Compiler-Assisted Transformations for Mitigating Timing Side Channels

Chanhee Cho, Henry Jung

1 Introduction

Side channel attacks are common in various software we use today, ranging from cryptographic libraries, web servers, processors, etc. These side channels can occur due to various sources such as timing, power, memory access patterns, etc. In this project, we explored timing side channels due to control-flow and how we may mitigate this using compiler transformations.

Approach We implemented a control-flow side channel mitigation (if-conversion) using LLVM targeting the x86 architecture, based on the approaches explained in [2] and [3]. We then explored how selectively applying transformations to input-dependent areas affects performance, and measured the performance overhead for our evaluation.

Related Work Molnar et al. presented a side channel mitigation based on bit masking using source-to-source transformation [1]. Coppens et al. used conditional execution to remove control-flow side channels, implementing their approach in a compiler backend [2]. Jordan et al. performed a comparison against IR-level and machine-level if-conversion implementations targeting the TI TMS320C64x+ architecture [3].

Contributions Our main technical contributions of this project is exploring one way of how mitigating timing side channels may be addressed through implementing an IR-level if-conversion using LLVM based on past work and evaluating its performance on benchmarks.

2 Control-Flow Transformation Implementation

2.1 If-conversion

If-conversion is a technique used to convert branches into conditional execution, eliminating control-flow. This can be used to mitigate timing side channels that may occur due to control-flow. Our implementation of if-conversion is performed at the IR-level, compared to past work that has targeted the compiler backend[2]. For predication, we make use of LLVM's select instruction to denote conditional execution (select is lowered into cmov by LLVM's x86 backend code). In our implementation, we identified common control-flow patterns that can be converted using if-conversion (e.g. if-then triangle, if-else diamond), and continuously apply if-conversion until no more patterns are identified in the target code.

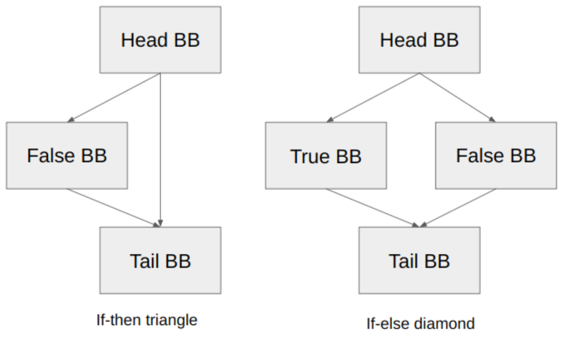


Figure 1: Control-flow patterns

For identifying control-flow patterns, we check if a given basic block is the head block that matches the required basic blocks structure of a control-flow pattern. We make use of our dominators pass to calculate dominators of basic blocks, and then perform relevant checks to see if a given basic block is the head basic block of a pattern. For example, for if-then triangle pattern, we check if the head basic block ends with a conditional branch, the head block has two successor blocks, the false-branch basic block has one successor block, and the head block dominates the false-branch basic block and the tail basic block. For if-else diamond pattern, similarly, we check if the head basic block ends with a conditional branch, the head block has two successor blocks, both the false-branch and true-branch basic blocks have the same successor block (tail block), and the head block dominates the false-branch and true-branch blocks (note the head block does not have to dominate the tail block).

For iterative if-conversion, we adapted the algorithm used in [2] for our implementation.

Algorithm 1 If-conversion

```
while true do
    $patterns \leftarrow$ empty list
    for basic block BB in function do
        if BB matches pattern then
            $patterns \leftarrow patterns \cup \{BB\}$
        end if
    end for
    $converted \leftarrow false$
    for pattern P in patterns do
        $headBB \leftarrow$ head basic block of $P$
        $tailBB \leftarrow$ tail basic block of $P$
        $predsBB \leftarrow$ branching predicate blocks of $headBB$
        Merge $predsBB$ into $headBB$
        Update $\phi$ nodes in $tailBB$ and successor nodes
        Delete dead blocks and instructions
    end for
end while
```

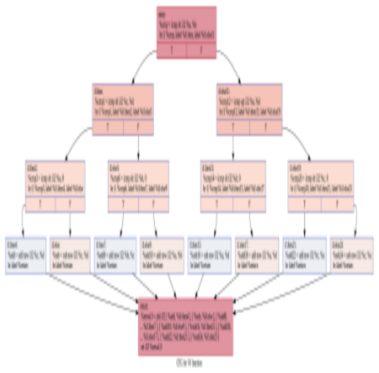
For performing if-conversion on a control-flow pattern, we first move all instructions in the false-branch block (and true-branch block if exists) to the head block, excluding any terminating branch instructions. Then, we move instructions in the tail block to the head block. For phi node instructions in the tail block, we replace the instruction with a SelectInst (conditional execution) based on the branch condition from the head block. In the case of converting an if-else diamond pattern, we want to first check if the head block dominates the tail block. If the head block doesn't dominate the tail block, then we need to preserve the phi node instruction, only removing the relevant incoming values (from false-branch and true-branch blocks) and inserting a new incoming value coming from the head basic block (since false-branch and true-branch blocks have been merged into head block). For our implementation, we also needed to update the phi node entries in successors of tail blocks if the tail block is completely merged into the head block to ensure incoming values in those phi node entries are still valid (updated such that incoming block is now the merged head block). Once all relevant blocks are merged, we delete any dead basic blocks (containing instructions already merged in head block) and obsolete instructions (e.g. old branch instruction in head block; if the tail block is not deleted due to the head block not dominating the tail block, then we create a new edge from the head block to the tail block).

2.2 Safe-guard Instructions

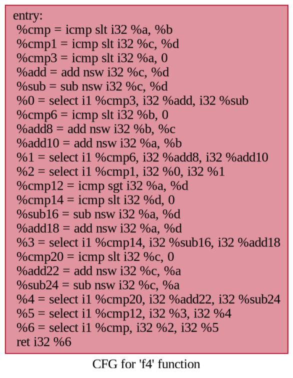
In x86, every instruction cannot be conditionally executed. Only *mov* instructions can be conditionally executed ($CMOV$). When performing if-conversion, some unsafe instructions such as division or loads/stores may cause exceptions or affect correctness. Thus, for these instructions, we insert "safe-guard instructions" to preserve program behavior. For division instructions (isIntDivRem), we insert a conditional execution instruction beforehand that sets the divisor to be 1 if the branch is not taken (if branch is taken, sets to the expected value). Similarly, for loads/stores (LoadInst, StoreInst), we insert a conditional execution instruction beforehand to perform the memory operation from a temporary valid memory location (created using AllocaInst), ensuring safe memory accesses. We also need to consider function calls (CallInst) as some functions may only execute conditionally during execution. To handle this, we duplicate the function that should be conditionally executed, adding an additional parameter that indicates whether instructions in the function should be executed or not (depending on the condition). For any store instructions within the duplicated function, we perform the same conditional store approach from before to ensure program behavior is preserved.

2.3 Microbenchmarks

We performed initial testing on a series of microbenchmarks consisting of different conversion patterns, nested control-flow branches, and handling unsafe instructions. A subset of the microbenchmarks with LLVM IR transformations performed are shown in Figures 2-4.



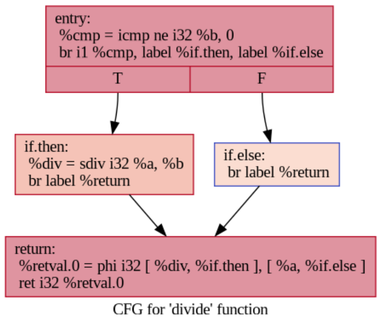
(a) Before Transformation



```
entry:
 %cmp = icmp slt i32 %a, %b
 %cmp1 = icmp slt i32 %c, %d
 %cmp3 = icmp slt i32 %a, 0
 %add = add nsw i32 %c, %d
 %sub = sub nsw i32 %c, %d
 %0 = select i1 %cmp3, i32 %add, i32 %sub
 %cmp6 = icmp slt i32 %b, 0
 %add8 = add nsw i32 %b, %c
 %add10 = add nsw i32 %a, %b
 %1 = select i1 %cmp6, i32 %add8, i32 %add10
 %2 = select i1 %cmp1, i32 %0, i32 %1
 %cmp12 = icmp sgt i32 %a, %d
 %cmp14 = icmp slt i32 %d, 0
 %sub16 = sub nsw i32 %a, %d
 %add18 = add nsw i32 %a, %d
 %3 = select i1 %cmp14, i32 %sub16, i32 %add18
 %cmp20 = icmp slt i32 %c, 0
 %add22 = add nsw i32 %c, %a
 %sub24 = sub nsw i32 %c, %a
 %4 = select i1 %cmp20, i32 %add22, i32 %sub24
 %5 = select i1 %cmp12, i32 %3, i32 %4
 %6 = select i1 %cmp, i32 %2, i32 %5
 ret i32 %6
```

(b) After Transformation

Figure 2: Microbenchmark (nested control-flow): LLVM IR transformation



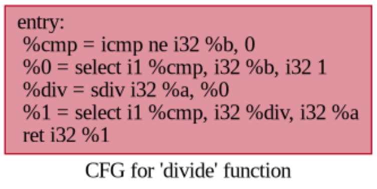
```
entry:
 %cmp = icmp ne i32 %b, 0
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 %div = sdiv i32 %a, %b
 br label %return
```

```
if.else:
 br label %return
```

```
return:
 %retval.0 = phi i32 [ %div, %if.then ], [ %a, %if.else ]
 ret i32 %retval.0
```

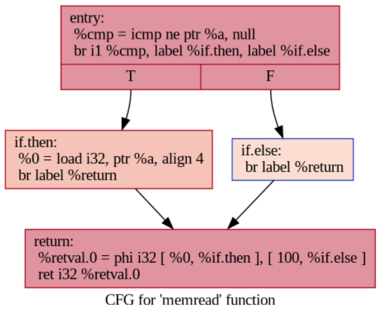
(a) Before Transformation



```
entry:
 %cmp = icmp ne i32 %b, 0
 %0 = select i1 %cmp, i32 %b, i32 1
 %div = sdiv i32 %a, %0
 %1 = select i1 %cmp, i32 %div, i32 %a
 ret i32 %1
```

(b) After Transformation

Figure 3: Microbenchmark (division): LLVM IR transformation



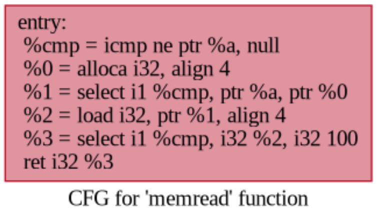
```
entry:
 %cmp = icmp ne ptr %a, null
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 %0 = load i32, ptr %a, align 4
 br label %return
```

```
if.else:
 br label %return
```

```
return:
 %retval.0 = phi i32 [ %0, %if.then ], [ 100, %if.else ]
 ret i32 %retval.0
```

(a) Before Transformation



```
entry:
 %cmp = icmp ne ptr %a, null
 %0 = alloca i32, align 4
 %1 = select i1 %cmp, ptr %a, ptr %0
 %2 = load i32, ptr %1, align 4
 %3 = select i1 %cmp, i32 %2, i32 100
 ret i32 %3
```

(b) After Transformation

Figure 4: Microbenchmark (memread): LLVM IR transformation

3 Experimental Setup

3.1 Performance

We evaluated our implementation by measuring the performance overhead of common functions that may be used in cryptographic libraries, such as modular exponentiation which may be susceptible to timing side channels. We chose three target programs implementing these functions for our benchmarks shown in Table 1.

For performance metrics, we measured the runtime and number of LLVM instructions dynamically executed, comparing the original program (baseline)

Benchmark	Description
modexp	32-bit modular exponentiation
montmul	32-bit montgomery reduction
pointmul	elliptic curve point multiplication

Table 1: Benchmarks

and if-converted program. Our benchmarks were run on a machine with Intel i7-10510U processor. For our experiments, we compiled using -O0 optimization level to avoid any if-conversion that LLVM may already perform (e.g. through its SimplifyCFG pass). For each benchmark, we randomly generated input parameters using a LCG with same seed and performed the computation across 10^5 iterations in a single run. We also verified the outputs matched for both original and if-converted programs to ensure correctness of the transformations.

3.2 Selective Conversion

Another area we explored for our evaluation is how our compiler transformation pass can be optimized using static taint analysis by selectively applying if-conversion to conditional branches that are input dependent. Our approach requires specifying source annotations in the source code to identify the variables that should be marked as secret. We implemented the taint analysis component using our dataflow analysis framework, implemented as an intraprocedural analysis pass.

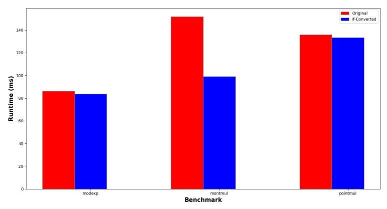
For benchmarks, we chose two target programs relying on secret input shown in Table 2. Given that if-conversion generally leads to better performance, we designed the benchmarks to represent a hypothetical scenario where selectively performing if-conversion may lead to better optimizations; specifically, we have a non-secret dependent branch containing code that may not typically be taken in a normal execution but performs a lot of unnecessary computation (and is not optimized out). In our benchmarks, we annotated any input-dependent parameters in the source code. Our evaluation compares the performance of three different versions of the benchmarks: original program (baseline), if-converted program, and selectively if-converted program (using taint analysis).

Benchmark	Description
passwd_check	password checker program
ecdsa	elliptic curve DSA

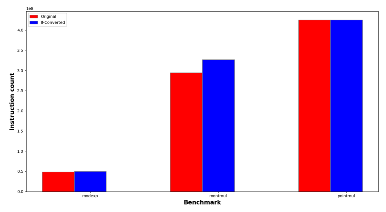
Table 2: Benchmarks (Using Selective Conversion)

4 Experimental Evaluation

We found that the if-converted program led to better performance with faster runtime despite having an increase in instruction count. The benchmark results are shown in Figure 5 below. These results are contrary to results from [2] (from 2009) which reported if-converted benchmarks to have significantly longer execution times. However, our results align with what we would expect on modern processors as due to if-conversion removing control flows, it leads to eliminating need for branch prediction and allows for better use of instruction-level parallelism.



(a) Runtime



(b) Instruction Count

Figure 5: Benchmark Measurements

We also found that selectively performing if-conversion can benefit performance for some applications, as shown in Figure 6. This aligns with what we would expect as converting all branches (versus only converting certain required branches) may not always be optimal as both parts of the branches are conditionally executed by the program after conversion. Note the benchmarks were designed to represent a hypothetical scenario as mentioned in the experimental setup; however this experiment demonstrates that selective if-conversion can be beneficial for some programs.

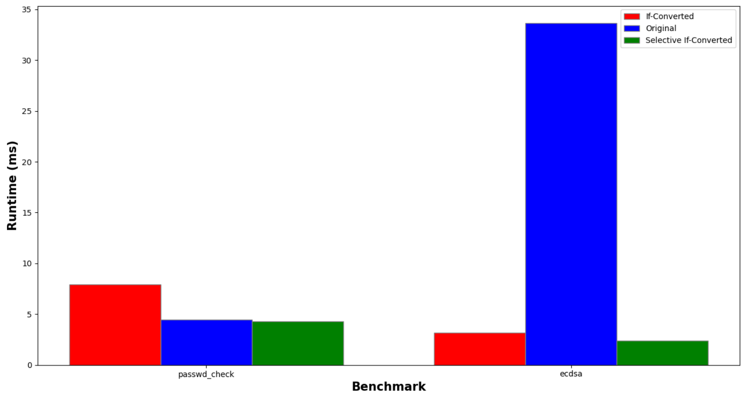


Figure 6: Benchmark Measurements (Using Selective Conversion)

5 Conclusion

5.1 Surprises and Lessons Learned

Through this project, we applied our learnings from this course to implement compiler transformations using LLVM to address security issues (side channel attacks), and also explored how selectively perfoming conversion may benefit in some cases. We were surprised to find that if-conversion can actually lead to better performance in modern processors, and not just serve as a means for side channel mitigation. Overall, we found this project an interesting area to explore.

5.2 Future Work

Future areas to explore outside this project may be determining optimal scenarios for performing if-conversion, while still mitigating against side channels. Also, learning more about and implementing mitigations for modern side channels (not limited to timing) using compiler passes would also be an interesting area to explore.

5.3 Distribution of Total Credit

50-50

References

[1] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The program counter security model: automatic detection and removal of control-flow side channel attacks. In Proceedings of the 8th international conference on Information Security and Cryptology (ICISC'05). Springer-Verlag, Berlin, Heidelberg, 156–168. https://doi.org/10.1007/11734727_14

[2] B. Coppens, I. Verbauwhede, K. De Bosschere and B. De Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," 2009 30th IEEE Symposium on Security and Privacy, Oakland, CA, USA, 2009, pp. 45-60, doi: 10.1109/SP.2009.19.

[3] Alexander Jordan, Nikolai Kim, and Andreas Krall. 2013. IR-level versus machine-level if-conversion for predicated architectures. In Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems (ODES '13). Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/2443608.2443611