using a simple expression matching algorithm. Starting at the root of each expression, KLEE-FP recursively compares pairs of subtrees from the two expressions. For integer subtrees, the STP constraint solver is used to determine the equivalence of the two subtrees. On the other hand, for floating point subtrees, the algorithm does not use the semantics of the floating point expressions themselves, which are instead treated as abstract binary functions. While this may not work very well for integers, it is a good fit for floating point — unlike integer arithmetic, constructing two equivalent values from the same inputs in floating point can usually only be done reliably by performing the same operations.

If the matching algorithm fails to prove expression equivalence, we try to substitute rewritten constraints that are implied by the original constraints (i.e., they impose fewer constraints on the input). This has the important property that no false negatives are produced, i.e., that there are no undetected errors. Any input that invalidates the original equivalence will also invalidate the less constrained rewritten one.

One important way in which we use this idea is in handling expressions of the form $\text{FPToSI}(X)$ and $\text{FPToUI}(X)$ (conversion from floating point to integer). Each expression of this form is substituted by an unconstrained symbolic integer variable. While a new variable is created for each unique expression of this type, identical expressions are substituted with references to the same variable. After the substitution, we can use our constraint solver STP to determine if the rewritten integer expressions are equivalent. If this is the case, then we know the original expressions are also equivalent. However, if the constraint solver cannot prove the equivalence, the mismatch could be a false positive.

4.4 Symbolic Expression Canonicalization

The expression canonicalization rules presented in this section are essential to the success of our expression matching approach. Their main goal is to bring expressions to a simplified normal form, in which they are easier to compare.

Table 3 lists the main rewrite rules we implemented. The first ten are specifically targeted to floating point expressions, while the other eight are applicable to both floating point and integer ones. The remainder of this section discusses these rules in more detail.

1. *Floating point relational operators*

As explained in Section 3.1, each floating point relational operator has an associated outcome set. Rules 1–3 apply simplifications to boolean And, Or and Not operators by manipulating the outcome set. For example, $\text{Or}(\text{FOlt}(X, Y), \text{FOeq}(X, Y))$ simplifies to $\text{FOle}(X, Y)$.

Rules 4–6 implement similar simplifications, making use of the swap function defined below:

$$\begin{aligned} \text{If } o \cap \{<, >\} = \{>\}, & \quad \text{swap}(o) = (o \setminus \{>\}) \cup \{<\} \\ \text{If } o \cap \{<, >\} = \{<\}, & \quad \text{swap}(o) = (o \setminus \{<\}) \cup \{>\} \\ \text{Otherwise} & \quad \text{swap}(o) = o \end{aligned}$$

2. *Category analysis*

Category analysis, a simplified form of interval analysis [Moore 1959], affords us a crude means of expression optimization using a simple abstract interpretation of the semantics of certain floating point expressions. We establish a category set $\mathbf{C} = \{\text{NaN}, -\infty, -, 0, +, +\infty\}$ which covers all categories of floating point values (NaN values, negative infinity, negative values except negative zero/infinity, positive or negative zero, positive values except positive zero/infinity, and positive infinity). The category set $\text{cat}(x) \subseteq \mathbf{C}$ of an expression x is defined as the set of categories the expression x may be in. We define $\text{cat}(x)$ recursively based on the category sets of subexpressions of x. For example, if $+ \in \text{cat}(x)$ and $+ \in \text{cat}(y)$ then $\{+, +\infty\} \subseteq \text{cat}(x + y)$. Our system is capable of computing an accurate category set for most floating point expressions.

Category sets are used to simplify and normalize floating point relational operations. For example, if $\text{cat}(x) = \{0, -\}$ and $\text{cat}(y) = \{0, +\}$ then both $x > y$ and $x \text{ UNO } y$ are infeasible. Therefore $x > y$ is simplified to false, $x \le y$ to true and $\neg(x < y)$ (unordered $\ge$) is normalized to $x = y$.

3. *Floating point equality comparison*

SSE code sometimes performs integer comparisons by first converting to floating point format. This may be due to combining floating point and integer comparisons in a single expression. An example of this is found in the OpenCV routine `cvUpdateMotionHistory` in the `silhouette` benchmark, which converts an integer vector to a floating point vector s0, compares the elements to 0 and performs a logical AND with another operation:

```
__m128 s0 = _mm_cvtepi32_ps (...);
__m128 fz = _mm_setzero_ps ();
__m128 m0 = _mm_and_ps(_mm_xor_ps(v0, ts4),
                       _mm_cmpneq_ps(s0, fz));
```

#	Condition	Expression	Result	Section
1	-	$\mathtt{And}(\mathtt{FCmp}(X,Y,O_1), \mathtt{FCmp}(X,Y,O_2))$	$\mathtt{FCmp}(X,Y,O_1 \cap O_2)$	
2	-	$\mathtt{Or}(\mathtt{FCmp}(X,Y,O_1), \mathtt{FCmp}(X,Y,O_2))$	$\mathtt{FCmp}(X,Y,O_1 \cup O_2)$	
3	-	$\mathtt{Eq}(\mathtt{FCmp}(X,Y,O), \mathtt{false})$	$\mathtt{FCmp}(X,Y,\mathbf{O} \setminus O)$	§4.4(1)
4	$O \cap \{<,>\} = \{>\}$	$\mathtt{FCmp}(X,Y,O)$	$\mathtt{FCmp}(Y,X,\mathtt{swap}(O))$	
5	-	$\mathtt{And}(\mathtt{FCmp}(X,Y,O_1), \mathtt{FCmp}(Y,X,O_2))$	$\mathtt{FCmp}(X,Y,O_1 \cap \mathtt{swap}(O_2))$	
6	-	$\mathtt{Or}(\mathtt{FCmp}(X,Y,O_1), \mathtt{FCmp}(Y,X,O_2))$	$\mathtt{FCmp}(X,Y,O_1 \cup \mathtt{swap}(O_2))$	
7		Category analysis		§4.4(2)
8	C constant, see §4.4(3)	$\mathtt{FOeq}(\mathtt{SIToFP}(X),C)$	$\mathtt{Eq}(X,\mathtt{FPToSI}(C))$	§4.4(3)
9	C constant, see §4.4(3)	$\mathtt{FOeq}(\mathtt{UIToFP}(X),C)$	$\mathtt{Eq}(X,\mathtt{FPToUI}(C))$	
10	$f \in \{\mathtt{FPToSI}, \mathtt{FPToUI}\}$	$f(\mathtt{FPExt}(X))$	$f(X)$	§4.4(4)
11	C_1, C_2 constants	$\mathtt{Concat}(C_1, \mathtt{Concat}(C_2, X))$	$\mathtt{Concat}(\mathtt{Concat}(C_1,C_2), X)$	§4.4(5)
12		Partial constant folding with equality		§4.4(6)
13	-	$\mathtt{ZExt}(X)$	$\mathtt{Concat}(0, X)$	
14	-	$\mathtt{And}(\mathtt{SExt}(P^1), X)$	$\mathtt{Select}(P^1, X, 0)$	§4.4(7)
15	C constant	$\mathtt{Shl}^W(X,C)$	$\mathtt{Concat}(\mathtt{Extract}^{W-C}(X,C), 0^C)$	
16	C constant	$\mathtt{LShr}^W(X,C)$	$\mathtt{Concat}(0^C, \mathtt{Extract}^{W-C}(X,0))$	
17	$f \in \{\mathtt{Or}, \mathtt{And}, \mathtt{Xor}\}$, $\mathrm{width}(X_0) = \mathrm{width}(X_1)$	$f(\mathtt{Concat}(X_0,Y_0), \mathtt{Concat}(X_1,Y_1))$	$\mathtt{Concat}(f(X_0,X_1), f(Y_0,Y_1))$	§4.4(8)
18	$f \in \{\mathtt{Or}, \mathtt{And}, \mathtt{Xor}\}$	$\mathtt{Extract}^W(f(X,Y), N)$	$f(\mathtt{Extract}^W(X,N), \mathtt{Extract}^W(Y,N))$	

Table 3. Symbolic expression canonicalization rules. Where necessary, bitwidths of expressions are denoted by superscripts.

The corresponding scalar code performs a straightforward integer comparison of the values here loaded to s0.

Rewrite rules 8 and 9 support such cases by providing a normalization of floating point comparisons to integer comparisons. It is not sound to perform this normalization unless two conditions are met. First, C must be representable in X's type. This means that C must not have a fractional component and must satisfy $-2^{W-1} \le C < 2^{W-1}$ (for signed conversion) or $0 \le C < 2^W$ (for unsigned conversion) where $W = \mathrm{width}(X)$. If C does not meet these requirements, the comparison will always yield $\mathtt{false}$.

Second, X must not be subject to rounding if it is to match C. If X could be rounded, then the comparison would match multiple values of X. For example, using the IEEE single precision format, with a 23-bit mantissa, the values 2^{24} and $2^{24} + 2$ have adjacent representations. If X were $2^{24} + 1$ it would be rounded to $2^{24} + 2$ during integer to floating-point conversion and would match a C of that value. We must therefore require that $|C| < 2^{M+1}$ where M is the mantissa bitwidth of C's type.

4. *Removing unnecessary FPExt operations*

Transformation rule 10 eliminates redundant floating-point extensions (e.g., from `float` to `double`) where the result is coerced to integer.

5. *Folding* Concat *sequences*

Rule 11 performs constant folding on sequences of `Concat` operations. For example, $\mathtt{Concat}(11, \mathtt{Concat}(00, X))$ gets simplified to $\mathtt{Concat}(1100, X)$.

6. *Partial constant folding with equality*

Given an expression of the form $\mathtt{Eq}(C, \mathtt{Concat}(X,Y))$ where C is a constant, if either X or Y is constant then we compare the higher order bits of C to X (or the lower order bits to Y). If the bits are not equal, we can safely replace the entire expression with $\mathtt{false}$. If the bits are equal, we replace the expression with an equality comparison of either the lower order bits of C with Y (if X constant) or the higher order bits of C with X (if Y constant).

7. *Simple normalization rules*

Rules 13–16 implement simple expression transformations via which certain bit-level operations are rewritten using `Concat`, `Extract` and `Select`. For example, a shift left on W bits by a constant amount C can be rewritten as an extract of length $W - C$ from offset C concatenated with C zero bits.

8. *Folding and unfolding of bitwise operations*

Rewrite rule 17 implements folding of bitwise operations through `Concat` to take advantage of partial constant folding. For example, if $f = \mathtt{And}$ and $X_0 = 0$ then X_1 can be completely eliminated since $\mathtt{And}(0, X_1)$ reduces to 0.

Note that this rewrite rule can also be applied if any of the operands to the bitwise operation is a constant expression, by treating the constant as a `Concat` of two smaller constants.

Rewrite rule 18 implements a similar transformation that unfolds the `Extract` of a bitwise operation to take advantage of partial constant folding. For example, if $W = 2$, $N = 0$, $f = \mathtt{Or}$ and $Y = 1100$, then the rule will simplify the entire expression to bitvector 00.

Source File (`src/`)	Benchmarks	# SIMD	Cov.
`cv/cvcorner.cpp`	eigenval harris	44	100%
`cv/cvfilter.cpp`	filter	1332	0%
`cv/cvimgwarp.cpp`	remap resize warpaff	1070	74.6%
`cv/cvmoments.cpp`	moments	35	100%
`cv/cvmorph.cpp`	morph	1220	43.6%
`cv/cvmotempl.cpp`	silhouette	43	100%
`cv/cvpyramids.cpp`	pyramid	125	44.0%
`cv/cvstereobm.cpp`	stereobm	270	53.3%
`cv/cvthresh.cpp`	thresh	238	100%
`cxcore/cxmatmul.cpp`	transcf.43 transsf.43 transff.43 transff.44	352	100%

Table 4. OpenCV code we tested with KLEE-FP. Coverage data refers to coverage of SIMD instructions, where an SIMD instruction is any instruction of vector type, any `extractelement` instruction, stores of vector operand type, casts from vector type and SSE intrinsics (name begins `llvm.x86.mmx`, `llvm.x86.sse` or `llvm.x86.ssse`).

5. Evaluation

We evaluated our technique on a set of benchmarks that compare scalar and SIMD variants of code developed independently by third parties. The codebase that we selected was OpenCV 2.1.0, a popular C++ open source computer vision library, initially developed by Intel, and now an open-source project available under a BSD license [Bradski 2008, Intel].

Although we had to make some changes to OpenCV for compatibility with KLEE-FP, these were minimal—they either replaced inline assembly code, which KLEE does not support, or disabled some functionality unrelated to the SSE code under test, but which KLEE had trouble executing.

Our benchmarks test a substantial amount of SSE code in OpenCV. Due to time constraints, out of the twenty OpenCV source code files containing SSE code, we arbitrarily selected ten files for testing with KLEE-FP. To build benchmarks, we had to acquire a (brief) understanding of how to invoke each OpenCV algorithm in order to build a test harness similar to that in Figure 1. Section 5.3 provides more details regarding the manual effort involved in constructing a test harness.

Table 4 presents the ten files we tested, together with a list of benchmarks for that code and coverage data. Each of our benchmarks tests one of the algorithms provided by OpenCV. For example, `harris` tests the Harris corner detection algorithm, which finds a *corner* in a given image, intuitively a window that produces large variations when moved in any direction [Bradski 2008]. Each benchmark takes a number of parameters, including the size and format of the

input and output images (represented by matrices) and the specific algorithm to test (for example, the `morph` benchmark can test an `erode` algorithm, which returns in each cell of the output matrix the minimum value of the corresponding cell in the input matrix and its neighbors, and a `dilate` algorithm which instead takes the maximum).

Since we are unable to use symbolically sized images (see §4.1), our methodology was instead to test each benchmark on all possible image sizes up to 16×16 pixels. More precisely, we start with the minimum size for which an SSE variant of the algorithm under test exists (usually 4×1 pixels), and test all possible sizes until we reach images of 16×16 pixels or are unable to test any further due to the high complexity of the generated queries.

The SIMD instruction count for each source file gives a rough approximation of the overall complexity of the SSE code tested by our benchmarks. While it does not necessarily follow that the equivalent scalar code or the surrounding control flow is of similar complexity, we found the SIMD instruction count to be a good metric for the complexity of the computational routines of interest to us.

Some coverage numbers do not reach 100%. We found that this was generally caused by the presence of unrolled SSE code that was unreachable due to query complexity. The `filter` benchmark has 0% coverage because we weren't able to run it at all. We discuss the reasons in §5.3.

We constructed a total of 58 benchmarks to cover the functions in these ten files. KLEE-FP was able to successfully verify 41 benchmarks up to a certain image size (§5.1) and find mismatches in 10 benchmarks (§5.2). In addition, three benchmarks triggered false positives (§5.3(2)) and four benchmarks couldn't be run at all by KLEE-FP (§5.3(3)).

5.1 Benchmarks verified up to a certain image size

Table 5 presents the list of benchmarks and associated parameters that we were able to verify using KLEE-FP up to a certain image size. The Format column shows the format of the input and output images in terms of the data type (`f` = floating point, `s` = signed integer, `u` = unsigned integer) and the bitwidth of the format. The Max Size column shows the maximum image size we tested using our methodology. Sizes of the form $X \rightarrow Y$ indicate that the benchmark's input and output images are of different sizes: X is the maximum input image size, and Y the maximum output image size that we tested. The K column is used by the `morph` benchmark, which contains two variants of its algorithm: one for rectangular kernels (represented by R) and one for non-rectangular kernels (represented by NR).

The `transff`, `transsf` and `transcf` benchmarks use fixed size matrices. The `.43` variants take a 3-channel source array of size 4×4 and a 1-channel transformation matrix of size 3×4 and produce a 3-channel array of size 4×4, while the `.44` variants take a 4-channel source array of size 4×4 and a 1-channel transformation matrix of size 4×4 and produce a 4-channel array of size 4×4.

#	Bench	Algo	K	Format	Max Size
1	morph	dilate	R	u8	5×5
2				s16	16×16
3				u16	16×16
4			NR	u8	8×3
5				s16	16×16
6				u16	16×16
7				f32	15×15
8		erode	R	u8	4×4
9				s16	16×16
10				u16	16×16
11			NR	s16	16×16
12				u16	16×16
13	pyramid			u8	$8 \times 2 \rightarrow 4 \times 1$
14	remap	nearest neighbor		u8	16×16
15				s16	16×16
16				u16	16×16
17				f32	16×16
18		linear		u8	16×16
19				s16	16×16
20				u16	16×16
21				f32	16×16
22		cubic		u8	16×16
23				s16	16×16
24				u16	16×16
25				f32	16×16
26	resize	linear		s16	$8 \times 8 \rightarrow 8 \times 8$
27				f32	$8 \times 8 \rightarrow 8 \times 8$
28		cubic		s16	$8 \times 8 \rightarrow 8 \times 8$
29				f32	$8 \times 8 \rightarrow 8 \times 8$
30	silhouette			u8 f32	16×16
31	thresh	BINARY		u8	16×16
32				f32	16×16
33		BINARY_INV		u8	16×16
34				f32	16×16
35		TRUNC		u8	16×16
36		TOZERO		u8	16×16
37				f32	16×16
38		TOZERO_INV		u8	16×16
39				f32	16×16
40	transff.43			f32	See §5.1
41	transff.44			f32	See §5.1

Table 5. OpenCV benchmarks verified up to a certain size.

The `remap` benchmark tests the `cvRemap` routine, which performs symbolic conditional branching over the data contained in two of its three input matrices. Because the phi node folding pass is unable to simplify this branching structure, exponential forking results. Our compromise for this benchmark is to supply two concrete matrices and one symbolic matrix to `cvRemap`.

As mentioned before, we ran each benchmark on matrices of up to 16×16 pixels or until we were unable to test any further due to the high complexity of the generated queries. While these are relatively small matrices, our results should be viewed in combination with the SIMD coverage data which shows that the image sizes we tested cover most SIMD code.

We evaluated our phi node folding technique (§4.2) by running our benchmarks both with and without this optimization enabled, and measuring the amount of branching. We found that phi-node folding was essential for two of our benchmarks, `silhouette` and `morph` (which itself encompasses a large number of algorithm/format combinations). In both cases, we were able to merge all program branches into a single large `select` expression. This in turn decreased the number of paths explored by KLEE-FP by an exponential factor of the number of elements in the input image. E.g., for the largest image we tested in the `morph` benchmarks, sized 16×16, the number of paths decreased from approximately 2^{256} paths (according to our theoretical calculations) to 1.

We measured the execution time taken by KLEE-FP for all of our experiments. However, because we ran our benchmarks on a heterogeneous cluster of machines, these times are mainly intended to give a rough idea of the computational cost involved in using our tool. The runtime of individual experiments (i.e., one benchmark run with a single matrix size) varied between less than one second to more than 40 hours. The total cumulative execution time per benchmark (i.e., for all matrix sizes) ranged from only a few seconds (for the `transff` benchmarks, which only work with a fixed matrix size) up to 27 days for `morph` (dilate, R, u16). Approximately 21.1% of benchmarks had cumulative execution times of under ten minutes, 34.2% between ten minutes and one hour, 18.4% between one and twelve hours, and 26.3% over twelve hours.

5.2 Invalidated Benchmarks

Table 6 presents the list of benchmarks in which we found mismatches between the scalar and SSE implementations. Each mismatch was detected by KLEE-FP in less than 30 seconds.

We discuss each of the mismatches found below:

1. **eigenval and harris:**

Both the `eigenval` and `harris` benchmarks compute certain values in double precision in the scalar implementation, which are computed in single precision in the SSE implementation. To determine whether this was the only difference between the implementations, we modified the scalar implementation to use single precision by replacing `double` with `float` and casting to single precision where appropriate (in C, a binary operation taking two floating point values promotes the lower precision operand to the type of the higher precision operand [Int 1999]).

This modification caused `eigenval` to pass our tests, but there was a further issue with `harris` regarding associativity. The scalar implementation of `eigenval` computes the expression `((float)k)*(a + c)*(a + c)`, which the SSE code computes as `_mm_mul_ps(_mm_mul_ps(t, t),`

#	Benchmark	Algorithm	K	Format	Size	Description
1	`eigenval`			`f32`	4×4	Precision
2	`harris`			`f32`	4×4	Precision, associativity
3		dilate	R	`f32`	4×1	
4	`morph`	dilate	NR	`f32`	4×1	Order of min/max operations
5		erode	R	`f32`	4×1	
6	`thresh`	`TRUNC`		`f32`	4×4	
7	`pyramid`			`f32`	$16 \times 2 \to 8 \times 1$	Associativity, distributivity
8	`resize`	linear		`u8`	$4 \times 4 \to 8 \times 8$	Precision
9	`transsf.43`			`s16 f32`	See §5.1	Rounding issue
10	`transcf.43`			`u8 f32`	See §5.1	Integer/FP differences

Table 6. OpenCV benchmarks in which we found mismatches between the scalar and the SSE versions.

`k4`), where the variable `t` initially holds the four $a+c$ values, and `k4` holds four copies of k.

The IEEE floating point operations $+$ and $\times$ are not associative, so these two expressions are not equivalent. The associativity issue may not be immediately obvious, but because `*` in C is left associative [Int 1999], the scalar multiplication is implicitly bracketed as `(((float)k)*(a + c))*(a + c)`, which is clearly not equivalent to the SSE version. The discrepancy is also revealed by KLEE-FP, which is capable of printing the symbolic expressions involved. In this case, KLEE-FP outputs the following expressions, where N_0 and N_{65} are complex subexpressions shared between the two expressions:

$$\textbf{SIMD}: \quad N_0 - ((N_{65} \times N_{65}) \times 0.04)$$
$$\textbf{Scalar}: \quad N_0 - ((0.04 \times N_{65}) \times N_{65})$$

As it can be seen, the KLEE-FP encoding of the operation, which provides explicit bracketing, makes associativity errors such as this much easier to spot.

2. **morph (f32) and thresh (`TRUNC`, f32)**

Both benchmarks involve floating point `min` and/or `max` operations. The SSE and scalar variants of the implementations apply `min` and `max` to the same operands but in a different order. We cannot consider the two expressions to be equivalent because the implementations of `min` and `max` used by the benchmarks are neither associative nor commutative.

The canonical way of expressing a floating point `min` or `max` operation, which is employed by the SSE instructions `MINPS` and `MAXPS`, is:

$$\text{min}(X, Y) = \text{Select}(\text{FOlt}(X, Y), X, Y)$$
$$\text{max}(X, Y) = \text{Select}(\text{FOlt}(Y, X), X, Y)$$

The STL functions `std::min` and `std::max` used by the scalar variants of the benchmarks are not required by the C++ 2003 standard [Int 2003] to be implemented in any specific way (the result is undefined if any of the operands is `NaN`), and the GNU STL implements them with the operand order reversed:

$$\text{stl\_min}(X, Y) = \text{min}(Y, X)$$
$$\text{stl\_max}(X, Y) = \text{max}(Y, X)$$

To see why the operations are not commutative, consider the evaluation of $\text{min}(X, Y)$ where one of the operands is `NaN` and the other is not `NaN`. In this case, the condition would always evaluate to `false` and Y is always returned regardless of which operand is `NaN`. A similar result can be drawn for `max`.

To see why the operations are not associative, consider $\text{min}(\text{min}(X, \text{NaN}), Y)$ and $\text{min}(X, \text{min}(\text{NaN}, Y))$. As we have seen $\text{min}(X, \text{NaN})$ evaluates to `NaN` and $\text{min}(\text{NaN}, Y)$ to Y so the expressions reduce to Y and $\text{min}(X, Y)$ respectively.

3. **pyramid (f32)**

The SIMD variant of this code produces radically different symbolic expressions than the scalar variant. To give an example, we show below an expression extracted from the scalar variant of the algorithm:

$$((N_0 + N_0) + (N_0 + N_0)) + ((N_3 + N_0) \times 4.0)$$

The corresponding SSE expression at the same position is:

$$(((N_0 \times 6.0) + (N_3 \times 4.0)) + N_0) + N_0$$

N_0 and N_3 are complex subexpressions shared between the two expressions. To rearrange the first form into the second would require not only associativity but distributivity properties. Because the IEEE floating point $+$ and $\times$ are neither associative nor distributive, the equality does not hold.

4. **resize (linear, u8)**

The scalar variant of this code produces expressions of the form (simplified to remove irrelevant saturation checks):

$$(((1536 \times N_0) + (512 \times N_0)) + 2097152) >> 22$$

whereas the SIMD variant produces expressions of the form:

$$(2 + (((1536 \times (N_0 >> 4)) >> 16) +$$
$$((512 \times (N_0 >> 4)) >> 16))) >> 2$$

All intermediate values are 32 bits. The SIMD variant loses 11 bits of precision through right shifts before the addition operation, while the scalar variant retains all precision until the final right shift. This leads to differences where the lower 11 bits of N_0 affect the upper 10 bits of the addition result.

5. **transsf.43**

The scalar variant of this code performs a rounds-to-nearest floating-point to unsigned 16-bit integer conversion. Because of the CPU's lack of support for floating-point to unsigned integer conversion, the conversion is performed by converting to a signed 32-bit integer and downcasting. On the other hand, the SIMD variant performs the conversion by first subtracting 32768 from the floating point number, performing a conversion directly to a 16-bit signed integer and adding 32768 to the result. While this may appear correct, it will produce different results in certain edge cases.

For example, consider the value $0.5 + \epsilon$, where ϵ is a value sufficient to shift 0.5 to the next highest floating point representation. If this value is converted directly to an integer, as in the scalar version of the code, the value is rounded up to the nearest integer value, this being 1. On the other hand if we subtract 32768 from the floating point value, as in the SIMD variant of this code, ϵ will be lost during rounding and the result is -32767.5. When this value is converted to an integer, it is rounded *down* to -32768 (under this rounding mode, ties are rounded to the nearest even value), and the result is 0 after adding 32768 back.

6. **transcf.43**

The scalar variant of this code performs floating point calculations whereas the SIMD variant operates over 32-bit fixed point values with 10 bits of precision below the radix point. When the SIMD variant converts the floating point input values into this format, precision can be lost if the floating point exponent is less than 13. This leads to different results where the lower order bits of the floating point input values affect the final result.

We reported the mismatches we found to the OpenCV developers. At the time of this writing, we have received an answer for five out of the ten mismatches listed in Table 6. The developers confirmed the precision and associativity mismatches in the `eigenval` and `harris` benchmarks as real issues and informed us of their intention to fix them. In response to the mismatches in `morph` caused by the different order of min/max operations, we received the following answer:

"I wonder, if your tool can be told to ignore the NaN's in the certain function? Because we never assumed that NaN's are possible in the morphological functions' input data and do not see any reason for such assumption." (Vadim Pisarevsky, personal communication)

As a result, we added an option to KLEE-FP which would ignore the order of min/max operations. The feature was implemented by simply adding another expression

transformation rule to KLEE-FP. With this rule enabled, KLEE-FP was now able to prove the equivalence of the respective benchmarks on images up to 15×15. The tool reported another mismatch on an image of 16×16, which we are currently investigating.

5.3 Applicability and Limitations

Our experimental evaluation has helped us better understand the applicability of our tool, and its main limitations. We have identified three main aspects that developers should be aware of when using KLEE-FP:

1. **Manual effort:** To use our tool, developers have to write a test harness, similar to the one implemented by the `main()` function in Figure 1. This requires the ability to construct the input data structures required to invoke the function under testing, and to identify the output structures that should be compared for equivalence. In the case of code operating on complex, application-specific data structures, this can be a difficult task, especially for people not familiar with the codebase under testing. This is a problem shared with testing in general, and unit testing in particular, and represents the main reason for which we did not have time to test all the SIMD code in OpenCV. However, KLEE-FP is designed as a developer tool, and the software developers familiar with the API of the code under testing would be in a better position to rapidly develop this kind of test harnesses.

2. **False positives:** Because KLEE-FP's verification process is based on expression matching augmented by canonicalization rules, it is prone to false positives, i.e., it can say that two expressions are not equivalent when in fact they are. (However, remember that KLEE-FP has no false negatives, i.e., when it says that two expressions are equivalent, this is guaranteed to be true).

We discovered three false positives in the OpenCV experiments. Two benchmarks—namely `resize` (linear, u16) and `resize` (cubic, u16)—used query expressions of the form $FPToSI(X)$ or $FPToUI(X)$, which were converted to unconstrained variables (see §4.3). While the variable was unconstrained, the underlying floating point expression X was limited in its range, and STP produced counterexamples for the unconstrained variables outside of their feasible range.

The SSE variant of `resize` (cubic, u8) performed floating point calculations whereas the scalar variant performed integer calculations. Analysis of such expressions would require reasoning about floating point semantics, which KLEE-FP is not capable of.

3. **Symbolic execution and constraint solving limitations:** There were also four benchmarks that we were unable to run at all using KLEE-FP. The `filter` benchmark invoked `malloc` with a symbolic argument. While KLEE is normally able to recover from a symbolic memory allocation using STP to determine the maximum value of the argument, in this case the argument was built from a floating point ex-

pression and KLEE-FP was unable to find a maximum, resulting in an error. The other three benchmarks (`stereobm`, `moments` and `warpaff`) presented queries to STP that were too complex to handle, meaning that they caused STP to run for an unbounded amount of time or consume all available memory.

5.4 KLEE-FP as a Development Tool

Manually translating scalar code into an equivalent SSE version is a difficult process. Due to the restrictions of floating point arithmetic, constructing two equivalent floating point expressions usually requires the same sequence of operations, and as a result, we found that in writing the SSE vectorizations, OpenCV developers try to closely imitate the operations performed by the scalar code. Unfortunately, the process is error-prone, and developers often make invalid assumptions about the properties of floating point arithmetic, such as those related to associativity, distributivity, precision, and rounding. We believe that KLEE-FP could be effectively applied as a development-time tool that would assist programmers with the vectorization process, or with any other optimization task that requires the equivalence of two different code fragments.

We believe the initial feedback we received from the OpenCV developers is consistent with our envisioned use of KLEE-FP as a development tool. Developers would incrementally apply our technique on increasingly bigger inputs until no more mismatches are found and/or they gain enough confidence in their translation. Once a mismatch is found, they would either fix the code and look for more problems, or they would improve the precision of the tool by adding additional expression rewrite rules. To improve the usability of KLEE-FP for the latter scenario, the tool would benefit from the ability to specify additional rules in a higher-level language like the one we use to describe the rules in Table 3.

6. Related Work

SIMD-vectorized code makes intensive use of floating point arithmetic. Previous work on formally verifying floating point programs has used theorem proving [Boldo 2007, Harrison 2007], constraint solving based on approximation with rationals or reals [Holzbaur 1995] and symbolic execution using projection functions over floating point intervals [Botella 2006, Michel 2002]. While promising, these techniques have only been shown to work on very small handcrafted programs. An alternative to formal verification is testing. For example, Aharoni [2003] uses randomized testing coupled with coverage requirements to test floating point programs. Random testing can easily be applied to large applications, but misses corner-case bugs that are common in floating point programs.

Our approach of using symbolic execution combined with expression matching and canonicalization rules has been successfully used in the past to verify code equivalence, e.g., in the context of hardware verification [Clarke 2003], embedded software [Currie 2006], compiler optimizations [Necula 2000] and block cipher implementations [Smith 2008]. The main contribution of this work lies in the techniques for handling floating-point arithmetic and in applying this approach to SIMD vectorizations.

Our application of phi-node folding [Chuang 2003, Lattner 2004] aims at reducing the state space explored by symbolic execution by statically merging paths. Recent work in the area provides alternative approaches that we could apply to reduce the number of paths explored: using compositional dynamic test generation to create function summaries [Godefroid 2007], using read-write sets to track the values accessed by the program [Boonstoppel 2008], or using information partitions to track information flow between inputs [Majumdar 2009].

Automatic vectorization techniques provide an alternative to verifying the correctness of manually written SIMD code [Eichenberger 2004, Larsen 2000, Naishlos 2003]. However, even as these techniques will start to be more widely adopted, the approach presented in this paper can be applied to verify these automatically generated SIMD-vectorizations.

7. Conclusion

SIMD computing is an increasingly popular means of improving the performance of programs by exploiting their data level parallelism. Unfortunately, manually translating scalar code into an equivalent SIMD version is a difficult task, because any programming error may cause the hand-optimized SIMD code to act differently from the original scalar version. In this paper, we introduced KLEE-FP, a novel technique for crosschecking an SIMD implementation against its scalar version, which we believe would be valuable to authors of SIMD code. KLEE-FP was able to successfully crosscheck 51 SSE benchmarks from the popular OpenCV library against their corresponding scalar versions, proving the bounded equivalence of 41 of them, and finding inconsistencies in the other 10 pairs.

8. Availability

KLEE-FP and our OpenCV benchmarks are freely available from our website:

`http://www.pcc.me.uk/~peter/klee-fp/`

Acknowledgments

References

[Aharoni 2003] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel. FPgen - a test generation framework for datapath floating-point verification. In *HLDVT '03*, 2003.

[Alpern 1988] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL'88*, Jan. 1988.

[Boldo 2007] Sylvie Boldo and Jean-Christophe Filliatre. Formal verification of floating-point programs. In *ARITH '07*, 2007.

[Boonstoppel 2008] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, Mar.-Apr. 2008.

[Botella 2006] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.

[Bradski 2008] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[Cadar 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08*, Dec. 2008.

[Chuang 2003] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-predication for light-weight if-conversion. In *CGO 2003*, March 2003.

[Clarke 2003] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *ASP-DAC'03*, Jan. 2003.

[Currie 2006] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34 (1):61–91, 2006.

[Eichenberger 2004] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04*, June 2004.

[Ganesh 2007] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV '07*, July 2007.

[Godefroid 2007] Patrice Godefroid. Compositional dynamic test generation. In *POPL'07*, Jan. 2007.

[Harrison 2007] John Harrison. Floating-point verification. *Journal of Universal Computer Science*, 13(5):629–638, 2007.

[Holzbaur 1995] Christian Holzbaur. clp(q,r) manual rev. 1.3.2. Technical report, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[IEEE Task P754 2008] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[Intel] Intel and Willow Garage. OpenCV 2.1.0: Open source computer vision library. http://opencv.willowgarage.com/.

[Int 1999] *ISO/IEC 9899-1999: Programming Languages—C*. International Organization for Standardization, December 1999.

[Int 2003] *ISO/IEC 14882:2003(E): Programming Languages—C++*. International Organization for Standardization, October 2003.

[King 1975] James C. King. A new approach to program testing. In *ICRS'75*, April 1975.

[klee.llvm.org] klee.llvm.org. KLEE website. http://klee.llvm.org.

[Larsen 2000] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00*, May 2000.

[Lattner 2004] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO '04*, Mar. 2004.

[Majumdar 2009] Rupak Majumdar and Ru-Gang Xu. Reducing test inputs using information partitions. In *CAV '09*, July 2009.

[Michel 2002] Claude Michel. Exact projection functions for floating point number constraints. In *AMAI '02*, 2002.

[Moore 1959] Ramon E. Moore and C. T. Yang. Interval analysis I. Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, USA, 1959.

[Naishlos 2003] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES '03*, Oct.–Nov. 2003.

[Necula 2000] George C. Necula. Translation validation for an optimizing compiler. May 2000.

[Smith 2008] Eric Whitman Smith and David L. Dill. Automatic formal verification of block cipher implementations. In *FMCAD '08*, November 2008.