# Documentation

This document outlines the changes made to the NotQuiteJava (NQJ) compiler[1] in order to support Object-Oriented Programming (OOP) constructs in the name and type analysis, as well as in the translation to LLVM [2]. This includes multiple sorts of design choices, such as the error generation paradigm, utilized data structures, and algorithm descriptions.

## 1 Data structures

This section describes the data structures extended and newly introduced to support OOP constructs throughout the NQJ compiler.

### 1.1 Class type representation

To support OOP, classes are introduced to the reference types.

The type *ClassType* caches field and method (member) types including such inherited, as well as a reference to the superclass. Members have their types stored by their names in *HashMap* structures, which makes lookups (for *MethodCall* and *FieldAccess*) for type evaluation efficient, and handles both method overriding (i.e. a subclass method replacing a superclass method with the same name) and field hiding (i.e. a subclass field hiding a superclass field with the same name) by design due to key uniqueness (i.e. there can only be one field and method respectively by a specific name).

The *ClassType* type extends *Type* and as such needs to implement the method *isSubtypeOf(Type other)*, which returns whether *other* is a supertype of the class type, and in consequence whether the current class can be upcast to *other*. The following algorithm describes the high-level idea of how this is implemented by *ClassType.isSubtypeOf()*.

1. If *other* is equal to *Type.ANY*, return true.
2. Otherwise, if *other* is not a valid *ClassType* instance, return false.
3. Otherwise, if *other* is equal to the current *ClassType* instance or a superclass', return true.
4. Otherwise, return false.

Step 1 ensures the convention that all types are compatible with *Type.ANY*, which is important to the error generation design (ref. section 2.6). Step 2 ensures that other non-class types are incompatible. Steps 3 and 4 ensure that exactly the current class and all of its superclasses are compatible classes.

**Note**: The indices of *ClassType* are used exclusively during the analysis phase. As such, the member indices quite literally hide hidden fields and overridden methods by using their names as unique hashing key. Thus, *ClassType* indexes exactly the visible members. The translation

[1] As of exercise sheet 4 [1]

phase constructs LLVM structures from scratch based on the AST class declarations, which contain the hidden and overridden declarations.

### 1.2 Type context

The type *TypeContext* is used to store the symbol and type context for the name and type analysis, which consists of local variables and their types, the type of *this*, and the return type of the current function. It is extended by a *HashMap* storage for fields that works the same way as the one for variables. The reason for the separation is the fact that fields may be shadowed by variables unlike other variables. In consequence, variables are queried first and fields are queried if and only if a variable with the requested name cannot be found.

### 1.3 Newly introduced indices

New indices are introduced to both the analysis and the translation phases analogous to the existing ones. When indexing values unique per class, for example references to the corresponding *ClassType* instances, this is done by the class name, as it is unique among classes and results in less data to hash than if the key was the AST class declaration itself. For values unique to methods however, the AST method declaration is used, as method names may not be unique globally and forging a unique name-based key would be overcomplicating the design internally. Every AST method declaration is unique as it contains both references to its parent (i.e. the source class) and its name, which is a unique tuple.

## 2 Name and type analysis phase algorithms

This section describes the algorithms extended and introduced to support OOP constructs in the NQJ compiler analysis phase.

### 2.1 Class indexing

Class indexing is required to efficiently evaluate class-related expressions. The following algorithm describes the high-level idea of how this is implemented by *NameTable.getClassType()*.

1. If a previously indexed class type exists, return it.
2. Otherwise, record the current class as visited.
3. Repeat from step 1 for the superclass, if existent. If the class is encountered a second time, terminate the class inheritance walk. Add analysis errors appropriately.
4. Initialize the member indices for the current class based on the superclass', if existent.
5. Expand the member indices with the current class' definitions. Add analysis errors appropriately.
6. Construct and publish the class index instance.

Step 1 accelerates class resolution by caching. Step 2 builds an index to check for inheritance cycles. Step 3 ensures superclasses are processed before subclasses, so that the superclass member indices are available to subclasses in step 4, and that the graph is acyclic. Step 4 makes sure all methods and fields of a superclass are inherited to its subclasses. Step 5 expands the index by the current class' own member definitions. Because *Map* structures with the member names as keys are used, field hiding trivially works and method overriding does not yield duplicate methods. Step 6 finalizes the class index structure and publishes it for resolution.

## 2.2 Class declaration Visitor run

The existing Visitor run is extended to accept all class declarations of the top-level dictionary. The following algorithm describes the high-level idea of how this is implemented by *Analysis.visit()*.

1. Index the visited class (section 2.1). Errors are added appropriately by the indexing algorithm.
2. Construct a new type context for method visits, set the *this* type of and add all field information to the context.
3. Visit the method list with the constructed context.

Step 1 indexes the fields necessary for the Visitor context construction and implicitly visits the class *extends* declaration and the field declarations (analogous to how blocks and function declarations work). Step 4 prepares the state to allow for analysis of the methods. Step 3 performs analysis on all methods, which reuses the existing Visitor for functions.

## 2.3 Field access expression checker

The existing expression checker is extended to check field accesses. The following algorithm describes the high-level idea of how it is implemented by *ExprChecker.case_FieldAccess()*.

1. If the receiver (i.e. the expression that is subject to the operation) is the *this* reference and the source (i.e. the function or method the expression originates from) is a function, add analysis errors appropriately and return *Type.ANY*.
2. Otherwise, if the receiver is not a class instance, add analysis errors appropriately and return *Type.ANY*.
3. Otherwise, if there is no field with that name in the receiver class, add analysis errors appropriately and return *Type.ANY*.
4. Otherwise, resolve and return the field type.

Step 1 ensures that *this* is only used in methods. Steps 2 and 3 ensure that the field access applies to a class instance that contains the specified field. Step 4 returns the type of the expression result, so that type analysis can be performed.

## 2.4 Variable use expression checker

The existing variable use expression checker is extended to allow for implicit field accesses and field shadowing. The following algorithm describes the high-level idea of how it is implemented by *ExprChecker.case_VarUse()*.

1. If a local variable with the requested name exists, return its type.
2. Otherwise, if the function is a method, and if a field with the requested name exists in the current class, return its type.
3. Otherwise, add analysis errors appropriately and return *Type.ANY*.

Step 1 ensures that variables shadow fields by taking precedence in the lookup. Step 2 ensures that implicit field accesses (i.e. such that omit *this*) are supported correctly. Step 3 ensures correct error behaviour.

## 2.5 Method call expression checker

The existing expression checker is extended to check method calls. The following algorithm describes the high-level idea of how it is implemented by *ExprChecker.case_MethodCall()*.

1. If the receiver is the *this* reference and the source is a function, add analysis errors appropriately and return *Type.ANY*.
2. Otherwise, if the receiver is not a class instance, add analysis errors appropriately and return *Type.ANY*.
3. Otherwise, if there is no method with that name in the receiver class, add analysis errors appropriately and return *Type.ANY*.
4. Otherwise, if the passed arguments are incompatible with the parameters or the amounts mismatch, add analysis errors appropriately and return *Type.ANY*.
5. Otherwise, resolve and return the method return type.

Step 1 ensures that *this* is only used in methods. Steps 2 to 4 ensure that the method call applies to a class instance that contains the specified method, including the parameter list. Step 5 returns the type of the expression result, so that type analysis can be performed.

## 2.6 Error generation

One of the most important tasks of a compiler infrastructure actually is error generation. It is not only relevant to report errors at compile time, but also to provide feedback within code editors (e.g. clangd [3]). In consequence, False Negatives (i.e. errors not emitted despite being valid errors, e.g. because the analysis is aborted early) and False Positives (i.e. errors emitted despite not being valid, e.g. missing member errors about objects with unresolved types) are limited by design.

The name and type analysis is not terminated prematurely when an error is encountered. Instead, it continues on a “best knowledge” base. This means, when a class type cannot be

resolved, it is substituted with the generic *Type.ANY* type. This instance is compatible with every type (ref. *isSubtypeOf()* implementations) and as such does not cause type errors on its own. This allows the name and type analysis to further progress. Also, class member analysis aborts early to not generate superfluous errors that cannot be addressed directly.

To further enhance error relevance, the indices of cyclic classes are fully populated, which means that every class has all visible members indexed. This is done by terminating the class inheritance walk when a class is encountered twice (which guarantees termination), while still processing this second occurrence as if it had no superclass. This way, all subclasses of this class inherit its members as expected. While its own member indices lack the members of its superclasses due to the path termination, this index is not actually published. The first resolution call for this class constructs and publishes a valid index with all members inherited from its superclasses. This way, analysis is performed without errors regarding absence of inherited members.

# 3 Translation phase algorithms

This section describes the algorithms extended and introduced to support OOP constructs in the NQJ compiler translation phase.

## 3.1 Virtual Method Table (VTable) construction

VTable construction is required to support dynamic method dispatching. The following algorithm describes the high-level idea of how it is implemented by *Translator.constructClassStructs()*.

1. Initialize the LLVM structure and the constant instance with the data of the superclass.
2. For all method declarations in the class declaration:
   a) If the VTable already contains a field with the method name, the method is overriding. Update the procedure address with the location of the overriding method.
   b) Otherwise, add a new entry to the VTable with the same information.
3. Publish the VTable information (structure, and constant instance as global variable).

Step 1 ensures that subclass VTables can be upcast to superclass VTables (i.e. former is binary-compatible to latter), which is required for inheritance in class upcast scenarios. Step 2 ensures that method overriding and introducing new methods work as expected. Step 3 ensures that method call translation has access to all required information.

**Note**: One Virtual Method instance per class is stored as a constant global variable. This is possible because they are immutable at runtime, and because they are unique only to the class type, not to the class instance.

## 3.2 Class structure construction

Class structure construction is required to support field accesses and method calls. The following algorithm describes the high-level idea of how it is implemented by *Translator.constructClassStructs()*.

1. Initialize the class' LLVM structure and add a field for the VTable.
2. Append all fields of the superclass structure to the class structure (excluding the VTable pointer).
3. Append all fields of the class declaration to the class structure.
4. Publish the class structure information.

Step 1 ensures that the VTable pointer is always at the same conventional position. Step 2 ensures that subclass structures can be upcast to superclass structures (i.e. former is binary-compatible to latter), which is required for inheritance in class upcast scenarios. Step 3 ensures that field hiding and introducing new fields work as expected, most notably that hiding fields do not actually replace hidden fields. Step 4 ensures that field access translation has access to all required information.

## 3.3 Class translation

Class translation is divided into separate stages (initialization, construction, and translation) to be able to handle complex dependency graphs, especially those that are cyclic. The following algorithm describes the high-level idea of how this is implemented by *Translator.translateClasses()*.

1. For all AST class declarations of the program (initialization):
   a) Index an empty class structure for the class.
   b) Queue the class and all of its superclasses in descending order (subtype relation) for construction. Do not add duplicates.
2. For all queued classes (construction):
   Construct the VTable (ref. section 3.1) and populate the class structure (ref. section 3.2) of the class.
3. For all queued classes (translation):
   Translate all methods of the class.

Step 1.a ensures that further class processing as part of steps 2 and 3 can fetch references to all referenced classes. Step 1.b is used to implement inheritance, as for all subclasses the VTable and the class structures are constructed based on the superclasses' structures. Step 2 constructs the structures required for class instantiation. Step 3 is required to correctly translate the methods of the current class and those of all class referencing it (e.g., field accesses, method calls).

## 3.4 Class instantiation translation

The following algorithm describes how class instantiation is performed at program runtime. The only way NQJ performs class instantiation is by implicit default constructors. The necessary instructions are generated by *ExprRValue.case_NewObject()* and *Translator.createNewClassFunc()*.

1. Allocate heap memory for the class structure. If unsuccessful, throw an OutOfMemoryException error.

2. Otherwise, store the VTable pointer obtained from global variables in the first class structure field.
3. Initialize every class instance field with the default value for its type.

Step 1 ensures there is a reserved memory space for the entire class instance memory. Steps 2 and 3 deterministically initialize the class instance memory to the expected state. The stored VTable pointer allows for dynamic method dispatching.

## 3.5 Field access translation

The following algorithm describes how field accesses are performed at program runtime. The necessary instructions are generated by *ExprLValue.case_FieldAccess()*.

1. If the receiver is null, throw a NullPointerException error[2].
2. Otherwise, retrieve the field offset from the class structure.
3. If the access is a read operation, execute a *load* instruction. Otherwise, execute a *store* instruction.

Step 1 ensures there is no unsafe behaviour invoked by the operation. The handling conforms to the language specification[3]. Step 2 retrieves the field memory location that is required to perform the access. Step 3 performs the actual field access operation based on the field location from step 2.

## 3.6 Method call translation

The following algorithm describes how method calls are performed at program runtime. The necessary instructions are generated by *ExprRValue.case_MethodCall()*.

1. If the receiver is null, throw a NullPointerException error.
2. Otherwise, cast all method arguments according to the method parameter types.
3. Retrieve the VTable pointer from the class structure.
4. Retrieve the method pointer from the VTable structure.
5. Call the method with the cast arguments.

Step 1 ensures there is no unsafe behaviour invoked by the operation. The handling conforms to the language specification[4]. Step 2 ensures all arguments have the expected types as LLVM is strictly typed. Compatibility has previously been ensured by the analysis phase. Steps 3 and 4 locate the method memory location. Step 5 executes a *call* instruction based on the method location from step 4 and the parameters from step 2.

**Note**: Method names are kept unique per class, so method overriding works trivially.

[2] This is part of the original design, provided by *Translator.addNullcheck()*.
[3] 15.6 Normal and Abrupt Completion of Evaluation, 5. [4]
[4] 15.6 Normal and Abrupt Completion of Evaluation, 6. [4]

# Reflection

This document is a reflection on the implementation task of OOP constructs in the NQJ compiler, the "Compilers and Language-Processing Tools" lecture contents, and further topics from the field.

## 4 Changes to the original design

The original design that served as a base for the portfolio work was the sample solution for the fourth exercise sheet of the lecture. As part of implementing the portfolio functionality, only a very few design choices were altered.

### 4.1 Gracious name and type analysis

The error generation component of a compiler infrastructure is a crucial part of the development workflow. False Positives can distract the user from the actual issues and cause confusion, False Negatives may lead to wasted time due to new analysis runs being required after fixing previous errors. But even for the developers of the compiler infrastructure this is crucial to find bugs in the error generation parts — some of the False Positives and False Negatives may not be emitted "by design" but due to hidden bugs.

For this reason I implemented a "gracious" name and type analysis, which aims to reduce both issues in a simple fashion. When a fatal type error is encountered (e.g. an undefined class type), an error is emitted and type *Type.ANY* is used for representation. *Type.ANY* is compatible with all types, thus implicit casts to them are allowed. Also, operations like field accesses and method calls do not emit errors when performed on *Type.ANY*. Hence, consequential errors are reduced to a minimum. At the same time, when encountering a non-fatal type error (e.g. an inheritance cycle), a representing type is chosen or constructed by best knowledge to allow the name and type analysis to proceed further. For example, every class in an inheritance cycle is constructed such that it contains all members of all other classes in the cycle as well as its own. In consequence, accessing fields and calling methods that were not defined at all will yield errors, while when they are defined somewhere in the class cycle, the only error will be the one regarding the inheritance cycle itself.

## 5 Lessons learned

The task of implementing a compiler is not trivial in the slightest. There are many design considerations to be made, and a lot of functionality is to be validated. I learned plenty of small lessons during the implementation and testing phases, and the largest ones are worth pointing out in this section.

## 5.1 Importance of consistent design

Not having a full, consistent design plan in mind can lead to bugs in unexpected places. While implementing the gracious name and type analysis, an incautious mixture between *Type.ANY* and *Type.INVALID* led to the exact kind of behaviours the approach was designed to prevent, such as False Positive error messages. Also, out of sync requirements between the analysis and the translation phases have caused multiple sorts of unexpected error output and False Negatives with the error unit tests. It is important to have a clear design plan reasonably ahead of time, and to have code in place to catch errors in the design's implementation. To approach this, I introduced *IllegalStateException*s to plenty of places in order to assert that certain conditions hold rather than just expecting them to.

## 5.2 Importance of simple design

For the initial design of the compiler, I had decided to attempt translating only such classes that are referenced as a primitive form of translation optimization. While the approach worked reasonably well at first, it complicated the code design in various ways. Most notably, no matter the implementation details, there is an intrinsic need to implement some form of reentrancy. This is due to the fact that while translating one class, another new class may be discovered, that then needs to be queued for translation as well. This caused hidden bugs where not all possible spots for new class discovery were covered "in time" (i.e. it was possible that reentrancy happened in a non-reentrant section of the algorithm). Furthermore, albeit not a significant issue when reentrancy is implemented correctly, it was integrated into a function that was not intended to start a class translation. As my implementation translated the newly discovered class immediately rather than implementing a queue (which would have had its own downsides), global state (e.g. current procedure and local variable indices) push and pop operations were required. After evaluation of the issues caused by the approach, I decided to abandon this idea and implement translation of all class declarations divided into clear stages (ref. section 3.3). This reduced code complexity, unintuitive design, and the probability for hidden bugs greatly.

## 5.3 Effectivity of unit tests

In many small projects, unit tests are used in a way that tests nothing beyond the most basic functionality of specific code functions. Often they are redundant considering the manual testing performed on the entire system, and test only for common inputs. For something of the scale of a full albeit basic compiler, unit tests have shown to greatly enhance testing of the entire system for edge cases (e.g. forbidden operations, pitfalls such as field shadowing, etc.) rather than just specific code functions. They were especially useful for regression testing when changing the design to allow for processing more complex constructs (e.g. cyclic dependencies). Having automated procedures to test functionality of complex constructs, such as inheritance, field shadowing, and method overriding, was a great mean of Quality Assurance in the development process. However, the most useful type of tests were the "error" tests that ensure invalid constructs (e.g. inheritance cycles, duplicate class names, and undefined symbols) actually emit analysis errors. Erroneous inputs are often not tested during typical functionality validation.

Refactorings to extend support for complex constructs more than once introduced regressions that were successfully caught with the help of error unit tests.

# 6 Lessons to be learned

Of course the field of compilers is far greater than what can be explored with a semester project. Even industry-class solutions have their own quirks and drawbacks that require more research to be done now and in the future. I will briefly discuss the areas I find most interesting and important.

## 6.1 Effectivity of automated memory management

The trend towards "safer" programming languages imposes the need for automated memory management. With manual memory management, there is a high risk of introducing safety defects, such as "use after free", "double free", or "memory leaks". In response to these risks, automated memory management approaches have surfaced, such as "Reference Counting" and "Mark-and-Sweep".

However, all those approaches have their own disadvantages. For example, Reference Counting may have a low overhead as there is no separate routine required to free unused memory, but it requires support from the developer to resolve dependency cycles, which puts the responsibility to avoid memory leaks partially back to manual, explicit actions. Mark-and-Sweep does not suffer from this particular issue, but it imposes higher requirements on the CPU as it is implemented by a separate thread, which also makes ensuring timing and resource constraints harder.

It is without doubt that the current solutions for "Garbage Collection" work very well for a very wide area of software, yet it is not on the same level of manual memory management that has been formally verified due to the drawbacks discussed above. How proofs regarding memory leaks could be implemented internally, and how resource consumption could be reduced overall for automated memory management, in my opinion, is a very interesting aspect of compiler infrastructures that yet has plenty of exploration left to be done.

## 6.2 Verifiability of safety

As mentioned regarding memory management (ref. section 6.1), developers are prone to introducing safety issues into codebases by nature, but it is not exclusive to memory management. There are all sorts of other safety threats, such as overflow arithmetic and usage of uninitialised memory. This is why modern languages such as Java and C++ have implicit default constructors, references, and other safety features. There is no definite answer on how to handle overflow arithmetic at this point — it is undefined behaviour in C++ [5], defined two's complement arithmetic in default Rust (i.e. Release mode) [6], and a forbidden operation that results in a trap in Swift [7]. However, optimizers have basic capabilities of proving arithmetic safety [8] by e.g. value range propagation, and future advancements could allow for efficient compile-time verification of the operation safety.

## 6.3 Verifiability of security

Security is a superset of safety which introduces an intelligent attacker into the model and especially in the modern world is becoming more relevant as information because more valuable. Only in recent years, attention has been drawn to code security problems that exceed issues of the algorithmic implementation. Especially side-channel attacks can very well be feasible on code that does not have a flaw in the performed operations themselves, but more so the implementation details of the targeted architecture cause "data leaks" and similar issues, as can be seen with the *Meltdown* and *Spectre* families of vulnerabilities [9]. The industry solutions have started to implement automated mitigations [10] [11], yet some threats still require manual developer intervention to fully mitigate [12]. Ideally, latter solutions can be implemented by compile-time algorithms in the future to statically prove side-channel security for compiled programs, and this sounds like a very interesting field of research.

## 6.4 Verifiability of transformations

Compiler optimizations are generally highly regarded today, and definitely are a very integral component of modern software development with regard to performance. Yet, all industry-class compilers have been found various times to produce erroneous output for correct code [13] [14]. For this reason, academic solutions such as CompCert have started to utilize formal methods to verify the correctness of transformations. An evaluation of the effectivity of the approach was overwhelmingly positive [13].

CompCert already can achieve an impressive level of performance for its compiled binaries as of now, but it still trails behind the industry solutions unfortunately [15]. Yet, it is undoubtedly a very promising approach that I hope the industry will adopt over time, and that I am personally interested in researching further at some point.

## 6.5 Ahead of Time (AOT) and Just in Time (JIT)

AOT and JIT have some obvious advantages over one another. AOT applications have a faster initial startup times and cause less overhead, as there is no need for an interpreter. JIT applications can dynamically optimize code based on live profiling data. Yet, there is no definite answer on when to use one over the other. An interesting approach for the Android Operating System suggests neither, but to use a middle way between the two [16]. It proposes that "hot" (very frequently used) functions should be AOT compiled in the background, while the rest remains JIT compiled. Especially modern devices like smartphones require the best of both worlds, and additional considerations to be made, such as the impact on battery life. Not only the future development of JIT techniques, but also smart combinations with AOT techniques are very interesting subject to further increase overall software quality.

# 7 Conclusion

The field of compilers and language-processing tools is very broad, and yet the lecture managed to capture a lot of the foundations in great detail. While I arrived with a background in Assembly and C development (including OOP patterns, analogous to how they are implemented with LLVM), I still learned a lot of things, especially about the early stages lexical analysis and parsing. Of course lexical analysis and parsing are not the stages with the most potential for innovation. Yet, solid knowledge of how they work and of possible culprits (e.g. shift/reduce conflicts) can be very helpful in understanding the design of present languages and architectures, as well as guide the way for evolutionary improvements for existing and new compiler infrastructures.

Another very interesting section was the one about the methods of automated memory management, as both low-level and high-level engineers alike rarely directly deal with them, which imposes the risk of their culprits being forgotten. Personally, I had a wrong conception of how Reference Counting worked prior to the course, and was not aware of some alternatives like Mark-and-Sweep. I think knowledge of their internal functioning is an important key to writing code that is easy for them to deal with (e.g. avoiding cyclic references, or utilizing weak references).

The portfolio task was a very interesting hands-on project. The culprits explained in the earlier sections all added up to various learned lessons. Unlike some similar tasks, it taught about educational topics as much as about implementation techniques and culprits common in the industry. The chance at failing (and hopefully subsequentially succeeding) to implement a compiler can be an even better teacher than even the best lecture materials to some people.

All in all, the lecture contents were very informative and covered a good amount of ground overall, and even with pre-existing knowledge, provided plenty of useful information, techniques, and lessons for people of all backgrounds, including myself.

## References


[1] Albert Schimpf. NQJ Compiler (exercise sheet 4). Available at https://pl-git.informatik.uni-kl.de/clp20-groups/clp_solutions/-/tree/master/ex4. Online. Accessed 22nd February 2021.

[2] The LLVM Project. The llvm compiler infrastructure. Available at https://llvm.org/. Online. Accessed 22nd February 2021.

[3] The LLVM Project. clangd features. Available at https://clangd.llvm.org/features.html. Accessed: 2020-02-21.

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification*. Oracle America, Inc., Java SE 15 edition, 2020.

[5] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[6] The Rust Project Developers. The Rust Reference: Operator expressions. Available at https://doc.rust-lang.org/reference/expressions/operator-expr.html. Online. Accessed 20th February 2021.

[7] Apple Inc. The Swift Programming Language: Basic Operators. Available at https://docs.swift.org/swift-book/LanguageGuide/BasicOperators.html. Online. Accessed 20th February 2021.

[8] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming Undefined Behavior in LLVM. *SIGPLAN Not.*, 52(6):633–647, June 2017.

[9] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[10] Chandler Carruth. Speculative Load Hardening. Available at https://llvm.org/docs/SpeculativeLoadHardening.html. Online. Accessed 20th February 2021.

[11] Daniel Donenfeld. More Spectre Mitigations in MSVC. Available at https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020. Online. Accessed 20th February 2021.

[12] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498. USENIX Association, August 2020.

[13] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[14] Microsoft Visual C++ Team. Visual C++ 2015 Update 2 Bug Fixes. Available at https://devblogs.microsoft.com/cppblog/visual-c-2015-update-2-bug-fixes/, 2016. Online. Accessed 20th February 2021.

[15] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[16] Rahul Kandu. Android: The Road to JIT/AOT Hybrid Compilation-Based Application User Experience. Available at https://software.intel.com/content/www/us/en/develop/articles/android-the-road-to-jitaot-hybrid-compilation-based-application-user-experience.html, 2016. Online. Accessed 20th February 2021.