Secure, Precise, and Fast
Floating-Point Operations on x86 Processors

Ashay Rane, Calvin Lin
Department of Computer Science
The University of Texas at Austin
{*ashay, lin*} @*cs.utexas.edu*

Mohit Tiwari
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
tiwari@austin.utexas.edu

Abstract

Floating-point computations introduce several side channels. This paper describes the first solution that closes these side channels while preserving the precision of non-secure executions. Our solution exploits micro-architectural features of the x86 architecture along with novel compilation techniques to provide low overhead.

Because of the details of x86 execution, the evaluation of floating-point side channel defenses is quite involved, but we show that our solution is secure, precise, and fast. Our solution closes more side channels than any prior solution. Despite the added security, our solution does not compromise on the precision of the floating-point operations. Finally, for a set of microkernels, our solution is an order of magnitude more efficient than the previous solution.

1 Introduction

To secure our computer systems, considerable effort has been devoted to techniques such as encryption, access control, and information flow analysis. Unfortunately, these mechanisms can often be subverted through the use of side channels, in which an adversary, with the knowledge of the program, monitors the program's execution to infer secret values. These side channels are significant because they have been used to discover encryption keys in AES [26], RSA [27], and the Diffie-Hellman key exchange protocol [14], thereby rendering these sophisticated schemes useless.

Numerous side channels exist, including instruction and data caches [27, 26], branch predictors [2], memory usage [12, 35], execution time [31, 4], heat [22], power [15], and electromagnetic radiation [9], but one particularly insidious side channel arises from the execution of variable-latency floating-point instructions [3, 10], in which an instruction's latency varies widely depending on its operands, as shown in Table 1.

Zero	Normal	Subnormal	Infinity	NaN
7	11	153	7	7

Table 1: Latency (in cycles) of the `SQRTSS` instruction for various operands.

Both x86[1] and ARM[2] provide variable-latency floating-point instructions. This variable latency stems from the desire to have graceful floating-point arithmetic behavior, which, as we explain in Section 3, requires the use of so-called *subnormal* values [8], which are processed using special algorithms. Since subnormal values are rare, hardware vendors typically support such values in microcode, so as not to slow down the common case. The resulting difference in instruction latency creates a timing side channel, which has been used to infer cross-origin data in browsers and to break differential privacy guarantees of a remote database [3].

However, variable latency floating-point instructions represent only a part of the problem, since higher level floating-point operations, such as sine and cosine, are typically implemented in software. Thus, the implementation of these floating-point operations can leak secret information through other side channels as well. Depending on the secret values, programs can throw exceptions, thereby leaking the presence of abnormal inputs through termination. Programs can also contain conditional branches, which can leak secrets through the instruction pointer, branch predictor, or memory access count. Finally, programs that index into lookup tables can leak secrets through the memory address trace.

To prevent information leaks in both floating-point instructions and floating-point software, a strong solution should ensure at least four key properties, which correspond to the side channels that we discussed above:

[1] `http://www.agner.org/optimize/instruction_tables.pdf`

[2] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344k/ch16s07s01.html`

(1) fixed-time operations that are independent of secret values, (2) disabled exceptions, (3) sequential control flow, and (4) uniform data accesses that are independent of the value of secret variables. Previous solutions [3, 5] are inadequate because they do not ensure all four properties, are slow, are orders of magnitude less precise, or are difficult to implement.

This paper presents a novel solution that closes side channels arising from both hardware and software implementations of floating point operations, providing all four properties mentioned above. Our compiler-based solution has two components.

The first component creates building blocks of elementary floating-point operations for instructions that are natively supported by the hardware (addition, subtraction, multiplication, division, square root, and type conversion). Our solution leverages unused SIMD lanes so that fast operations on normal operands are accompanied by slower dummy computations on subnormal operands, yielding a consistent yet low instruction latency for all types of operands.

The second component is a software library of higher-level floating-point operations like sine and cosine. The key to creating this second component is a new code transformation that produces fixed-latency functions through normalized control flows and data access patterns. Code generated by our compiler closes *digital side-channels*, which have been defined to be those side channels that carry information over discrete bits [28]. Whereas previous work in closing digital side channels employs a runtime system [28], our solution shifts much of the work to compile time, yielding a significantly smaller runtime overhead.

This paper makes the following contributions:

1. We present a novel compiler-based system, called *Escort*, for closing digital side channels that arise from the processing of floating-point instructions.

2. **Secure:** We demonstrate that our solution is secure not just against timing but also against digital side channels. We demonstrate Escort's capabilities by defeating a machine-learning side-channel attack, by defending against a timing attack on the Firefox web browser, by conducting extensive performance measurements on an x86 processor, and by verifying our solution's code using typing rules.

3. **Precise:** We show that Escort provides precision that is *identical* to that of the standard C math library. By contrast, the previous solution's precision is off by several million floating-point values.

4. **Fast:** We show that our solution is fast. On a set of micro-benchmarks that exercise elementary floating-point operations, Escort is 16× faster than the previous solution [3].

5. As an ancillary contribution, we introduce a methodology for evaluating the precision and security of floating-point operations, which is fraught with subtleties.

The rest of this paper is organized as follows. Section 2 describes our threat model, related work, and system guarantees. We provide background in Section 3 before presenting our solution in Section 4. We evaluate our solution in Sections 5–7 . Finally, we conclude in Section 8.

2 Threat Model and Related Work

This section begins by describing our threat model, which shapes our subsequent discussion of related work and of Escort's security guarantees.

Threat Model. Our goal is to prevent secret floating-point operands from leaking to untrusted principals that either read digital signals from the processor's pins or that are co-resident processes.

We assume that the adversary is either an external entity that monitors *observation-based* side channels (*e.g.* time [14], memory address trace [11], or the `/proc` pseudo-filesystem [12]) or a co-resident process/VM that monitors *contention-based* side channels (*e.g.* cache [27] or branch predictor state [2]).

For off-chip observation-based channels, we assume that the processor resides in a sealed and tamper-proof chip that prevents the adversary from measuring physical side channels like heat, power, electromagnetic radiation, etc. We assume that the CPU encrypts data transferred to and from DRAM. All components other than the processor are untrusted, and we assume that the adversary can observe and tamper with any digital signal. For on-chip contention-based channels, we assume that the OS is trusted and does not leak the victim process's secret information. We also assume that the adversary cannot observe or change the victim process's register contents. Our trusted computing base includes the compilation toolchain.

Side-Channel Defenses. Decades of prior research have produced numerous defenses against side channels, the vast majority of which close only a limited number of side channels with a single solution. For instance, numerous solutions exist that close only the cache side channel [6, 36, 39, 37, 16] or only the address-trace side channel [33, 20, 32, 29]. Raccoon [28] is the first solution that closes a broad class of side channels—in

particular, the set of digital side channels—with a single solution. Similar to Raccoon, Escort also closes digital side channels with a single solution, but unlike Raccoon, Escort focuses on closing floating-point digital side channels, which can arise from variable latency floating-point instructions and from software implementations of floating-point libraries, in which points-to set sizes are typically small. Given Escort's narrower focus on floating-point computations, Escort is faster than Raccoon by an order of magnitude.

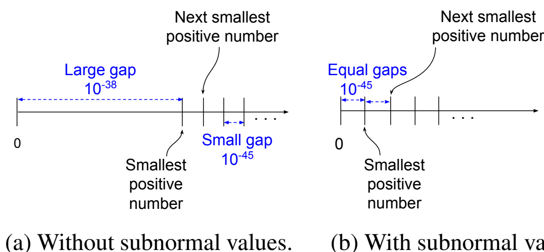
Timing Side-Channel Defenses. Prior defenses against timing side-channel attacks utilize new algorithms [30], compilers [23], runtime systems [21], or secure processors [18]. However, these solutions only address one source of timing variations—either those stem from the choice of the algorithm [31] or those that stem from the microarchitectural design [10]. By contrast, Escort closes timing variations from both sources.

Floating-Point Side-Channel Defenses. Andrysco et al. [3] present `libfixedtimefixedpoint` (FTFP), the first software solution for closing the floating-point timing channel. FTFP has some weaknesses, as we now discuss, but the main contribution of their paper is the demonstration of the significance of this side channel, as they use variable-latency floating-point operations to break a browser's same-origin policy and to break differential privacy guarantees of remote databases. FTFP is a *fixed-point* library that consists of 19 hand-written functions that each operates in fixed time, independent of its inputs. FTFP is slow, it is imprecise, and it exposes secrets through other side channels, such as the cache side channel or the address trace side channel. Cleemput et al. [5] introduce compiler transformations that convert variable-timing code into fixed-timing code. Their technique requires extensive manual intervention, applies only to the division operation, and provides weak security guarantees. Both solutions require manual construction of fixed-time code—a cumbersome process that makes it difficult to support a large number of operations. By contrast, Escort implements a fixed-time floating-point library, while preventing information leaks through timing as well as digital side channels. Escort includes a compiler that we have used to automate the transformation of 112 floating-point functions in the Musl standard C library, a POSIX-compliant C library. Escort also provides precision identical to the standard C library.

Escort's Guarantees. Escort rejects programs that contain unsupported features—I/O operations and recursive function calls. Unlike prior work [18, 28], Escort *does* transform loops that leak information through trip counts. Escort is unable to handle programs containing irreducible control flow graphs (CFGs), but standard compiler transformations [24] can transform irreducible CFGs into reducible CFGs. Escort assumes that the input program does not use vector instructions, does not exhibit undefined behavior, does not terminate abnormally through exceptions, and is free of race conditions. Given a program that abides by these limitations, Escort guarantees that the transformed code produces identical results as the original program, does not leak secrets through timing or digital side channels, and that the transformed code does not terminate abnormally.

3 Background

The variable latency of floating-point instructions creates security vulnerabilities. In this section, we explain subnormal numbers, which are the cause of the variable latency, and we explain the difficulty of fixing the resulting vulnerability. We also explain how the Unit of Least Precision (ULP) can be used to quantify the precision of our and competing solutions.



(a) Without subnormal values. (b) With subnormal values.

Figure 1: Impact of allowing subnormal numbers. Without subnormal values, there exists a much larger gap between zero and the smallest positive number than between the first two smallest positive numbers. With subnormal numbers, the values are more equally spaced. (The figure is not drawn to scale.)

3.1 Subnormal Numbers

Subnormal numbers have tiny exponents, which result in floating-point values that are extremely close to zero: $10^{-45} < |x| < 10^{-38}$ for single-precision numbers and $10^{-324} < |x| < 10^{-308}$ for double-precision numbers. Subnormal values extend the range of floating-point numbers that can be represented, but more importantly, they enable *gradual underflow*—the property that as floating-point numbers approach zero along the number scale, the difference between successive floating-point numbers does not increase[3]. Figures 1a and 1b show the

[3] `https://www.cs.berkeley.edu/~wkahan/ARITH_17U.pdf`

differences between zero and the two smallest positive floating-point numbers. With subnormal numbers, the gap between any two consecutive floating-point values is never larger than the values themselves, thus exhibiting Gradual Underflow. Subnormal numbers are indispensable because gradual underflow is required for reliable equation solving and convergence acceleration [8, 13].

To avoid the added hardware complexity of supporting subnormal numbers, which occur infrequently, vendors typically process subnormal values in microcode, which is orders of magnitude slower than hardwired logic.

The resulting difference in latencies creates a security vulnerability. An adversary that can measure the latency of a floating-point instruction can make reasonable estimates about the operand type, potentially inferring secret values using the timing channel. While subnormal values occur infrequently in typical program execution, an adversary can deliberately induce subnormal values in the application's inputs to enable subnormal operand timing attacks.

3.2 Floating-Point Error Measurement

Unlike real (infinite precision) numbers, floating-point numbers use a limited number of bits to store values, thus making them prone to rounding errors. Rounding errors in floating-point numbers are typically measured in terms of the Unit of Least Precision (ULP) [25]. The ULP distance between two floating-point numbers is the number of distinct representable floating-point numbers between them, which is simply the result of subtracting their integer representations. If the result of the subtraction is zero, the floating-point numbers must be exactly the same.

4 Our Solution: Escort

Escort offers secure counterparts of ordinary non-secure floating-point operations, including both elementary operations and higher-level math operations. The elementary operations include the six basic floating-point operations that are natively supported by the ISA—type conversion, addition, subtraction, multiplication, division, and square root—and a conditional data copy operation. The 112 higher-level math operations are those that are implemented using a combination of native instructions. Examples of higher-level functions include sine, cosine, tangent, power, logarithm, exponentiation, absolute value, floor, and ceiling.

The next subsections describe Escort's design in three parts. First, we describe the design of Escort's secure elementary operations. These operations collectively form the foundation of Escort's security guarantees. Second, we describe Escort's compiler, which accepts non-secure code for higher-level operations and converts it into secure code. This compiler combines a code transformation technique with Escort's secure elementary operations. Third, we present an example that shows the synergy among Escort's components.

4.1 Elementary Operations

The key insight behind Escort's secure elementary operations is that the latencies of SIMD instructions are determined by the slowest operation among the SIMD lanes (see Figure 2), so the Escort compiler ensures that each elementary instruction runs along side a dummy instruction whose operand will produce the longest possible latency. Our analysis of 94 x86 SSE and SSE2 instructions (which includes single- and double-precision arithmetic, comparison, logical, and conversion instructions) reveals: (1) that only the multiplication, division, square root, and single-precision to double-precision conversion (upcast) instructions exhibit latencies that depend on their operands and (2) that subnormal operands induce the longest latency.

In particular, Escort's fixed-time floating-point operations utilize SIMD lanes in x86 SSE and SSE2 instructions. Our solution (1) loads genuine and dummy (subnormal) inputs in spare SIMD lanes of the same input register, (2) invokes the desired SIMD instruction, and (3) retains only the result of the operation on the genuine inputs. Our tests confirm that the resulting SIMD instruction exhibits the worst-case latency, with negligible variation in running time (standard deviation is at most 1.5% of the mean). Figure 3 shows Escort's implementation of one such operation.

Escort includes Raccoon's conditional data copy operation (see Figure 4) which does not leak information through digital side channels. This operation copies the contents of one register to another register if the given condition is true. However, regardless of the condition, this operation consumes a fixed amount of time, executes the same set of instructions, and does not access application memory.

4.2 Compiling Higher-Level Operations

Escort's compiler converts existing non-secure code into secure code that prevents information leakage through digital side channels. First, our compiler replaces all elementary floating-point operations with their secure counterparts. Next, our compiler produces straight-line code that preserves control dependences among basic blocks while preventing instruction side effects from leaking secrets. Our compiler then transforms array access statements so that they do not leak information through memory address traces. Finally, our compiler transforms

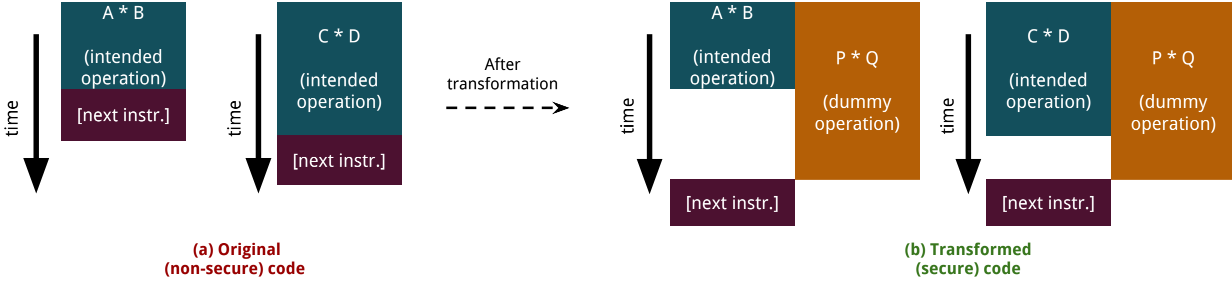


Figure 2: The key idea behind Escort's secure elementary operations. The operation is forced to exhibit a fixed latency by executing a fixed-latency long-running operation in a spare SIMD lane.

```
double escort_mul_dp(double x, double y) {
    const double k_normal_dp = 1.4;
    const double k_subnormal_dp = 2.225e-322;

    double result;
    __asm__ volatile(
            "movdqa     %1, %%xmm14;"
            "movdqa     %2, %%xmm15;"
            "pslldq     $8, %1;"
            "pslldq     $8, %2;"
            "por        %3, %1;"
            "por        %4, %2;"
            "movdqa     %2, %0;"
            "mulpd      %1, %0;"
            "psrldq     $8, %0;"
            "movdqa     %%xmm14, %1;"
            "movdqa     %%xmm15, %2;"
            : "=x" (result), "+x" (x), "+x" (y)
            : "x" (k_subnormal_dp), "x" (k_normal_dp)
            : "xmm15", "xmm14");
    return result;
}
```

Figure 3: Escort's implementation of double-precision multiplication, using the AT&T syntax.

```
01: copy(uint8_t pred, uint32_t t_val, uint32_t f_val) {
02:     uint32_t result;
03:     __asm__ volatile (
04:         "mov    %2, %0;"
05:         "test   %1, %1;"
06:         "cmovz  %3, %0;"
07:         "test   %2, %2;"
08:         : "=r" (result)
09:         : "r" (pred), "r" (t_val), "r" (f_val)
10:         : "cc"
11:     );
12:     return result;
13: }
```

Figure 4: Code for conditional data copy operation that does not leak information over digital side channels. This function returns `t_val` if `pred` is **`true`**; otherwise it returns `f_val`. The assembly code uses AT&T syntax.

loops whose trip count reveals secrets over digital side channels. We now describe each step in turn.

4.2.1 Step 1: Using Secure Elementary Operations

The Escort compiler replaces x86 floating-point type-conversion, multiplication, division, and square root assembly instructions with their Escort counterparts. However, Escort's secure elementary operations can be up to two orders of magnitude slower than their non-secure counterparts. Hence, our compiler minimizes their usage by using taint tracking and by employing the quantifier-free bit-vector logic in the Z3 SMT solver [7], which is equipped with floating-point number theory. If the solver can prove that the operands can never be subnormal values, then Escort refrains from replacing that instruction.

In effect, the Escort compiler constructs path-sensitive Z3 expressions for each arithmetic statement in the LLVM IR. For every Φ-node that produces an operand for an arithmetic expression, Escort creates one copy of the expression for each input to the Φ-node. If the solver reports that no operand can have a subnormal value, then Escort skips instrumentation of that floating-point operation.

We set a timeout of 40 seconds for each invocation of the SMT solver. If the solver can prove that the instruction never uses subnormal operands, then Escort skips replacing that floating-point instruction with its secure counterpart. Figure 5 shows the percentage of floating-point instructions in commonly used math functions that are left untransformed by Escort.

This optimization is conservative because it assumes that all floating-point instructions in the program have subnormal operands unless proven otherwise. The correctness of the optimization is independent of the code's use of pointers, library calls, system calls, or dynamic values. The static analysis used in this optimization is flow-sensitive, path-sensitive, and intra-procedural.

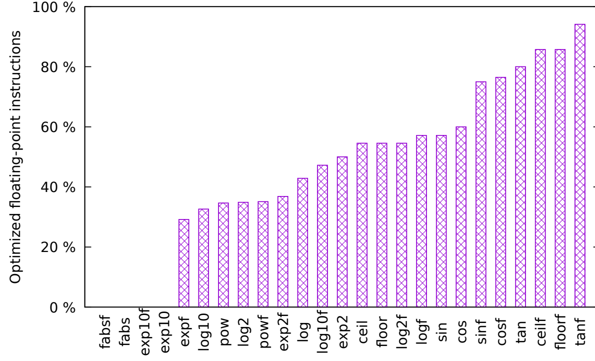


Figure 5: Percentage of instructions that are left uninstrumented (without sacrificing security) after consulting the SMT solver.

4.2.2 Step 2: Predicating Basic Blocks

Basic block predicates represent the conditions that dictate whether an instruction should execute. These predicates are derived by analyzing conditional branch instructions. For each conditional branch instruction that evaluates a predicate p, the Escort compiler associates the predicate p with all basic blocks that execute if the predicate is true, and it associates the predicate $\neg p$ with all basic blocks that execute if the predicate is false. For unconditional branches, the compiler copies the predicate of the previous block into the next block. Finally, if the Escort compiler comes across a block that already has a predicate, then the compiler sets the block's new predicate to the logical `OR` of the input predicates. At each step, the Escort compiler uses Z3 as a SAT solver to simplify predicates by eliminating unnecessary variables in predicate formulas. Figure 6 shows the algorithm for basic block predication.

4.2.3 Step 3: Linearizing Basic Blocks

The Escort compiler converts the given code into straight-line code so that every invocation of the code executes the same instructions. To preserve control dependences, the basic blocks are topologically sorted, and then the code is assembled into a single basic block with branch instructions removed.

4.2.4 Step 4: Controlling Side Effects

We now explain how Escort prevents side effects from leaking secrets. Here, *side effects* are modifications to the program state or any observable interaction, including memory accesses, exceptions, function calls, or I/O. Escort controls all side effects except for I/O statements.

1: **for each** basic block *bb* in function **do**
2:     **if** $entry_block(bb)$ **then**
3:         $pred[bb] \leftarrow$ **true**
4:     **else**
5:         $pred[bb] \leftarrow$ **false**
6:     **end if**
7: **end for**
8:
9: **for each** basic block *bb* in function **do**
10:     $br \leftarrow branch(bb)$
11:     **if** $unconditional_branch(br)$ **then**
12:         $\{s\} \leftarrow successors(bb)$
13:         $pred[s] \leftarrow pred[s] \vee pred[bb]$
14:         $pred[s] \leftarrow simplify(pred[s])$
15:     **else**   ▷ Conditional Branch.
16:         $\{s_1, s_2\} \leftarrow successors(bb)$
17:         **if** $loop_condition_branch(br)$ **then**
18:             ▷ Skip branches that represent loops.
19:             $pred[s_1] \leftarrow pred[s_1] \vee pred[bb]$
20:             $pred[s_2] \leftarrow pred[s_2] \vee pred[bb]$
21:         **else**
22:             $p \leftarrow condition(br)$
23:             $pred[s_1] \leftarrow pred[s_1] \vee (pred[bb] \wedge p)$
24:             $pred[s_2] \leftarrow pred[s_2] \vee (pred[bb] \wedge \neg p)$
25:         **end if**
26:         $pred[s_1] \leftarrow simplify(pred[s_1])$
27:         $pred[s_2] \leftarrow simplify(pred[s_2])$
28:     **end if**
29: **end for**

Figure 6: Algorithm for predicating basic blocks.

Memory Access Side Effects. To ensure proper memory access side effects, the Escort compiler replaces store instructions with conditional data-copy operations that are guarded by the basic block's predicate, so memory is only updated by instructions whose predicate is true.

Unfortunately, this naïve approach can leak secret information when the program uses pointers. Figure 7 illustrates the problem: If store instructions are not allowed to update a pointer variable when the basic block predicate is false, then the address trace from subsequent load instructions on the pointer variable will expose the fact that the pointer variable was not updated.

The Escort compiler prevents such information leaks by statically replacing pointer dereferences with loads or stores to each element of the points-to set[4]. Thus Escort replaces the statement in line 8 (Figure 7) with a store operation on b. When the points-to set is larger than a

[4]Escort uses a flow-sensitive, context-insensitive pointer analysis: `https://github.com/grievejia/tpa`. Replacing a pointer dereference with a `store` operation on all elements of the points-to set is feasible for Escort because points-to set sizes in the Musl C library are very small.

```
1: p ← &a
2: secret ← input()      ▷ Assume input() returns true.
3: if secret = true then
4:     ...
5: else
6:     ...
7:     p ← &b ▷ Instruction does not update pointer p,
   since basic block's execution-time predicate is false.
8:     *p ← 10                 ▷ Accesses a instead of b!
9: end if
```

Figure 7: The use of pointers can leak information. If `store` instructions are not allowed to access memory when the basic block's predicate is `false`, then pointer ***p*** will dereference the address for ***a*** instead of ***b***, thus revealing that *secret* is `true`.

singleton set, Escort uses the conditional data copy operation on all potential *pointees i.e.* the elements of the points-to set. The predicate of the conditional copy operation checks whether the pointer points to the candidate pointee. If the predicate is false, the pointee's existing value is overwritten, whereas if the predicate is true, the new value is written to the pointee.

Function Call Side Effects. Adversaries can observe the invocation of functions (or lack thereof) using side channels like the Instruction Pointer. Thus, a solution incapable of handling function calls will leak information to the adversary. While inlining functions is a potential solution, inlining is impractical for large applications.

Escort handles side effects from function calls by propagating the predicate from the calling function to the callee. Thus, each user-defined function is given an additional argument that represents the predicate of the call site's basic block. The callee ensures correct handling of side effects by `AND`ing its own predicates with the caller's predicate.

Side Effects from Exceptions. Program termination caused by exceptions will leak the presence or absence of abnormal operands. To prevent such information leakage, Escort requires that exceptions not occur during program execution[5].

Escort manages floating-point and integer exceptions differently. Escort requires that the programmer disable floating-point exceptions (*e.g.* using `feclearexcept()`). For integer exceptions, Escort borrows ideas from Raccoon by replacing abnormal operands with benign operands (*e.g.* Escort prevents integer division-by-zero by replacing a zero divisor with a non-zero divisor).

[5] Escort assumes that the input program does not throw exceptions, so masking exceptions does not change the semantics of the program.

4.2.5 Step 5: Transforming Array Accesses

Array index values reveal secrets as well. For instance, if the adversary observes that accesses to `array[0]` and `array[secret_index]` result in accesses to locations 10 and 50, then the adversary knows that `secret_index` = 40. To eliminate such information leaks, the Escort compiler transforms each array access into a linear sweep over the entire array, which hides from the adversary the address of the program's actual array index.

Of course, the transformed code is expensive, but this approach is feasible because (1) math library functions typically use only a few small lookup tables, thus requiring relatively few memory accesses and (2) the processor's caches and prefetchers dramatically reduce the cost of sweeping over the arrays.

4.2.6 Step 6: Transforming Loops

Some loops introduce timing channels because their trip counts depend on secret values. The Escort compiler transforms such loops using predictive mitigation [38]. The loop body executes as many times as the smallest power of 2 that is greater than or equal to the loop trip count. For instance, if the actual loop trip count is 10, then the loop body is executed 16 times. The basic block predicate ensures that dummy iterations do not cause side effects. With this transformed code, an adversary that observes a loop trip count of l can infer that the actual loop trip count l' is between l and $0.5 \times l$. However, the exact value of l' is not revealed to the adversary.

Unfortunately, this naive approach can still leak information. For instance, if two distinct inputs cause the loop to iterate 10 and 1000 times respectively, the transformed codes will iterate 16 and 1024 times respectively—a large difference that may create timing variations. To mitigate this problem, Escort allows the programmer to manually specify the minimum and maximum loop trip counts using programmer annotations. These annotations override the default settings used by the Escort compiler.

4.3 Example Transformation: `exp10f`

We now explain how Escort transforms an example non-secure function (Figure 8a) into a secure function (Figure 8c). To simplify subsequent analyses and transformations, the Escort compiler applies LLVM's `mergereturn` transformation pass, which unifies all exit nodes in the input function (see Figure 8b).

First, the Escort compiler replaces elementary floating-point operations in lines 8 and 10 with their secure counterpart function shown in lines 21 and 22 of the transformed code. Second, using the algorithm shown in Figure 6, the Escort compiler associates predicates with

```
float e10(float x) {
  float n, y = mf(x, &n);
  if (int(n) >> 23 & 0xff < 0x82) {
    float p = p10[(int) n + 7];
    if (y == 0.0f) {
      return p;
    }
    return exp2f(3.322f * y) * p;
  }
  return exp2(3.322 * x);
}
```

(a) Original code for `exp10f()`.

```
01: float e10(float x) {
02:   float n, y = mf(x, &n);
03:   if (int(n) >> 23 & 0xff < 0x82) {
04:     float p = p10[(int) n + 7];
05:     if (y == 0.0f)
06:       result = p;
07:     else
08:       result =
                exp2f(3.322f * y) * p;
09:   } else
10:     result = exp2(3.322 * x);
11:   return result;
12: }
```

(b) Result after applying LLVM's `mergereturn` pass. This code becomes the input for the Escort compiler.

```
12: float e10(float x) {
13:  return e10_cloned(x, true);
14: }
15:
16: float e10_cloned(float x, uint pred) {
17:   float n, y = mf_cloned(x, &n, pred);
18:   float p = write(int(n) >> 23 & 0xff
        < 0x82, stream_load(p10, (int) n + 7]));
19:   bool p2 = y == 0.0f;
20:   write(pred & p1 & p2, p, &result);
21:   write(pred & p1 & !p2,
          escort_mul(
           escort_mul(
             exp2f_cloned(3.322f,
               pred & p1 & !p2),
             y),
           p),
        &result);

22:   write(!p1,
        escort_mul(
          exp2_cloned(3.322, pred & !p1),
          escort_upcast(x))),
        result);

23:   return result;
24: }
```

(c) Result of the Escort compiler's transformation.

Figure 8: Escort's transformation of `exp10f()`.

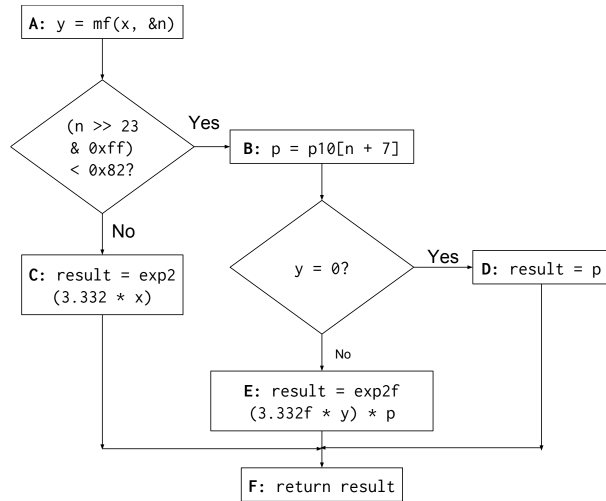


Figure 9: Control flow graph with labeled statements for the code in Figure 8b. **A**, **B**, **D**, **E**, **C**, **F** is one possible sequence of basic blocks when linearized by the Escort compiler.

Line #	Predicate
2, 3, 11	TRUE
4, 5	`(n >> 23 & 0xff) < 0x82`
6	`(n >> 23 & 0xff) < 0x82 ∧ y = 0`
8	`(n >> 23 & 0xff) < 0x82 ∧ y ≠ 0`
10	`¬((n >> 23 & 0xff) < 0x82)`

Table 2: Predicates per line for function in Figure 8b.

each basic block, which we list in Table 2. Third, the Escort compiler linearizes basic blocks by applying a topological sort on the control flow graph (see Figure 9) and fuses the basic blocks together. Finally, the Escort compiler replaces the array access statement in line 4 with a function that sweeps over the entire array. The resulting code, shown in Figure 8c, eliminates control flows and data flows that depend on secret values. In addition to closing digital side channels, the code also uses secure floating-point operations.

5 Security Evaluation

This section demonstrates that Escort's floating-point operations run in fixed time and do not leak information through digital side channels. Since precise timing measurement on x86 processors is tricky due to complex processor and OS design, we take special measures to ensure that our measurements are accurate. In addition to Escort's timing and digital side channel defense, we also demonstrate Escort's defense against a floating-point timing channel attack on the Firefox web browser.

5.1 Experimental Setup

We run all experiments on a 4-core Intel Core i7-2600 (Sandy Bridge) processor. The processor is clocked at 3.4 GHz. Each core on this processor has a 32 KB private L1 instruction cache, a 32 KB private L1 data cache, and a 256 KB private L2 cache. A single 8 MB L3 cache is shared among all four cores. The host operating system is Ubuntu 14.04 running kernel version 3.13. We implement compiler transformations using the LLVM compiler framework [17] version 3.8.

We measure instruction latencies using the `RDTSC` instruction that returns the number of elapsed cycles since resetting the processor. Since the latency of executing the `RDTSC` instruction is usually higher than the latency of executing operations, our setup measures the latency of executing 1024 consecutive operations and divides the measured latency by 1024. Our setup uses the `CPUID` instruction and `volatile` variables for preventing the processor and the compiler from reordering critical instructions. Finally, our setup measures overhead by executing an empty loop body—a loop body that contains no instructions other than those in the test harness. By placing an empty `volatile __asm__` block in the empty loop body, our setup prevents the compiler from deleting the empty loop body.

5.1.1 Outlier Elimination

Many factors outside of the experiment's control, like interrupts, scheduling policies, etc., may result in outliers in performance measurements. We now explain our procedure for eliminating outliers, before demonstrating that the elimination of these outliers does not bias the conclusions.

We use Tukey's method [34] for identifying outliers, but we adapt it to conservatively classify fewer values as outliers (thus including more values as valid data points). The original Tukey's method first finds the minimum (M_n), median (M_d), and maximum (M_x) of a set of values. The first quartile, Q_1, is the median of values between M_n and M_d. The third quartile, Q_3, is the median of values between M_x and M_d. The difference between the first and the third quartiles ($Q_3 - Q_1$) is called the Inter-Quartile Range, R_{IQ}. Tukey's method states that any value v, such that $v > Q_3 + 3 \times R_{IQ}$ or $v < Q_1 - 3 \times R_{IQ}$ is a probable outlier. In our evaluation, we weaken our outlier elimination process (*i.e.* we count fewer values as outliers), by (1) setting the R_{IQ} to be at least equal to 1.0, and (2) classifying v as an outlier when $v > Q_3 + 20 \times R_{IQ}$ or $v < Q_1 - 20 \times R_{IQ}$. Results presented in the following sections use the relaxed Tukey method described above.

	Mean	**Median**	**Std. Dev.**
Different Operands	847,323 (0.81%)	1,066,270 (1.02%)	381,467
Same Operands	929,703 (0.89%)	1,139,961 (1.09%)	364,192

Table 3: Number of discarded outliers from 100 million double-precision square-root operations. The results indicate that our outlier elimination process is statistically independent of the input operand values.

To demonstrate that our outlier elimination process does not bias conclusions, we compare the distribution of outliers between (a) 100 million operations using randomly-generated operands, and (b) 100 million operations using one fixed operand. The two experiments do not differ in any way other than the difference in their input operands. Table 3 shows the mean, median, and standard deviation of outliers for the double-precision square-root operation. Results for other floating-point operations are similar and are elided for space reasons. Since the difference in mean values as well as the difference in median values is within a quarter of the standard deviation from the mean, we conclude that the discarded outlier count is statistically independent of the input operand values.

5.2 Timing Assurance of Elementary Operations

Since exhaustively testing all possible inputs for each operation is infeasible, we instead take the following three-step approach for demonstrating the timing channel defense for Escort's elementary operations: (1) We characterize the performance of Escort's elementary operations using a specific, fixed floating-point value (*e.g.* 1.0), (2) using one value from each of the six different types of values (zero, normal, subnormal, +∞, -∞, and not-a-number), we show that our solution exhibits negligible variance in running time, and (3) to demonstrate that each of the six values in the previous experiment is representative of the class to which it belongs, we generate 10 million normal, subnormal, and not-a-number (NaN)

values, and show that the variance in running time among each set of 10 million values is negligible. Our key findings are that Escort's operations run in fixed time, are fast, and that their performance is closely tied to the performance of the hardware's subnormal operations.

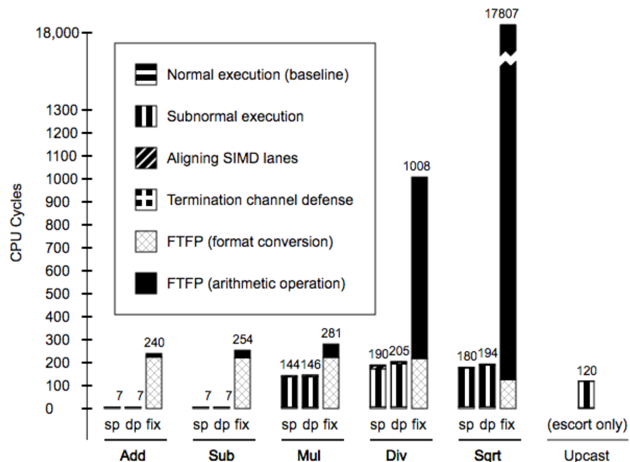


Figure 10: Comparison of running times of elementary operations. **sp** identifies Escort's single-precision operations, **dp** identifies Escort's double-precision operations, and **fix** identifies FTFP's fixed-point operations. Numbers at the top of the bars show the total cycle count. We see that Escort's execution times are dominated by the cost of subnormal operations, and we see that FTFP's overheads are significantly greater than Escort's.

Figure 10 compares the running times of elementary operations of Escort and of previous solutions (FTFP). First, we observe that the running times of Escort's single- and double-precision operations are an order-of-magnitude lower than those of FTFP's fixed-precision operations. Second, Escort's running time is almost entirely dominated by the processor's operation on subnormal numbers. Third, conversion between fixed-point and floating-point takes a non-trivial amount of time, further increasing the overhead of FTFP's operations. Overall, Escort elementary operations are about 16× faster than FTFP's.

Table 4 shows the variation in running time of elementary operations across six different types of inputs (zero, normal value, subnormal value, +∞, −∞, and not-a-number value) and compares it with the variation of SSE (native) operations. While SSE operations exhibit high variation (the maximum observed standard deviation is 176% of the mean), Escort's operations show negligible variation across different input types.

Finally, we measure Escort's running time for 10 million random normal, subnormal, and not-a-number values. We observe that the standard deviation of these measurements, shown in Table 5, is extremely low (at most

Function	Escort	Native (SSE)
add-sp	0	0
add-dp	0	0
sub-sp	0	0
sub-dp	0	0
mul-sp	0	49.2 (175%)
mul-dp	0	49.2 (175%)
div-sp	0.66 (0.4%)	65.67 (163%)
div-dp	1.66 (0.8%)	69.08 (164%)
sqrt-sp	1.49 (0.8%)	62.7 (170%)
sqrt-dp	2.98 (1.5%)	66.87 (169%)
upcast	0	40.99 (178%)

Table 4: Comparison of standard deviation of running times of elementary operations across six types of values (zero, normal, subnormal, +∞, −∞, and not-a-number). Numbers in parenthesis show the standard deviation as a percentage of the mean. The **-sp** suffix identifies single-precision operations while the **-dp** suffix identifies double-precision operations. Compared to SSE operations, Escort exhibits negligible variation in running times.

3.1% of the mean). We thus conclude that our chosen values for each of the six classes faithfully represent their class.

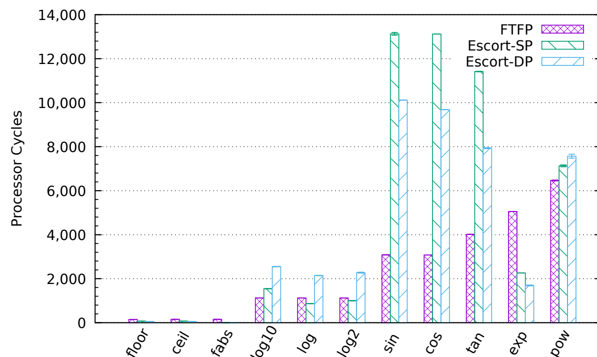


Figure 11: Comparison of running times of commonly used higher-level functions. Error bars (visible for only a few functions) show the maximum variation in running time for different kinds of input values.

5.3 Timing Assurance of Higher-Level Operations

Using different types of floating-point values (zero, normal, subnormal, +∞, −∞, and not-a-number), Figure 11 compares the performance of most of the commonly used

Fn.	NaN	Normal	Subnormal
add-sp	0.21 (3.1%)	0.21 (2.9%)	0.19 (2.7%)
add-dp	0.21 (3.0%)	0.20 (2.9%)	0.21 (3.0%)
sub-sp	0.18 (2.6%)	0.19 (2.7%)	0.20 (2.9%)
sub-dp	0.19 (2.7%)	0.19 (2.7%)	0.19 (2.7%)
mul-sp	0.98 (0.7%)	0.94 (0.7%)	1.05 (0.7%)
mul-dp	0.90 (0.6%)	1.04 (0.7%)	1.02 (0.7%)
div-sp	1.22 (0.6%)	1.27 (0.7%)	1.23 (0.6%)
div-dp	1.39 (0.7%)	1.37 (0.6%)	1.17 (0.6%)
sqrt-sp	1.15 (0.6%)	1.13 (0.6%)	1.14 (0.6%)
sqrt-dp	1.29 (0.7%)	1.41 (0.7%)	1.33 (0.7%)
upcast	1.03 (0.9%)	0.89 (0.8%)	0.95 (0.8%)

Table 5: Standard deviation of 10 million measurements for each type of value (normal, subnormal, and not-a-number). All standard deviation values are within 3.1% of the mean. Furthermore, the mean of these 10,000,000 measurements is always within 2.7% of the representative measurement.

single- and double-precision higher-level operations[6]. Overall Escort's higher-level operations are about 2× slower than their corresponding FTFP operation, which is the price for closing side channels that FTFP does not close.

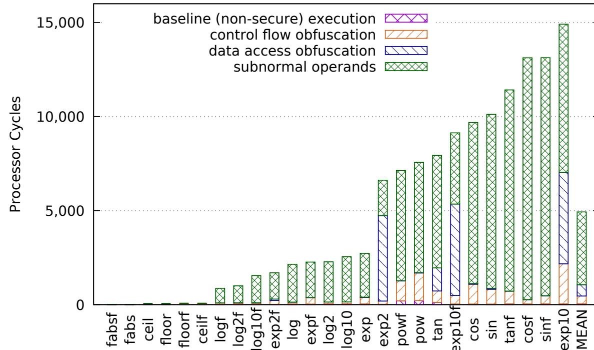


Figure 12: Performance breakdown of Escort's commonly used higher-level functions. The baseline (non-secure) execution and exception handling together cost less than 250 cycles for each function, making them too small to be clearly visible in the above plot.

Figure 12 shows the breakdown of the performance of commonly used higher-level functions. We observe that the performance of most higher-level functions is dominated by the latency of operations on subnormal operands, which is closely tied to the performance of the underlying hardware. A handful of routines (`exp10()`, `exp10f()`, `exp2()`, and `exp2f()`) use lookup tables that are susceptible to address-trace-based side-channel information leaks, so the code transformed by Escort sweeps over these lookup tables for each access to the table. Finally, we see that the cost of control flow obfuscation (*i.e.* the cost of executing all instructions in the program) contributes the least to the total overhead.

[6]We exclude the `exp2()` (6,617 cycles), `exp10()` (14,910 cycles), `exp2f()` (1,693 cycles), and `exp10f()` (9,134 cycles) from Figure 11 because FTFP does not implement these operations.

5.4 Side-Channel Defense in Firefox

We now evaluate Escort's defense against the timing channel attack by Andrysco et al. [3] on the Firefox web browser. The attack reconstructs a two-color image inside a victim web page using only the timing side channel in floating-point operations. The attack convolves the given secret image with a matrix of subnormal values. The convolution step for each pixel is timed using high resolution Javascript timers. By comparing the measured time to a threshold, each pixel is classified as either black or white, effectively reconstructing the secret image.

We integrate Escort into Firefox's convolution code[7] and re-run the timing attack. The results (see Figure 13c) show that Escort successfully disables the timing attack.

5.5 Control- and Data-Flow Assurance

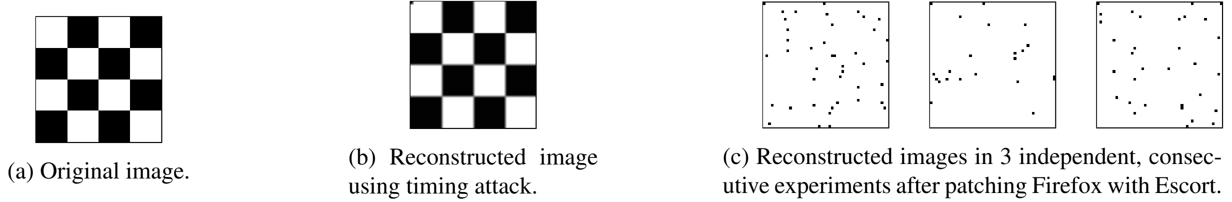
We now show that Escort's operations do not leak information through control flow or data flow. We first use inference rules over the LLVM IR to demonstrate non-interference between secret inputs and digital side channels. We run a machine-learning attack on Escort and demonstrate that Escort successfully disables the attack.

5.5.1 Non-Interference Using Inference Rules

Since Escort's elementary operations are small and simple—they are implemented using fewer than 15 lines of assembly code, they do not access memory, and they do not contain branch instructions—they are easily verified for non-interference between secret inputs and digital side channels. Using an LLVM pass that applies the inference rules from Table 6, tracking labels that can be either **`L`** (for low-context *i.e.* public information) or **`H`** (for high-context *i.e.* private information), we verify that Escort's higher-level operations close digital side channels. This compiler pass initializes all function arguments with the label **`H`**, since arguments represent secret inputs.

Inference rules for various instructions dictate updates to the labels. The environment Γ tracks the label of each pointer and each address. The Escort compiler tags `load`

[7]Specifically, we replace three single-precision multiplication operations with invocations to the equivalent Escort function. All source code changes are limited to the code in the `ConvolvePixel()` function in `SVGFEConvolveMatrixElement.cpp`.



(a) Original image.

(b) Reconstructed image using timing attack.

(c) Reconstructed images in 3 independent, consecutive experiments after patching Firefox with Escort.

Figure 13: Results of attack and defense on a vulnerable Firefox browser using timing-channel information leaks arising from the use of subnormal floating-point numbers.

and `store` instructions as secret if the pointer is tainted, or public otherwise. Unlike a public `load` or `store` instruction, a secret `load` or `store` instruction is allowed to use a tainted pointer since Escort generates corresponding loads and stores to *all* statically-determined candidate values in the points-to set. The sanitization rule resets the value's label to `L` and is required to suppress false alarms from Escort's loop condition transformation. Escort's transformed code includes instructions with special LLVM metadata that trigger the sanitization rule.

During verification, the compiler pass iterates over each instruction and checks whether a rule is applicable using the rule's antecedents (the statement above the horizontal line); if so, it updates its local state as per the rule's consequent (the statement below the horizontal line). If no applicable rule is found, then the compiler pass throws an error. The compiler pass processes the code for Escort's 112 higher-level operations without throwing errors.

5.5.2 Defense Against Machine-Learning Attack

We use the TensorFlow [1] library to design a machine-learning classifier, which we use to launch a side-channel attack on the execution of the `expf()` function, where the input to the `expf()` function is assumed to be secret. Using three distinct inputs, we run this attack on the implementations in the (non-secure) Musl C library and in the (secure) Escort library. We first use the Pin dynamic binary instrumentation tool [19] to gather the full instruction address traces of both `expf()` implementations[8]. We train the TensorFlow machine-learning classifier by feeding the instruction address traces to the classifier, associating each trace with the secret input to `expf()`. We use cross entropy as the cost function for TensorFlow's training phase. In the subsequent testing phase, we randomly select one of the collected address traces and ask the classifier to predict the secret input value.

We find that for the Musl implementation, the classifier is accurately able to predict the correct secret value from the address trace. On the other hand, for the Escort implementation, the classifier's accuracy drops to 33%, which is no better than randomly guessing one of the three secret input values.

[8]Using the `md5sum` program, we observe that Escort's address traces for all three inputs are identical.

6 Precision Evaluation

We examine the precision of Escort and FTFP by comparing Escort's and FTFP's results with those produced by a standard C library.

6.1 Comparison Using Unit of Least Precision

Methodology. We adopt an empirical approach to estimate precision in terms of Unit of Least Precision (ULP), since formal derivation of maximum ULP difference requires an intricate understanding of theorem provers and floating-point algorithms. We run various floating-point operations on 10,000 randomly generated pairs (using `drand48()`) of floating-point numbers between zero and one. For elementary operations, we compare the outputs of Escort and FTFP with the outputs of native x86 instructions. For all other operations, we compare the outputs of Escort and FTFP with the outputs produced by corresponding function from the Musl C library.

Results. We observe that Escort's results are identical to the results produced by the reference implementations, *i.e.* the native (x86) instructions and the Musl C library. More precisely, the ULP difference between Escort's results and reference implementation's results is zero. On the other hand, FTFP, which computes arithmetic in fixed-point precision, produces output that differs substantially from the output of Musl's double-precision functions (see Table 7). The IEEE 754 standard requires that addition, subtraction, multiplication, division, and square root operations are computed with ULP difference of at most 0.5. Well-known libraries compute results for most higher-level operations within 1 ULP.

Rule	Premises	Conclusion
T-PUBLIC-LOAD	$\Gamma(\texttt{ptr}) = \texttt{L}$; $P = ptset(\texttt{ptr})$; $\texttt{m} = \max_{\texttt{addr} \in P} \Gamma(\texttt{addr})$; $\Gamma' = \Gamma[\texttt{val} \mapsto \texttt{m}]$	$\Gamma \vdash \texttt{val := public-load ptr} : \Gamma'$
T-PUBLIC-STORE	$\Gamma(\texttt{ptr}) = \texttt{L}$; $\forall\, \texttt{addr} \in ptset(\texttt{p})$; $\texttt{m} = max(\Gamma(\texttt{val}), \Gamma(\texttt{addr}))$; $\Gamma' = \Gamma[\texttt{addr} \mapsto \texttt{m}]$	$\Gamma \vdash \texttt{public-store ptr, val} : \Gamma'$
T-SECRET-LOAD	$\Gamma' = \Gamma[\texttt{val} \mapsto \texttt{H}]$	$\Gamma \vdash \texttt{val := secret-load ptr} : \Gamma'$
T-SECRET-STORE	$\forall\, \texttt{addr} \in ptset(\texttt{p})$; $\Gamma' = \Gamma[\texttt{addr} \mapsto \texttt{H}]$	$\Gamma \vdash \texttt{secret-store ptr, val} : \Gamma'$
T-BRANCH	$\Gamma(\texttt{cond}) = \texttt{L}$	$\Gamma \vdash \texttt{br cond,block1,block2} : \Gamma$
T-OTHER	$\Gamma' = \Gamma[\texttt{x} \mapsto \Gamma(\texttt{y})]$	$\Gamma \vdash \texttt{x:=y} : \Gamma'$
T-COMPOSITION	$\Gamma \vdash S_1 : \Gamma',\ \ \Gamma' \vdash S_2 : \Gamma''$	$\Gamma \vdash S_1;S_2 : \Gamma''$
T-SANITIZER	$\Gamma' = \Gamma[\texttt{x} \mapsto \texttt{L}]$	$\Gamma \vdash S(\texttt{x}) : \Gamma'$

Table 6: Inference rules for verifying the security of Escort's higher-level operations.

Function	Min.	Median	Max.
add	16	1,743,272	210,125,824
sub	1,312	6,026,976	84,089,503,744
mul	317	8,587,410	112,134,679,849
div	829	5,834,095	30,899,033,427
sqrt	562	2,815,331	21,257,836,468
floor	0	0	0
ceil	0	0	0
log	1,698	5,908,547	2,705,277,8104
log2	262	5,812,840	13,890,632,367
log10	981	10,105,199	40,631,590,323
exp	132	1,409,624	6,066,894
sin	1,316	4,173,786	40,138,955,131
cos	2,166	2,241,360	10,127,702
tan	717	5,576,540	40,126,401,802
pow	522	3,425,870	26,876,068,127
fabs	352	3,129,984	40,134,770,688

Table 7: Floating-point difference for 10,000 operations on random inputs in terms of Unit of Least Precision (ULP) in **FTFP versus Musl C library**. Since we observe zero ULP distance between Escort's results and Musl's results, this table omits Escort's results.

$< 10^{-5}$	10^{-5} to 10^{-3}	10^{-3} to 10^{0}	10^{0} to 10^{3}	$> 10^{3}$
49%	9%	21%	10%	11%

Table 8: Distribution of differences in answers produced by MINPACK-FTFP and MINPACK-C. In all, 321 values differ between the outputs of the two programs.

6.2 Comparison of Program Output

Methodology. Since differences in program outputs provide an intuitive understanding of the error introduced by approximate arithmetic operations, we compare the output of the test suite of Minpack[9], a library for solving non-linear equations and non-linear least squares problems. We generate three variants of Minpack: MINPACK-C uses the standard GNU C library, MINPACK-ESCORT uses the Escort library, and MINPACK-FTFP uses the FTFP library. We run the 29 programs in Minpack's test suite and compare the outputs produced by the three program variants.

Results. We observe that MINPACK-ESCORT produces output that is identical to MINPACK-C's output. We also observe that all outputs of MINPACK-FTFP differ from MINPACK-C. Specifically, 321 values differ between the outputs of MINPACK-FTFP and MINPACK-C. We analyze all 321 differences between MINPACK-FTFP and MINPACK-C by classifying them into the following five categories: (1) smaller than 10^{-5}, (2) between 10^{-5} and 10^{-3}, (3) between 10^{-3} and 10^{0}, (4) between 10^{0} and 10^{3}, and (5) larger than 10^{3}. As seen in Table 8, almost half of the differences (49%) are extremely small (less than 10^{-5}), possibly arising from relatively small differences between fixed-point and floating-point calculations. However, we hypothesize that differences amplify from propagation, since nearly 42% of the differences are larger than 10^{-3}.

[9] `https://github.com/devernay/cminpack`

7 Performance Evaluation

We now evaluate the end-to-end application performance impact of Escort's floating-point library and Escort's control flow obfuscation.

Application	Escort Overhead	Static (LLVM) Floating-Point Instruction Count
433.milc	29.33×	2,791
444.namd	57.32×	9,647
447.dealII	20.31×	21,963
450.soplex	4.74×	4,177
453.povray	82.53×	25,671
470.lbm	56.19×	711
480.sphinx3	52.46×	629
MEAN	**32.63×** **(geo. mean)**	**9,370** **(arith. mean)**

Table 9: Overhead of SPEC-ESCORT (SPECfp2006 using Escort operations) relative to SPEC-LIBC (SPECfp2006 using libc).

7.1 Impact of Floating-Point Library

This section evaluates the performance impact of Escort on the SPEC floating point benchmarks, as well as on a security-sensitive program SVM^{light}, a machine-learning classifier.

Evaluation Using SPEC Benchmarks. We use the C and C++ floating-point applications in the SPEC CPU 2006 benchmark suite with reference inputs. We generate two versions of each program—the first version (SPEC-LIBC) uses the standard C library functions, and the second version (SPEC-ESCORT) uses functions from the Escort library[10]. We compile the SPEC-LIBC program using the Clang/LLVM 3.8 compiler with the `-O3` flag, and we disable auto-vectorization while compiling the SPEC-ESCORT program. The following results demonstrate the *worst* case performance overhead of Escort for these programs, since we transform *all* floating-point operations in SPEC-ESCORT to use the Escort library. More precisely, we do not reduce the number of transformations either using taint tracking or using SMT solvers.

Table 9 shows that Escort's overhead is substantial, with a geometric mean of 32.6×. We expect a lower average overhead for applications that use secret data, since taint tracking would reduce the number of floating-point operations that would need to be transformed.

Evaluation Using SVM^{light}. To evaluate Escort's overhead on a security-sensitive benchmark, we measure Escort's performance on SVM^{light}, an implementation of Support Vector Machines in C, using the four example test cases documented on the SVM^{light} website[11]. We mark the training data and the classification data as secret. Before replacing floating-point computations, Escort's taint analysis discovers all floating-point computations that depend on the secret data, thus reducing the list of replacements. We also instruct Escort to query the Z3 SMT solver to determine whether candidate floating-point computations could use subnormal operands. Escort then replaces these computations with secure operations from its library. We compile the baseline (non-secure) program using the Clang/LLVM 3.8 compiler with the `-O3` flag, and we disable auto-vectorization while compiling SVM^{light} with Escort. We measure the total execution time using the `RDTSC` instruction. Table 10 shows that Escort's overhead on SVM^{light}. We observe that Escort's overhead on SVM^{light} is substantially lower than that on SPEC benchmarks. Using the `md5sum` program, we verify that the output files before and after transformation of SVM^{light} are identical.

Test Case	Overhead for Training	Overhead for Classification
#1	8.66×	1.34×
#2	30.24×	0.96×
#3	1.41×	1.11×
#4	12.75×	0.92×
GEO MEAN	**8.28×**	**1.07×**

Table 10: Overhead of Escort on SVM^{light} program.

7.2 Impact of Control Flow Obfuscation

To compare the performance impact of Escort's control flow obfuscation technique with that of Raccoon, we use the same benchmarks that were used to evaluate Raccoon [28], while compiling the baseline (non-transformed) application with the `-O3` optimization flag. Although both Escort and Raccoon obfuscate control flow *and* data accesses, we compare the cost of control flow obfuscation only, since both Escort and Raccoon obfuscate data accesses using the identical technique. Table 11 shows the results.

We find that programs compiled with Escort have a significantly lower overhead than those compiled with Raccoon. Escort's geometric mean overhead is 32%, while that of Raccoon is 5.32×. The worst-case overhead for Escort is 2.4× (for `ip-tree`).

The main reason for the vast difference in overhead is that Raccoon obfuscates branch instructions at *execution* time, which requires the copying and restoring of

[10] We also ran the same programs using the FTFP library, but the programs either crashed due to errors or ran for longer than two hours, after which they were manually terminated.

[11] `http://svmlight.joachims.org/`

Benchmark	Raccoon Overhead	Escort Overhead
ip-tree	1.01×	2.40×
matrix-mul	1.01×	1.01×
radix-sort	1.01×	1.06×
findmax	1.01×	1.27×
crc32	1.02×	1.00×
genetic-algo	1.03×	1.03×
heap-add	1.03×	1.27×
med-risks	1.76×	1.99×
histogram	1.76×	2.26×
map	2.04×	1.01×
bin-search	11.85×	1.01×
heap-pop	45.40×	1.44×
classifier	53.29×	1.24×
tax	444.36×	1.67×
dijkstra	859.65×	1.10×
GEO MEAN	**5.32×**	**1.32×**

Table 11: Performance comparison of benchmarks compiled using Raccoon and Escort. We only compare the control flow obfuscation overhead, since both Raccoon and Escort use the same technique for data access obfuscation.

the stack for each branch instruction. Since the stack can be arbitrarily large, such copying and restoring adds substantial overhead to the running time of the program. On the other hand, Escort's code rewriting technique obfuscates code at *compile* time using basic block predicates, which enables significant performance boosts on the above benchmarks.

8 Conclusions

In this paper, we have presented Escort, a compiler-based tool that closes side channels that stem from floating-point operations. Escort prevents an attacker from inferring secret floating-point operands through the timing channel, though micro-architectural state, and also through off-chip digital side channels, such as memory address trace.

Escort uses native SSE instructions to provide speed and precision. Escort's compiler-based approach enables it to support a significantly larger number of floating-point operations (112) than FTFP (19).

Escort's design motivates further research into hardware support for side-channel resistant systems. For example, by allowing software to control the timing of integer instruction latencies and their pipelined execution, Escort's guarantees could be extended to instructions beyond floating-point instructions.

Acknowledgments. We thank our shepherd Stephen McCamant and the anonymous reviewers for their helpful feedback. We also thank David Kohlbrenner for giving us the Firefox timing attack code. We are grateful to Jia Chen for providing us the pointer analysis library, and to Joshua Eversmann for help with code and discussions. This research was funded in part by NSF Grants DRL-1441009, CNS-1314709, and CCF-1453806, C-FAR (one of the six SRC STARnet Centers sponsored by MARCO and DARPA), and a gift from Qualcomm.

References

[1] ABADI, M., ET AL. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Computing Research Repository abs/1603.04467* (2016).

[2] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the Power of Simple Branch Prediction Analysis. In *Symposium on Information, Computer and Communications Security* (2007), pp. 312–320.

[3] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On Subnormal Floating Point and Abnormal Timing. In *Symposium on Security and Privacy (S&P)* (2015), pp. 623–639.

[4] BRUMLEY, D., AND BONEH, D. Remote Timing Attacks are Practical. *Computer Networks 48*, 5 (2005), 701–716.

[5] CLEEMPUT, J. V., COPPENS, B., AND DE SUTTER, B. Compiler Mitigations for Time Attacks on Modern x86 Processors. *Transactions on Architecture and Code Optimization 8*, 4 (Jan. 2012), 23:1–23:20.

[6] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium* (2015).

[7] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), pp. 337–340.

[8] DEMMEL, J. W. Effects of Underflow on Solving Linear Systems. Tech. Rep. UCB/CSD-83-128, EECS Department, University of California, Berkeley, Aug 1983.

[9] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic Analysis: Concrete Results. In *Third International Workshop on Cryptographic Hardware and Embedded Systems* (2001), pp. 251–261.

[10] GROSSSCHÄDL, J., OSWALD, E., PAGE, D., AND TUNSTALL, M. Side-Channel Analysis of Cryptographic Software via Early-terminating Multiplications. In *International Conference on Information Security and Cryptology* (2010), pp. 176–192.

[11] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium, NDSS* (2012).

[12] JANA, S., AND SHMATIKOV, V. Memento: Learning Secrets from Process Footprints. In *Symposium on Security and Privacy (S&P)* (2012), pp. 143–157.

[13] KAHAN, W. Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard. *Interval Mathematics* (1980), 99–128.

[14] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology* (1996), pp. 104–113.

[15] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *19th Annual International Cryptology Conference on Advances in Cryptology* (1999), pp. 388–397.

[16] KONG, J., ACIIÇMEZ, O., SEIFERT, J., AND ZHOU, H. Hardware-Software Integrated Approaches to Defend Against Software Cache-Based Side Channel Attacks. In *International Conference on High-Performance Computer Architecture* (2009), pp. 393–404.

[17] LATTNER, C., AND ADVE, V. S. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization* (2004), pp. 75–88.

[18] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), pp. 87–101.

[19] LUK, C., ET AL. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation* (2005), pp. 190–200.

[20] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Conference on Computer and Communications Security* (2013), pp. 311–324.

[21] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks. In *International Symposium on Computer Architecture* (2012), pp. 118–129.

[22] MASTI, R. J., ET AL. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security Symposium* (2015), pp. 865–880.

[23] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *International Conference on Information Security and Cryptology* (2005), pp. 156–168.

[24] MUCHNICK, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[25] MULLER, J.-M. On the definition of ulp(x). Tech. Rep. 2005-009, ENS Lyon, February 2005.

[26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *RSA Conference on Topics in Cryptology* (2006), pp. 1–20.

[27] PERCIVAL, C. Cache Missing for Fun and Profit. In *Proceedings of the Technical BSD Conference* (2005).

[28] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Conference on Security Symposium* (2015), pp. 431–446.

[29] REN, L., YU, X., FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *International Symposium on Computer Architecture* (2013), pp. 571–582.

[30] SAKURAI, K., AND TAKAGI, T. A Reject Timing Attack on an IND-CCA2 Public-key Cryptosystem. In *International Conference on Information Security and Cryptology* (2003), pp. 359–374.

[31] SCHINDLER, W. A Timing Attack Against RSA with the Chinese Remainder Theorem. In *International Workshop on Cryptographic Hardware and Embedded Systems* (2000), pp. 109–124.

[32] SHI, E., CHAN, T. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((logN)^3)$ Worst-Case Cost. In *Advances in Cryptology* (2011), pp. 197–214.

[33] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C. W., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Conference on Computer and Communications Security* (2013), pp. 299–310.

[34] TUKEY, J. *Exploratory Data Analysis*. Pearson, 1977.

[35] WANG, Y., FERRAIUOLO, A., AND SUH, G. E. Timing Channel Protection for a Shared Memory Controller. In *International Symposium on High Performance Computer Architecture* (2014), pp. 225–236.

[36] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture* (2007), pp. 494–505.

[37] WANG, Z., AND LEE, R. B. A Novel Cache Architecture with Enhanced Performance and Security. In *International Symposium on Microarchitecture* (2008), pp. 83–93.

[38] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive Mitigation of Timing Channels in Interactive Systems. In *Conference on Conference on Computer and Communications Security* (2011), pp. 563–574.

[39] ZHANG, Y., AND REITER, M. K. Duppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Conference on Computer and Communications Security* (2013), pp. 827–838.