LLVM-Blade: an LLVM Pass that Removes Speculative Leaks

by

Maximilian Gallup 2692698

primary supervisor: Klaus von Gleissenthall
daily supervisor: Robin Webbers
second reader: ...

1 Terminology

- *protections* or *guards*: an instruction or procedure inserted into the code to stop speculative execution in a processor.
- *cut-set*: The minimal set of edges in a flow network that upon removal disconnect the source from the sink.

2 Abstract

Blade is a compiler hardening pass developed by Vassena et al. [1] that automatically eliminates speculative leaks by minimally augmenting the code. Secrets leaked under speculation originate from a source expression such as an out-of-bounds memory read. Then the secret flows along a specific data path to a sink expression (any expression used as an array index), which is what ultimately leaks the secret to the attacker. Blade utilizes the fact that it suffices to disconnect the source from the sink expression anywhere along the data path to prevent secret leaks. Thus, instead of preventing speculative execution at all source expressions, Blade minimally inserts protections that can greatly improve performance while still preventing speculative leaks. Blade was implemented for a Webassembly compiler framework which lacks industry-wide adoption and language compatibility when compared to the LLVM Compiler framework. It also lacks aggressive compiler optimization passes which LLVM has been progressing for the last decades. This paper presents LLVM-Blade, an implementation of Blade for the LLVM framework and builds directly on top of the findings made in the original paper [1]. LLVM-Blade places a minimal amount of fences compared to the naive solution, making it a possible candidate for securing cryptographic libraries with minimal to no performance overhead. While LLVM-Blade performs better than the naive solution in some cases, it highly depends on the source code itself, since most findings show that both LLVM-Blade and the naive solution incur no significant overhead compared to the baseline. Additionally, due to register spills and inherent differences between native executables and Webassembly runtimes, it is inconclusive to compare the performance of LLVM-Blade and the original implementation. Implementing Blade in LLVM also poses specific intricacies and challenges, which will be discussed in this paper.

3 Introduction

Speculative execution improves performance by executing instructions before knowing if they need to be executed. For example, in the case of conditional branches, while the condition is being evaluated the processor will speculatively start executing one of the branches before it is known whether that was the right branch. If it was, a performance gain is achieved. However, if it speculated incorrectly, instructions are reverted and execution is rolled back, which can lead to page faults if the incorrect branch made illegal memory accesses. However, all memory accesses during speculation can introduce data into the cache, where it often continues to persist. Because of this, one can reveal what data was copied into the cache through timing measurements and potentially reveal secret information. For example, during speculative execution, the bounds check in Listing 1 (line 1) can be bypassed if the branch predictor has been *mistrained* before the attack. The attacker can thus read memory that contains a secret (line 2 of Listing 1). However, to leak this value to the attacker, the secret is used as an index value and thus loads the value of `reload_buf[secret]` into the cache. The timing measurements across the entire array can reveal the secret value to the attacker.

```
1.  if (attacker_controlled_length < size(vulnerable)) {
2.    int secret = vulnerable[attacker_controlled_length];
3.    int x = reload_buf[secret]
4.  }
```

Listing 1: Speculative execution vulnerability example.

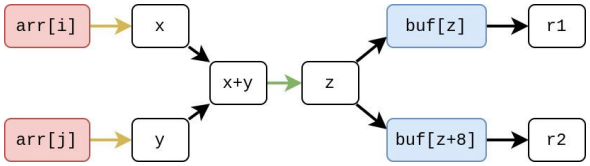
This became known as the infamous Spectre attack [2], which upended the cyber security community because of its huge potential impact. Researchers have been investigating software preventions to be able to protect the many existing processors that are vulnerable to such an attack, since speculative execution as a feature had at that point become entirely omnipresent in processors. Most compiler-based prevention efforts involve placing protections after instructions that might introduce secrets to prevent any speculative execution from happening. One form of protection is a `fence` instruction that prevents the processor from executing speculatively which could mitigate the vulnerability if it was inserted after line 2 of Listing 1. More sophisticated approaches such as Speculative Load Hardening [3] will augment load instructions to mask potential secrets, but both approaches place protections in all places where secrets are introduced resulting in a considerable performance penalty.

Enter Blade [1], a novel approach to eliminate Spectre V1 while introducing minimal overhead. It does so by harnessing the key insight that the point at which secrets are introduced until they are leaked forms a data flow and cutting this data flow is sufficient to prevent speculative leaks. Blade places a minimal number of guards and results in a runtime performance improvement when compared to existing approaches that place protections naively. However, the authors of Blade implemented it for the WebAssembly Lucet [4] compiler which lacks industry-wide adoption and performance benefits from aggressive optimizations compared to the LLVM Compiler Infrastructure Project [5]. Implementing Blade as an LLVM pass is crucial to test whether widespread adoption is feasible and enables the full capabilities of LLVM's multi-decade-long efforts in compiler optimizations. Furthermore, many libraries of cryptographic algorithms supported by the LLVM framework would efficiently be protected against speculative execution with LLVM-Blade. This paper discusses the implementation details of Blade in the LLVM framework and will refer to this implementation [6] as "LLVM-Blade". Furthermore, it presents how different Clang optimization levels vary the resulting placements of protections, discusses compile-time performance considerations for the Blade algorithm and compares performance improvements to the existing Webassembly implementation of Blade [4].

4 Overview

Speculative leaks originate at source expressions, which in terms of high-level source code, are explicit memory reads. For example, the expression `arr[i]` can introduce a secret value under speculation. Then, the secret can flow through the program by being used by other expressions that define new values that are now tainted by the secret, which is known as a *Def-Use* chain. The secret can then flow to a sink expression, for example, `reload_buf[secret]` from Listing 1 which ends up leaking it to the attacker. Blade introduces a type system, that annotates expressions as either ***transient*** if they can speculatively introduce secrets and ***stable*** if they can leak secrets through memory accesses. The problem becomes stopping speculative execution at a minimal number of locations instead of at all source expressions to have the smallest possible performance overhead. To stop speculative execution, Blade introduces the concept of a `protect()` statement, which given an expression, will make sure that no secret can leak because of it. This abstraction allows for different underlying implementations to stop speculative execution. While Blade implements two versions of protect, one with a `fence` instruction and one with SLH [3], LLVM-Blade only implements `protect` with the `fence` instruction. The question then becomes where to place such protections while making sure that the Def-Use chain from source to sink expressions is broken. For example, Listing 2 depicts the Def-Use chain of a simple program involving two ***transient*** expressions (red) and two ***stable*** expressions (blue) connected by a narrow waist. Naively protecting both ***transient*** expressions as seen by the two yellow arrows in Listing 2 involves more protect statements than finding the minimal cut denoted by the single green arrow in Listing 2. Thus, an optimal solution would involve finding the minimal cut over the Def-Use graph of all expressions.

```
int x = arr[i];
int y = arr[j];
int z = x + y;
int r1 = buf[z];
int r2 = buf[z+8];
```



Listing 2: C source code and resulting Def-Use graph with naive (yellow arrow) and optimal (green arrow) cuts.

To efficiently eliminate speculative leaks, LLVM-Blade runs the following steps over each function in the module. First, it marks expressions as either ***transient*** or ***stable***. Then, it builds a Def-Use graph of all instructions, connects a source node ***T*** to all ***transient*** instructions and connects all ***stable*** instructions to a sink node ***S*** (source and sink nodes are not shown in Listing 2). Within this graph, all edges that form a path from source ***T*** to sink ***S*** are part of a leaky path and cutting this path would result in the removal of a speculative secret leak. Next, a Min-Cut Algorithm (the heart of Blade) is performed over this graph to find the minimal number of cuts to separate the source from the sink, known as the *cut-set*. Finally, LLVM-Blade inserts protections over only those instructions in the cut-set resulting in a program that is protected against Spectre V1 attacks.

5 Background

5.1 Min-Cut Algorithm

At the core of finding the cut-set, Blade runs a version of Ford-Fulkerson's Max-Flow Min-Cut algorithm [7]. The algorithm is commonly used to maximize the throughput of flow networks containing a source and a sink node. In such a network, each directed edge can push flow up to a maximum capacity, which is simply the maximum integer value of flow that a given edge can take. Initially, all capacities are 0 and no edge can take more flow than its maximum capacity. The maximum flow capacity from source to sink is found by making arbitrary traversals through the network (from source to sink) and augmenting the capacities with the bottleneck value of the given path. Figure 1 is an example of one such traversal, notice the bottleneck capacity of the green path is 1 because of the edge from `Src -> B`. Furthermore, each augmented path adds a residual path in the opposite direction with the capacity that the augmented path took. Residual paths represent how much flow can be reversed, which means they can also be traversed in the next iteration of the algorithm. The *residual graph* refers to the graph made up of only residual paths, which is used to find the Min-Cut. After no more paths can be taken through the network due to full capacities the algorithm completes and the total outgoing capacity of the source node is the Maximum Flow.

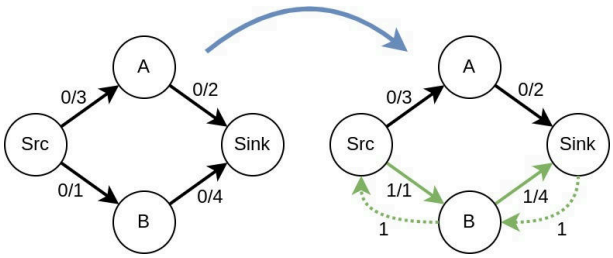


Figure 1: Example of single capacity augmentation after random source to sink traversal

By the Max-Flow Min-Cut Theorem [8] the total weight of the Min-Cut becomes known since it is equal to the Max Flow. However, Blade is interested in the *cut-set* which are those edges that upon removal disconnect the source from the sink. The cut-set can be found by performing a depth-first search on the residual graph and marking all visited nodes. Finally, all edges from a visited node to a

non-visited node of the original graph are part of the cut-set. The nuances of the implementation of this algorithm will be explained in Section 7.3.

6 Threat Model

The scope of this project focuses on preventing the Bounds Check Bypass vulnerabilities, also known as Spectre V1. Thus, an attacker in such a scenario would be able to manipulate the index value that breaks past the bounds of the array during speculative execution, namely `attacker_controlled_variable` in Listing 1. To then retrieve potential secrets, the attacker would have to perform cache timing measurements on the cache of the victim's machine to reveal the value of the secret. Such an attack doesn't require the attacker to have physical access to the victim's machine, for example in the context of a cloud service. While a successful attack allows the attacker to access arbitrary unauthorized information resulting in a huge impact, no such attack has yet been observed in real life [9]. Nonetheless, eliminating the threat at compile time fully prevents the potential for exploitation and is a crucial step in global cyber security.

7 Design

7.1 Assumptions

Explicit memory accesses leak secrets under speculative execution which translates annotating all LLVM `load` instructions as ***transient***. Furthermore, LLVM-Blade conservatively marks all `call` instructions as ***transient***, since calling into shared libraries that might not have been compiled with LLVM-Blade, could be a source of secrets under speculative execution. Additionally, no guarantees can be made about function arguments, so they are also annotated as ***transient***. Furthermore, all expressions that end up being used as array indices for memory accesses are annotated as ***stable***, since those are the sinks in which secrets can leak to the cache. It finds the instructions that need to be stable by checking if any users of a given instruction are part of a `load` instruction. If they are, they will be marked ***stable***. For a concise description of which how instructions are marked see Appendix 1-2 or inspect the source code ***(link)***.

Additionally, register spills happen when data is moved out of a register and into main memory to make space for other data. This potentially introduces more `load` instructions which should be part of LLVM-Blade's analysis so that they don't lead to secret leaks. However, register spills are target dependent and any information about register spills is not available during the LLVM's optimization phase using LLVM IR. Thus, this project entirely forgoes protecting against register spills, because it is out of scope.

7.2 Link Time Optimizations

Compilers for languages that use LLVM (like Clang, Swift and Rust) compile down to LLVM's Intermediate Representation(IR), which is an extensible assembly-like syntax that can be then used to generate machine code instructions. LLVM then performs optimizations over this in-memorry IR. Since LLVM compiler optimization passes can radically change the in-memory LLVM IR, it is crucial that LLVM-Blade runs after all other optimizations to preserve the Def-Use chains of potential speculative leaks. Link Time Optimizations (LTO) are applied during the final (linking) stage of the compilation including all translation units [10]. This allows the compiler to have a full view of the program and run any final optimizations over it [11]. Blade should thus be implemented as the final Link Time Optimization to ensure no alterations to instructions can break the dependency chain and accidentally leak transient secrets.

7.3 Ford-Fulkerson's Algorithm

Ford-Fulkerson's Max-Flow Min-Cut algorithm has been designed for flow networks whose edges have varying capacities. However, in the case of Blade, the flow network represents the Def-Use chain of all expressions in the function, which means that all edges have uniform capacity. While simplifying the construction of the residual graph, the uniform capacities do not change the fact that Ford-Fulkerson's algorithm is necessary to correctly find the cut-set.

The Listing 2 example shows that the maximum flow of the network is 1 because of the "narrow-waist" (green arrow in Listing 2), which by the Max-Flow Min-Cut Theorem, makes it the minimal cut. It is thus not sufficient to only take the sum of edges connected to the source or sink node into account when finding the Max Flow. Listing 2 is an example of why a correct algorithm such as Ford Fulkerson's is needed to find correctly the cut-set, because it does not suffice to only look at the number of ***transient*** or ***stable*** nodes.

8 Implementation Details

8.1 Instruction Marking

LLVM offers the `MDNode` Class as a lightweight mechanism for adding textual meta-data to instructions. Each instruction within a function is checked and annotated with metadata that can then easily be checked with idempotent helper functions such as `markInstructionTransient(&I)` and `isTransientInstruction(&I)`. Additionally, metadata information is displayed in a human-readable format next to instructions within the LLVM IR which makes debugging and manual analysis possible.

8.2 Leaky Path Isolation Inefficiency

Initially, it seemed worthwhile to only run the Ford-Fulkerson Algorithm over the dependency graphs of those instructions involved in a secret leak and not over the entire program. However, this required traversing the entire dependency chain once and isolating only those instructions that are on a dependency path that involves a leak (where there is a path from ***T*** to ***S***). However, such a function resulted in stack overflows during compilation due to the recursive implementation at the time. After refactoring to an iterative approach, compilation still incurred a vast memory overhead and took too much time to complete. Thus, the final implementation did not perform this isolation step and instead converted the entire program (including non-leaky paths) into a matrix representation that was then passed to the Max-Flow Min-Cut algorithm. Despite there being superfluous paths in the graph that would never even be considered, Ford-Fulkerson's algorithm has a time complexity of $O(Ef)$ where E represents the number of edges and f the Max-Flow. Ford-Fulkerson's polynomial time algorithm performs better than the algorithm used to first isolate leaky paths, making it completely redundant.

8.3 Dependency Graph Representation

A matrix representation of the dependency graph allows for a straightforward implementation of the residual graph and simplifies implementing graph traversal. Such a data structure takes the shape of $N \times N$ where $N = (\text{total number of instructions} + 2)$ since the source and sink node need to be inserted at the top and bottom respectively. For each instruction i and j where $i \neq j$, there is a 1 in the j-th column if there is an edge between instruction (i, j). For example, the first row in Figure 2 represents the ***T*** node which is only connected to all ***transient*** instructions and all ***stable*** instructions connect to the last row representing the sink. To construct such a matrix, first, all instructions of the function get numbered in chronological order, by simply adding each instruction pointer to a set named `inst_set` for example. This allows us to identify each instruction by its index which will become the row number of the matrix. Then, we get the indices of all users of a given instruction and mark the respective columns of a given row with a 1. Listing 3 shows how simple this construction is. The final step is to add a 1 to each column representing a ***transient*** instruction of the first row,

effectively connecting the source node. Finally, all rows representing ***stable*** instructions must add a 1 to their last column, connecting them to the sink node.

```
for current_inst in instructions:
    for user in current_inst:
        row = getInstructionIndex(inst_set, current_inst);
        col = getInstructionIndex(inst_set, user);
        graph[row][col] = 1;
```

Listing 3: Pseudo-code of graph construction.

Once the graph in Figure 2 has been constructed, each vertex of the graph needs to be extended by a *dummy-vertex* in order to circumvent the following problem: The Ford-Fulkerson's algorithm will find the minimal set of **edges** to disconnect the source from the sink, but the insertion of protections translates to the removal of vertices instead of edges of the graph. This discrepancy leads to the fact that certain programs will not have optimal cuts. For example, Graph A of Figure 3 places two cuts although there is a single instruction that can be protected which would be a more optimal cut. To still place minimal cuts each vertex must be extended with a dummy vertex so that the single edge can be found by the Min Cut algorithm as seen by the yellow nodes in Graph B of Figure 3. The main reason behind this is that Blade centers around conceptualizing a Def-Use graph which has expressions as nodes. However, LLVM uses **instructions** as nodes in the Def-Use chain, which changes the semantics of a node accordingly. Furthermore, since the protection used in this project is a `fence` instruction, it simply needs to be added after or before the right target instruction, making it necessary to operate at the instruction level, as opposed to expressions.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2: Matrix representation of Graph A in Figure 3

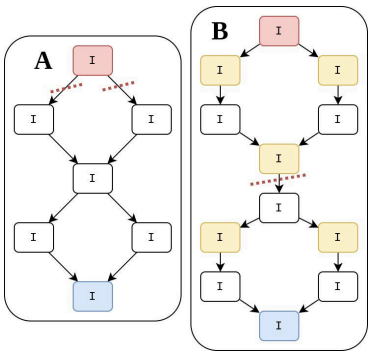


Figure 3: Edge cut problem results in a less efficient data structure.

While the addition of dummy-instructions effectively doubles the size of the data structure in Figure 2, it does not affect the overall performance to such an extent that compile times become unreasonable. However, the space usage for this internal representation is now in the order of $(2N)^2 + 2$ for N total instructions.

8.4 Store Instruction Disruption

LLVM compilers like Clang offer various levels of how aggressively the code will be optimized from a developer's perspective. The correctness of LLVM-Blade must not depend on this and allow for all optimization levels, but in some cases compiling source code with flags `-O0` versus `-O3` may lead to different cut-sets with a non-minimal amount of protections inserted. The root cause of this when using Clang has to do with the fact that at the compiler optimization level `-O0` variables are allocated on the stack with the `alloca` instruction and due to *Single Static Assignment* [12], are only written to via the `store` instruction.

A:

```
// Clang Optimization Level 0
%call1 = call i32 @rand() #3, !BLADE-T !9
%rem = srem i32 %call1, 5
store i32 %rem, ptr %i, align 4
%0 = load i32, ptr %i, align 4, !BLADE-T !9
%idxprom = sext i32 %0 to i64
%arrayidx = gep (...) i64 %idxprom, !BLADE-S !8
```

B:

```
// Clang Optimization Level >= 1
%call1 = tail call i32 @rand() #2, !BLADE-T !7
%rem = srem i32 %call1, 5
%idxprom = sext i32 %rem to i64
%arrayidx = gep (...) i64 %idxprom, !BLADE-S !8
```

Listing 4: Clang optimization level differences due to store instruction.

For example, LLVM Intermediate Representation (IR) snippet A of Listing 4 the `%call1` instruction introduces the speculative leak and is marked as ***transient***. Then, its Def-Use Chain continues to the next instruction `%rem` but ends there due to the following `store` instruction breaking the Def-Use chain. Although this seems like it would affect correctness, on the following line the value stored in `%i` is loaded into `%0` marking it ***transient*** and allowing for a Def-Use Chain to follow until the ***stable*** instruction, correctly indicating a leaky path. In comparison to snippet B of Listing 4, the resulting Def-Use chains that contain leaks from source to sink are as follows: `A: %0 -> %idxprom -> %arrayidx` and `B: %call1 -> %rem -> %idxprom -> %arrayidx`. Notice that B's chain starts with the `%call1` instruction which is the actual source of the secret, whereas A's chain starts with the `load` that is a result of the previous store instruction. When compiling with `-O0` there's a higher chance for leaky paths to have fewer instructions in common, potentially resulting in fewer opportunities to place minimal cuts.

8.5 Development Details

The source code for this project can be found under [6] and utilizes a docker container for reproducibility. The whole implementation exists within a single file under the `llvm/lib/Transforms/Utils` directory and is registered as an optimization pass enabled via the `-p blade` flag for the `opt` program. Although the optimization requires to be performed at link time so that it brings all translation units into scope, due to time constraints and for practical reasons, LLVM-Blade was developed as a standard optimization that ran after Clang's standard set of optimizations over a single translation unit, however, it can be trivially deployed at link time as well. The testing of this project involved compiling C source code to LLVM IR and then running the modified `opt` executable over the LLVM IR files to result in LLVM IR files with LLVM-Blade protections inserted. Evaluations then compiled the resulting IR to executable programs for testing.

9 Evaluation

The solution that Blade proposes stops speculative execution with a minimal amount of insertions, which inherently worsens performance. To test what kind of performance hits LLVM-Blade introduces, the following experiment involved compiling a selection of crypto algorithms from the HACL* Library [13]. This library is a collection of formally verified common crypto algorithms. The following strategies have been tested when compiling the library and benchmarking code.

- **Baseline**: No modifications were done to the optimizer - no protections were inserted.
- **LB**: The implementation of LLVM-Blade as described in this paper using fences.
- **LN**: Fences inserted after every ***transient*** instruction to prevent secret leaks naively.

Once the C source code was compiled with the `-O3` optimization level, benchmarks provided by the HACL* Library were executed and the runtime of each algorithm was recorded. Once 20 consecutive tests were made, an average was recorded. After that, for each algorithm, the average runtimes of **LB** and **LN** strategies were divided by the **Baseline** runtime to get a *Runtime Increase Factor (RIF)*. This number represents the factor by which the runtime of a given strategy increases compared to the Baseline. Additionally, the number of fences inserted by both strategies was recorded. All tests and

experiments were conducted on an *IdeaPad 5 Pro 16ACH6* Laptop with a *16-core AMD Ryzen 7 5800H* CPU and 16GB of RAM.

Since LLVM-Blade implements the protect statement with fences, the evaluation compares the equivalent strategy in the existing implementation of Blade [4]. However, three out of seven benchmarks conducted in the original paper stem from the cryptographic code of the CT-wasm [14] repository which has constant time cryptographic code implementations in Webassembly. However, using a tool to translate Webassembly to LLVM-IR could potentially break any constant time guarantees, which is why those tests have been purposefully left out of this evaluation. Finally, the original Blade benchmarking repository [15] was downloaded and run locally to produce measurements that are comparable to the benchmarks of LLVM-Blade.

9.1 Results

Table 1 uses the following acronyms in its columns for brevity and readability:

- **LB** refers to the LLVM-Blade strategy.
- **LN** refers to LLVM based Naive strategy which inserts fences at all ***transient*** instructions.
- **B** refers to the original WebAssembly implementation [4] from Blade paper [1].

Columns with "Fences" indicate the number of fences inserted by the strategy while columns with "RIF" refer to the *Runtime Increase Factor* and indicate by what factor the runtime of the given algorithm and strategy increase compared to the baseline. *RIF* values close to 1.0 indicate no significant change in performance. Furthermore, the columns involving the original implementation of Blade (**B**) only show results for the 4 final rows of Table 1, since only those benchmarks were made available by the benchmarking repository of Vassena et al. [15] and due to time constraints tests for the remaining algorithms were not adapted to the Webassembly implementation.

Algorithm	LB. Fences	LN. Fences	B. Fences	LB. RIF	LN. RIF	B. RIF
Salsa20 64 bytes	2	734	-	1.007	7.143	-
SHA2 (32-bit)	0	6	-	0.996	0.999	-
Blake2 8192 bytes	2	38	-	1.011	1.007	-
ED25519 32768 bytes	0	62	-	1.0	1.0	-
K256_ECDSA 16384 btyes	0	13	-	1.111	1.053	-
ECDH Curve25519	2	2,702	235	1.005	3.0	1.34
ChaCha20 8192 bytes	2	156	3	1.002	1.009	1.033
Poly1305 1024 bytes	2	84	3	1.006	1.002	1.003
Poly1305 8192 bytes	2	84	3	1.01	0.999	1.029

Table 1: Runtime for various algorithms with Blade and naive strategies compared against the Baseline with no protections.

Comparing columns 2 and 3 of Table 1, we can see that LLVM-Blade places vastly fewer fences than the naive approach and would also result in code size reduction. However, as seen in columns 5 and 6, the change in performance for both **LB** and **LN** compared to the baseline for most algorithms is mostly insignificant. This has to do with the intricacies of the algorithm itself and whether or not ***transient*** instructions exist along the *fast-path* of the program. However, *Salsa20* and *ECDH Curve25519* algorithms are examples where LLVM-Blade greatly outperforms the naive solution.

However, due to register spilling there is the potential for values to be loaded from memory instead of from registers which leads to missing placements of protection (also mentioned in Section 7.1 and Section 10). Because of this, it is inconclusive whether or not there is a performance gain over

the original Webassembly implementation of Blade, despite the test for *ECDH Curve25519* suggesting so. It is also highly dependent on the source code itself whether or not there's a clear performance improvement when using LLVM-Blade over the naive solution. Furthermore, since the original implementation of Blade compiles Webassembly which is then run by a runtime (WASI), this makes it very difficult to directly compare LLVM-Blade's native executables to the performance of Webassembly runtime. Finally, the key takeaway from the results above is that implementing Blade for LLVM is certainly possible and would offer wide adoption potential due to LLVM's vast reach in the space of programming languages. Being able to secure native machine binaries with LLVM-Blade also brings performance benefits over being confined to a Webassembly runtime.

10 Limitations and Future Work

While the scope of this project was to understand the specifics of the Blade implementation inside the LLVM framework, many observations were made along the way about the implementation of Blade regardless of which compiler framework is being used. The following is a list of limitations pertaining to the LLVM-Blade implementation.

- **Register Spills** - Depending on the underlying hardware and decisions made by the machine code generator, data stored in registers might be spilled out and stored on the stack resulting in load instructions that aren't taken into account Blade.
- **SLH Protections** - Speculative Load Hardening is a method to prevent speculative execution by assuring that control flow following `load` instructions preserves validity without branching. However, SLH requires a more involved implementation [16] and goes beyond the scope of this project, but could be implemented alongside LLVM-Blade.
- **Efficient Graph Representation** - The *dummy-vertex* inefficiency outlined in Section 8.3 is a result of the matrix representation used for the directed graph traversal. Certain aggressive compiler optimizations can hugely inflate the number of instructions within a function to improve performance, leading to a large memory footprint of the matrix representation used. Work can be done to improve this memory footprint by finding a more efficient way to perform the Min-Cut algorithm over the function's dependency graph.
- **Potential Improvements to Blade** - Further work outlined by the original Blade paper [1], contains points that would also apply to the LLVM implementation of Blade. LLVM-Blade only implements protections via fences which could be improved upon further by also implementing SLH. Additionally, LLVM's vast ecosystem of static analysis information that is not present in the current implementation of Blade [4] might lead to new insights on better ways to insert protections beyond the Min-Cut approach.

11 Conclusion

The implementation [6] of Blade as an LLVM compiler pass enables the many languages that use the LLVM framework to be protected from Spectre V1 attacks at compile-time. LLVM-Blade minimally inserts fences to cut the data flow of data dependencies that would lead to a secret leak under speculative execution. This paper discussed the discrepancies that LLVM optimization levels produce when analyzing the data flow of transient to stable instructions to prevent secret leaks. It also pointed out inefficiencies of the data structures used by the Min-Cut algorithm that allows for minimally inserting protections in the first place. Evaluations show that in some cases LLVM-Blade incurs almost no performance overhead while the naive approach significantly reduces performance when compared to not placing any protections. Finally, due to register spills and inherent differences between native executables and Webassembly runtimes, a direct comparison between LLVM-Blade and the original implementation is inconclusive.

12 Appendix

1. The following LLVM instructions are annotated with the ***transient*** metadata node.
   - `llvm::Instruction::Load`
   - `llvm::Instruction::Call`
   - All arguments of a given function
2. For every instruction `I` in a BasicBlock the following is performed to annotate with the ***stable*** metadata node.

```
for (User *U : I->users()) {
  if (Instruction *II = dyn_cast<Instruction>(U)) {
    if (II->getOpcode() == Instruction::Load) {
      markInstructionStable(I);
    }
  }
}
```

Bibliography

[1] M. Vassena, C. Disselkoen, et al., “Automatically eliminating speculative leaks from cryptographic code with blade,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021, doi: 10.1145/3434330. [Online]. Available: https://doi.org/10.1145/3434330

[2] P. Kocher, J. Horn, et al., “Spectre attacks: exploiting speculative execution,” *Commun. Acm*, vol. 63, no. 7, p. 93, Jun. 2020, doi: 10.1145/3399742. [Online]. Available: https://doi.org/10.1145/3399742

[3] C. Carruth, “Speculative load hardening¶.” [Online]. Available: https://llvm.org/docs/SpeculativeLoadHardening.html

[4] “Lucet-blade,” 2020. [Online]. Available: https://github.com/PLSysSec/lucet-blade

[5] C. Lattner, and V. Adve, “Llvm: a compilation framework for lifelong program analysis & transformation,” in *Int. Symp. Code Gener. Optimization, 2004. CGO 2004.*, vol. 0, 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[6] M. Gallup, “Maxgallup/llvm-blade: bachelor's project implementing blade as a link time optimization in llvm.” [Online]. Available: https://github.com/maxgallup/llvm-blade

[7] Wikipedia contributors, “Ford–fulkerson algorithm --- Wikipedia, the free encyclopedia,” 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ford%E2%80%93Fulkerson_algorithm&oldid=1136325806

[8] Wikipedia contributors, “Max-flow min-cut theorem --- Wikipedia, the free encyclopedia,” 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Max-flow_min-cut_theorem&oldid=1124909077

[9] D. NL, “Microprocessor side-channel vulnerabilities (cve-2017-5715, cve-2017-5753, cve-2017-5754): impact on dell products.” [Online]. Available: https://www.dell.com/support/kbdoc/nl-nl/000180621/microprocessor-side-channel-vulnerabilities-cve-2017-5715-cve-2017-5753-cve-2017-5754-impact-on-dell-products?lang=en

[10] J. Engelen, “Link time optimization(lto), c++/d cross-language optimization.” [Online]. Available: https://johanengelen.github.io/ldc/2016/11/10/Link-Time-Optimization-LDC.html

[11] “Llvm link time optimization: design and implementation.” [Online]. Available: https://www.llvm.org/docs/LinkTimeOptimization.html#multi-phase-communication-between-liblto-and-linker

[12] Wikipedia contributors, “Static single-assignment form --- Wikipedia, the free encyclopedia,” 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Static_single-assignment_form&oldid=1157695726

[13] F. Kiefer, and P.-N. Madelaine, “Hacl*, a formally verified cryptographic library written in f*.” [Online]. Available: https://github.com/hacl-star/hacl-star

[14] “Ct-wasm,” 2018. [Online]. Available: https://github.com/PLSysSec/ct-wasm-ports

[15] “Blade,” 2020. [Online]. Available: https://github.com/PLSysSec/blade

[16] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, “Ultimate slh: taking speculative load hardening to the next level,” 2022. [Online]. Available: https://eprint.iacr.org/2022/715 (Cryptology ePrint Archive, Paper 2022/715)