Type Annotation for SAST

Marco Pereira
Checkmarx, Braga, Portugal
University of Minho, Braga, Portugal

Alberto Simões
Checkmarx, Braga, Portugal
2Ai, School of Technology, IPCA, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre/ LASI, DI-University of Minho, Braga, Portugal

Abstract

Static Application Security Testing (SAST) is a type of software security testing that analyzes the source code of an application to identify security vulnerabilities and coding errors. It helps detect security vulnerabilities in software code before deployment reducing the risk of exploitation by attackers.

The work presented in this document describes the work performed to upgrade Checkmarx's SAST tool allowing the execution of vulnerability detection taking into account expression types. For this to be possible, every expression in the Document Object Model needs to have a specific type assigned accordingly to the kind of operation and to the different operand types.

At the current stage, this project is already supporting the expression type annotation for three programming languages: C, C++ and C#. This support has been done through the addition of a new Resolver Rule to the Resolver stage, allowing for the generalization of languages. We also compare the complexity of writing vulnerability detection queries with or without access to type information.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Software and its engineering → Compilers

Keywords and phrases Static Application Security Testing, Type Annotation, C, C++, C#

Digital Object Identifier 10.4230/OASIcs.SLATE.2023.12

Funding *Alberto Simões*: This paper was funded by national funds (PIDDAC), through the FCT - Fundação para a Ciência e Tecnologia and FCT/MCTES under the scope of the projects UIDB/05549/2020 and UIDP/05549/2020.
Pedro Rangel Henriques: This work has been supported by FCT - Fundação para a Ciência e Tecnologia within R&D Units Projects Scope: UIDB/00319/2020.

1 Introduction

Static Application Security Testing (SAST) is a fundamental tool for the software development life-cycle. Although SAST is limited to detecting vulnerabilities and software issues during compile time, not being able to find run-time bugs, it is able to discover a large variety of problems. The number of results depends on the quality of the developed queries that find vulnerable code structures. But, these queries' effectiveness is constrained by the information the language parsers are able to extract and infer from the source code.

This section will start by discussing what is SAST and type annotation and concludes with a brief description of the main goals of this work. Section 2 discusses some related work, following Section 3 with an introduction to Checkmark's product pipeline. The implementation of the type annotation system is depicted in Section 4 and in Section 5 we

© Marco Pereira, Alberto Simões, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY 4.0
12th Symposium on Languages, Applications and Technologies (SLATE 2023).
Editors: Alberto Simões, Mario Marcelo Berón, and Filipe Portela; Article No. 12; pp. 12:1–12:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

discuss the changes needed to support C, C++ and C# with the same code base. Finally, Section 6 compare the complexity of vulnerability detection queries before and after the availability of type annotation. Section 7 concludes with some remarks and future work.

1.1 Static Application Security Testing

Static Application Security Testing (SAST) [9] is a kind of security testing that is performed on the source code of an application to identify security vulnerabilities and coding errors before the application is deployed. It analyzes the application's source code for potential security flaws such as input validation errors, buffer overflow vulnerabilities, and SQL injection flaws. SAST tools use a combination of pattern matching, data flow analysis, and other techniques to identify vulnerabilities, providing detailed information about each issue detected and guidance on how to remediate them.

SAST is an essential part of the software development life cycle (SDLC) because it helps detect and address security issues early in the development process, reducing the cost and effort required to fix them later on. By identifying potential vulnerabilities before the code is deployed, SAST reduces the risk of exploitation by attackers and helps ensure the security and integrity of the application.

Usually, SAST scans an application before the code is compiled, and thus is known as white box testing.

1.2 Checkmarx's Engine

Checkmarx's primary product is a SAST Engine responsible for processing projects in various programming languages. To guarantee language independence the engine produces a generic structure, named DOM (Domain Object Model). Thus, the DOM is a language-agnostic structure that contains information on variable declarations, assignments, conditions, expressions and so on. The DOM is then queried by different queries in order to find vulnerabilities in the code like SQL Injections. Each query is created following the flow of the vulnerability it tries to detect, looking either for code constructs or to data-flows.

1.3 Type Annotation

Programming languages can be statically typed or dynamically typed (or even, in some special situations, completely non-typed). For typed languages, each variable or expression yields a type, either specified clearly (for static typing) or inferred by the context (for dynamic typing) [5, 12].

Most programming languages allow declaring variables with a specific type. This type can range from a byte or character, an integer, used to represent whole numbers, to floating point values or even pointers and objects.

Along with the type of a variable, there can be information on its size. The most common example is declaring an integer, where it can be specified to take two, four, eight or sixteen bytes (in C the default integer usually takes four bytes and the remaining sizes can be stated using the keywords `short`, `long` or the repetition `long long`). There can be also information on the signedness of the variable: whether the variable can represent both positive and negative values (the default) or only positive ones.

1.4 SAST Problems solved with Type Annotation

There are different kinds of software vulnerabilities that can be tackled using type information. This section describes one real-world example that falls in the class of Integer Overflow situations.

Integer Overflow, also known as Wraparound, has been ranked number 13 in 2022 CWE Top 25 vulnerabilities [6] and has been consistently ranked top 15 in the past years. The vulnerability consists of trying to store a too-large (or too-small) value in a variable, whose data type is not able to deal with. If we consider an integer variable, with default size and signedness modifiers, in a C program running on a 64-bit machine, it will be able to store up a maximum value of 2 147 483 647. This means that, if a variable of this type is yielding this value and is incremented (for example, using the increment operator), the value will wrap, and the variable will have the value -2 147 483 647.

An example of an integer overflow that results in a buffer overflow was detected in an older version of OpenSSH (v3.3) [1]. For the code presented in Listing 1 the variable `nresp` is obtained through a user-controlled function. This means that if the variable gets any value higher than $\frac{1}{8} \times$ `MAX_UINT`, the expression inside the `xmalloc` function will wrap (here we consider that we are running in a 64-bit machine, where a pointer is eight bytes). Given that `xmalloc` function parameter is an unsigned integer, the wrapped value will be too small to deal with the size of the packed being received, and data will be written in non-reserved memory.

Listing 1 OpenSSH 3.3 excerpt vulnerable to integer overflow

```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(char*));
    for (i = 0; i < nresp; i++)
       response[i] = packet_get_string(NULL);
}
```

While SAST is unable to detect the value present in the `nrest` variable, it can detect that variables of the same type are being multiplied and that their values are not checked for possible overflows.

1.5 Objectives

The goal of the development described in this document is to create a proof of concept system that adds information about types to each declared variable and each and any expression that uses them. This information should be as complete as possible, including not only the type itself, but also its size or signedness details, or even the target type in case of arrays, pointers or objects.

Given that Checkmarx's tool is language-independent, the developed solution must be extensible to support statically or dynamically typed languages with the same code base.

2 Related work

There is not much work related to type annotation for SAST. Nevertheless, in this section we present some related work at different levels: on how to represent types, on type inference tools and on tools that rely on type information to find code vulnerabilities.

A comparison can be performed with the way Checkmarx deals with languages and the way LLVM (formerly known as Low-Level Virtual Machine) does. LLVM is a collection of modular and reusable compiler and tool-chain technologies. It provides a set of tools

for building compilers, code analyzers, debuggers, and other related software. The key idea behind LLVM is to provide a set of low-level, platform-independent instructions and optimization steps that can be used to build high-level language compilers.

In LLVM, type annotations are added to the intermediate representation during the front-end stage of compilation. The front end is responsible for translating the source code of a programming language into the LLVM intermediate representation, much like the way Checkmarx's Engine constructs a Domain Object Model (DOM). To represent types, the LLVM project provides a type system that is designed to be platform-independent and flexible. It includes built-in types such as integers and floating-point numbers and user-defined types such as structures and arrays. Figure 1 presents a diagram that describes all the implementations of the LLVM type system [11].

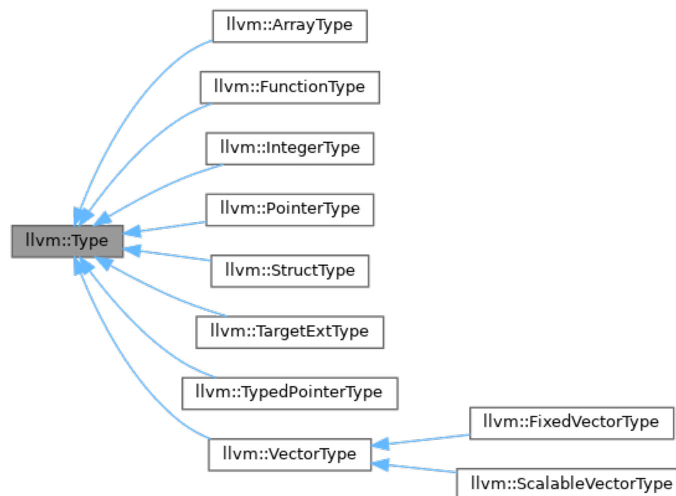

Figure 1 LLVMType diagram.

Type inference, by itself, is common, especially for dynamically typed languages. Note that these language interpreters can benefit from run-time to perform their type analysis. Nevertheless, there are works for static type analysis for these languages. For the Python programming language, such an example is presented by [4]. A more complex example for type inference [3] has the goal to infer types from code executable files, and therefore, not having any hints from the code itself.

A technical report [7] from the University of California at Berkeley focuses on a similar problem as some of the queries being developed at Checkmarx that will take advantage of type inference. Their goal is to detect signedness conversions that can cause application bugs. Their work is developed on top of the Valgrind [8] tool.

3 Checkmarx's Pipeline

In this section, some information on the current state of Checkmarx's engine pipeline will be provided for context.

Initially, the source code gets parsed by ANTLR [10], generating a parse tree. This tree is then used by two sets of visitors. The first set of visitors creates a global symbol table for the variables found, trying to store their type information, accordingly to their declaration type (statically typed languages) or based on the values that are assigned to them. The second set of visitors produces an abstract syntax tree (in fact, it is called Universal Abstract

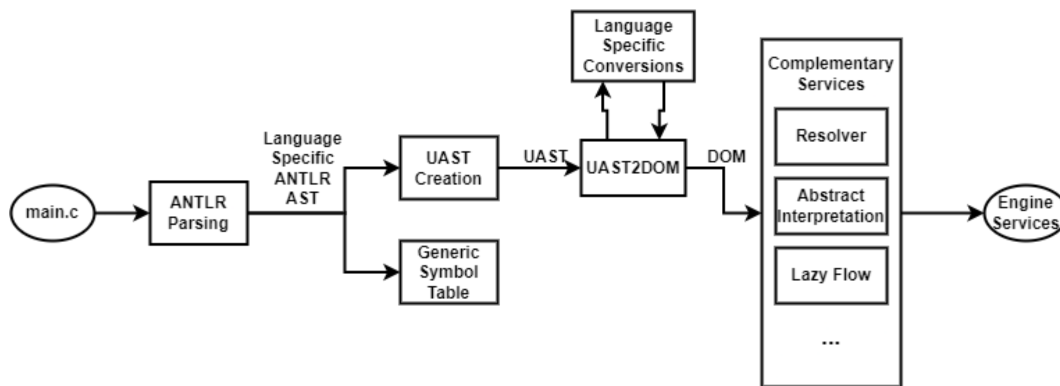

Figure 2 Checkmarx source code parsing pipeline.

Syntax Tree – UAST – as it is almost language independent). This tree is then converted to a Domain Object Model (DOM) tree, that is shared among all programming languages. The vulnerability scans run on top of this tree. A set of other services are then run on top of this model, namely a resolver, that finds internal references for symbols, an abstract interpretation engine, or even a lazy data-flow service. This structure is depicted in Figure 2.

Follows an elaborated explanation of the relevant steps involved in the type annotation process.

Symbol Table

The first stage of the pipeline is responsible for generating the symbol table. A set of visitors traverse the syntax tree constructed by the ANTLR grammar that gathers information about declared variables and literal values. Thus, some information on types is obtained at this point, as well as some type modifiers. Nevertheless, this information is not available for expressions, as no resolution of symbols is performed.

Listing 2 shows the type information collected by this stage that will be useful for the type annotation process. The example shows some information about an `unsigned long` variable. type associated with each variable contains, among other data, a list of modifiers and the name of the type. The list of modifiers will contain all the strings of the type that aren't the base type name, in this case, the two modifiers present are "unsigned", the signedness modifier of the type, and "long", the size modifier of the type. The `Modifiers` array can also contain strings like "static" and "volatile". The name of the type is "int" as that is the type declared implicitly for an `unsigned long` variable.

Listing 2 Information on the type and its modifiers for an `unsigned long` expression after the Type Inference stage

```
Type: {
    ...
    Modifiers: [ unsigned, long ],
    Name: int
    ...
}
```

UAST Creation

The parsing tree is traversed a second time using another set of visitors to create a Universal Abstract Symbol Table (UAST). This tree structure is shared among other languages, with minor language-specific details. Regarding types, the UAST does not add much information, it just compiles the data collected in the symbol table and adds it to the correct tree nodes.

UAST to DOM Conversion

The Domain Object Model (DOM) is a tree structure shared among all programming languages. This step processes the UAST in order to create this DOM structure. In the DOM, type information is much more clear and straightforward as can be seen in Listing 3. This Listing relates to the same data presented in Listing 2. At this stage the type information is saved in a class named "TypeRef", there are class variables which save each of the type-related modifiers instead of having them all on a single list.

Listing 3 DOM information on type modifiers for `unsigned long`

```
TypeRef: {
    TypeName: int,
    TypeSignedness : Unsigned,
    TypeSize : Long,
    ...
}
```

Resolver

The Resolver is one of the complementary services that uses the information available in the DOM to link all references, namely variables and functions with their definitions. This information is crucial for the type annotation process, as the return type of an expression that uses a variable requires the knowledge of that variable's type, as well as the return type of an expression that uses a function call, requires its return type.

4 Development

This section starts by discussing the different points, in Checkmark's engine pipeline, where the type annotation process could be implemented, and the challenges that arise from that approach. Follows the description of the implemented solution.

4.1 Solution Discussion

Three different stages were analyzed as possible implementation points for the type annotation system:

1. The first one was using the first pipeline step when the global symbol table is constructed. While this stage would allow easy implementation, the drawback is how specific is this stage accordingly with the language begin analyzed. This would mean that future improvements and the addition of type annotation on other languages would require the replication of similar code
2. The second approach was using the traversal done when converting the UAST into the DOM. The implementation of the type annotation step at this point would have little overhead, as the traversal through every tree node was already being performed either way.

This would, also, be a much better solution in terms of allowing for the generalization of different languages. The implementation would be a simple visitor for each node type in the DOM structure.

Unfortunately, the type of declared variables is not associated with the calls of said variables at the UAST2DOM stage, this means that no information on variable types is present. Thus, the type annotation implementation had to be moved further to the end of the pipeline.

3. There are complementary services that run on top of the final DOM structure. As already referred, one of those services is the Resolver. The Resolver is composed of rules, each one of them responsible for a different task. For example, one of them is responsible for the attribution of types to variables, functions and method calls. Adding a new rule at the end of all the other Resolver rules would allow access to all the information gathered by the previous steps. Thus, the information on the types would already be present and allow for the expression type annotation with all the information needed.

Taking into account the different approaches, the development was performed considering the addition of a new Resolver rule. It supports the infrastructure for type annotation for all languages, as the DOM structure that supports the type annotation engine is language-independent. The main challenge, which will be discussed in the next section, is to deal with different types from different languages, as well as the different type-inferring mechanisms that each one of these languages implements.

4.2 Type Annotation Implementation

As discussed before, the solution was implemented as a new rule in the Resolver engine. As expected, this approach requires a new traversal of all nodes in the DOM structure. However, given only expressions will have a type (statements do not have an inferred type), the solution is able to select only nodes that inherit from the expression node type, mitigating this problem to some extent.

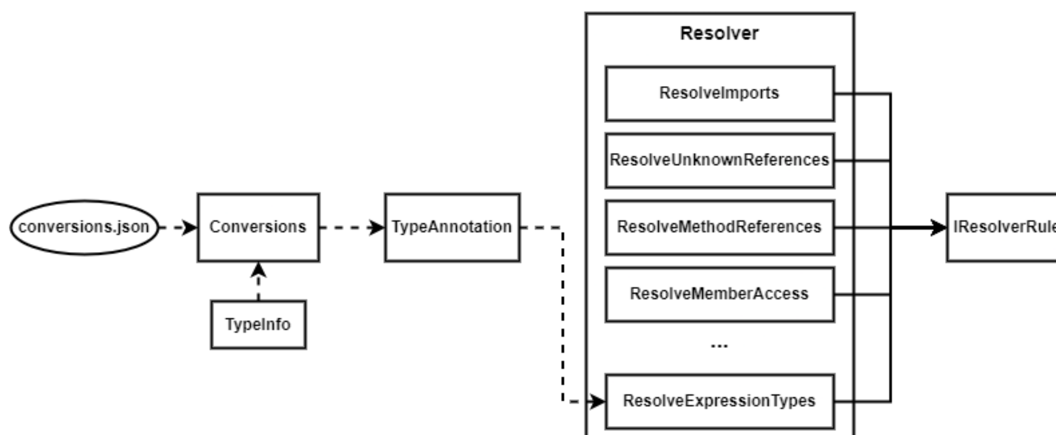

Figure 3 Resolver architecture.

Figure 3 shows that the Resolver is composed of various rules, each of them implementing the `IResolverRule` interface. This interface defines a main method that is called by the Resolver for each one of the rules. Along with this method, for the rule that is implemented, the constructor method is also be very important.

The constructor method is responsible for calling the TypeAnnotation class constructor and retrieving, from every node in the final tree, those of type *Expression*. These expressions are then processed by the mentioned Execute method, which iterates through the expressions and calls the AddAnnotatedType method.

The dotted lines simply denote files/classes that are used. The TypeAnnotation class, for example, is used inside the ResolveExpressionTypes rule.

4.2.1 The TypeAnnotation Class

The TypeAnnotation class goes through the different types of expressions (binary and unary operators, method invocations, casts, etc) and calls the DetermineExpressionType method.

This method is polymorphic, meaning that there is a different implementation for each expression type, thus making the code maintainable and easily extensible, and also recursive, as some expression types depend on the types of the child expressions. Thus, every time a node is visited, the process looks for the existence of the annotated type and returns if it exists. This is required given that the class constructs a list with all the expressions in the DOM structure.

The type annotation is stored as a custom attribute, meaning that no change was required in the base DOM object structure. To decide which type to assign to each expression another class, named Conversions, is used. It is responsible for deciding the type of a node given the child node types. For example, to decide on the type of a binary expression the method looks for the type of the left and right operands, to the operator.

4.2.2 The Conversions Class

The Conversions class is responsible for providing the TypeAnnotation class with the type resultant of different operand and operator combinations. It has two different stages that are equally important: a first stage, implemented in the class constructor, and a second one implemented in the ResultOf methods that are called by the TypeAnnotation class:

1. The constructor of this class uses JSON file, as shown on Figure 3, that describes, for each node type, operator or method, and operands, the inferred result type. This file content is loaded and a structure is created with the required information for dealing with the expression type annotation.
   This file uses a domain-specific language in order to reduce the number of repetitions. Details on this syntax are described in the next Section. For example, it is possible to specify that, for binary operators, all the combinations of operand types will always have the same result, regardless of the specific operator.
   In order to support different languages a new feature has been added, that will allow this main conversions file to be overridden, for specific node types. For example, considering the C programming language, the result of the division of two integer operands is, also, an integer, but in other languages, like Python, the result would be a floating-point value. So, for these exceptions, a Python-specific conversion file will be created that will just specify the main behaviour changes when compared with the base conversions file.
   The conversion information is stored on an instance of the TypeInfo structure. This structure is a simple type representation that contains basic type information: the type name (char, integer, etc.) and its signedness and size properties.
2. The ResultOf method uses this information. It receives all the operand types present and the operator (or other information like the name of the method being invoked), and the type of the expression that is being processed.
   It then iterates the map containing the corresponding expression type conversions and returns the TypeRef resultant of the received operands and operator combination.

4.2.3 The conversions JSON file

As explained previously, there is a JSON file that keeps all the combinations of operands, operators and nodes, as well as the inferred expression type for that node. The main focus of this file is to keep all this information in a simple and easily processable file, and also readable by the programmer.

Listing 4 Excerpt from `conversions.json`

```
"BinaryOperator": {
    "*": {
      "bool": {
        "default": "itself"
      },
      "char": {
        "Signed Short int": "Signed Short int",
        "Unsigned Short int": "Unsigned Short int",
        [...]
        "float": "float",
        "double": "double",
        "Long double": "Long double",
        "default": "char"
      },
      "Signed Long int": {
        "Unsigned Long int": "Unsigned Long int",
        "Signed LongLong int": "Signed LongLong int",
        [...]
        "Long double": "Long double",
        "default": "Signed Long int"
      },
      "Long double": {
        "default": "Long double"
      }
    }
  }
```

Consider the binary operator and the C programming language as an example. The resulting type for a binary operator expression is only dependent on the operands and not on the operator itself (except for the comparison operators, whose resulting type is a Boolean). To reduce the number of entries in the conversions file, the operator can be replaced by a special symbol: the asterisk (‘*’). Note that the operators are not represented by their symbol but by an internal name. Therefore, this asterisk will not conflict with the multiplication operator. Listing 4 shows a partial excerpt of the binary operator rules.

The C programming language always selects the more precise type from a binary operator. If the first operand is a Boolean value, then no matter what the second operand is: the inferred expression type will always be the other operand type. As another example, for long double values, there is no more precise type so the resulting type will always be a long double. The “default” keyword means any other type not specified in that rule, just like the `default` keyword in a `switch` statement.

There is also a “`itself`“ keyword that is used in conjunction with the “`default`” keyword. It means that the resulting type is always the type matching the *default* keyword.

5 C, C++ and C#

While the prototype started for the C and C++ programming languages, there was the need to guarantee that the approach is easily extensible for other programming languages. Before trying a dynamically typed language, where SAST will not be able to perform much

Listing 5 Excerpt from the `R10_03.cxql` query.

```
CxList assignExprs = Find_AssignExpr();
CxList paramDecls = Find_ParamDecl();
CxList decls = All.NewCxList(Find_Declarators(), paramDecls);
CxList unknownReferences = Find_UnknownReference();
CxList enumDecls = Find_Enum_Declarations();
CxList realLiterals = Find_RealLiterals();
CxList integerLiterals = Find_Integer_Literals();
CxList charLiterals = Find_CharLiteral();

CxList boolDecl = decls.FindByType("bool");
CxList charDecl = decls.FindByType("char");
CxList enumDecl = All.FindAllReferences(enumDecls) - enumDecls;
CxList shortDecl = decls.FindByTypeModifiers(TypeSizeModifiers.Short);
CxList intDecl = decls.FindByType("int")
         .FindByTypeModifiers(TypeSizeModifiers.Default);
CxList longDecl = decls.FindByTypeModifiers(TypeSizeModifiers.Long);
CxList floatDecl = decls.FindByTypes("float", "double", "long␣double");

// Get all references from essential types
CxList boolDeclRefs = unknownReferences.FindAllReferences(boolDecl);
[...]
// Find for integer literals with 'L' suffix
CxList literalsWithSuffix =
  integerLiterals.FindByRegex(@"[1-9]?[0-9]+[lL]?", false, false, false);
[...]

// Checks if the type of return statement references is different from
    ↪ the return type of the methodDecl
foreach(CxList retRefs in returnStmtRefs)
{
 CxList methodDeclReturn = retRefs.GetAncOfType<MethodDecl>();
 methodDeclReturn = methodDeclReturn.CxSelectDomProperty<MethodDecl>(
     ↪ method => method.ReturnType);

 CxList types = paramDecls.FindDefinition(retRefs);
 types = types.CxSelectDomProperty<ParamDecl>(parDecl => parDecl.Type);

 string retTypeName = types.CxSelectElementValue<TypeRef, string>(x => x.
     ↪ TypeName).FirstOrDefault();
 string methodDeclRetTypeName = methodDeclReturn.CxSelectElementValue<
     ↪ TypeRef, string>(x => x.TypeName).FirstOrDefault();

 if (methodDeclReturn != null && retTypeName != null &&
     retTypeName != methodDeclRetTypeName)
   result.Add(retRefs.GetFathers().FindByType<ReturnStmt>());
}
[...]

// Add to result non-addition assign expressions that have char
// on the left side and real, integer literals on the right side
addToRefs.Add(integerLiterals);
assignExprs -= assignAddExprs;
result.Add(getRefs(charDeclRefs, addToRefs, assignExprs));
```

inference at compile time, tests were performed for C#. While not too different from C, it has some relevant differences that allowed the understanding of possible limitations.

The implementation of the C# language was made simply by adding support for expressions that do not exist in C, such as the Try/Catch expressions, although this expression exists as a statement, it can never be used as an assignment to a variable in C/C++ contrary to C#.

C# also adds the Throw as an expression. Nevertheless, it works as a statement, as the evaluation of the current expression is interrupted and the exception is raised. Thus, its return type is irrelevant.

Another issue is the type system. C# does not support type modifiers. Instead, there are different types, one for each combination of signedness and size. For example, the type "`long`" in C# is the equivalent of a "`long int`" in C/C++. Thus, these new types need to be taken into account in the conversions file. As these types do not overlap with the C/C++ ones, they can be described in the same file without any issue.

6 CXQL Query exploration

Checkmarx uses queries written in a Domain Specific Language (DSL) based of C#, known as CxQL, to look up vulnerabilities in the DOM. Each query searches for a specific vulnerability in a specific language (although there are some exceptions).

Before the addition of expression type annotation queries that needed type information were hard to write. Consider Rule 10.3 from MISRA [2] (a group of safety guidelines for writing code for the automotive industries). It states that "*The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.*"

This rule is currently implemented without any annotation of types. This query is quite large and not readable, as can be seen in Listing 5. The code is very repetitive, with more than 200 lines of code. It is partially replicating the type annotation process but in query time. Also, other queries rely on similar information and repeat most of this code to be able to infer expression types.

The code gets simpler by rewriting this query using the information provided by the type annotation system. Listing 6 shows the complete new query. It has less than 50 lines of code and is more maintainable. Testing the query in a project of 177 558 lines of code, we notice that the original query takes 8.17 seconds, while the new version takes 7.86 seconds. Even though it provides a speed improvement, it should be noted that the main gain of this query is to allow for more readable and understandable code.

7 Conclusion

This document presents the implementation of a POC for a type annotation system for Checkmarx's SAST engine. While the tool is still in a proof-of-concept stage, it has proven to be possible to extend to other statically typed languages and its usage in query development makes them more readable and maintainable. In fact, the type of code assessment that can be done with this new information is larger, allowing Checkmarx to develop new directions for vulnerability scans.

Currently, there are some main directions to improve this prototype and integrate it officially:

Listing 6 Rule `new_R10_03.xql`

```
var expressionsWithAnnotatedType = All.FindExpressionWithCustomAttribute(
    ↪ "AnnotatedType");
var assignments = All.FindByType<AssignExpr>();

var typeSizeModifiersRanking = new Dictionary<TypeSizeModifiers,int>{
   { TypeSizeModifiers.LongLong, 1 }, { TypeSizeModifiers.Long, 2  },
   { TypeSizeModifiers.Default,  3 }, { TypeSizeModifiers.Short, 4 }}

var typeNameRanking = new Dictionary<string,int> {
    {"double", 1}, {"float", 2}, {"int", 3}, {"char", 4}, {"bool", 5}};

foreach (CxList assign in assignments) {
 var leftTypeList = assign.CxSelectDomProperty<AssignExpr>(x => x.Left);
 var rightTypeList = assign.CxSelectDomProperty<AssignExpr>(x => x.Right)
     ↪ ;
 var assignExpr = All.NewCxList(leftTypeList, rightTypeList);

 if ((expressionsWithAnnotatedType * assignExpr).Count == 2) {

   TypeRef leftAnnotatedType =
      leftTypeList.CxSelectDomProperty<Expression>(x =>
        (x.CustomAttributes.First(attr => attr.Name == "AnnotatedType")
        .Parameters[0] as TypeOfExpr).Type).GetFirstGraph() as TypeRef;
   TypeRef rightAnnotatedType =
      rightTypeList.CxSelectDomProperty<Expression>(x =>
        (x.CustomAttributes.First(attr => attr.Name == "AnnotatedType")
        .Parameters[0] as TypeOfExpr).Type).GetFirstGraph() as TypeRef;
  try {
   if (
    typeSizeModifiersRanking[leftAnnotatedType.TypeSize] >
        ↪ typeSizeModifiersRanking[rightAnnotatedType.TypeSize]
    || (
    typeSizeModifiersRanking[leftAnnotatedType.TypeSize] ==
        ↪ typeSizeModifiersRanking[rightAnnotatedType.TypeSize]
    && typeNameRanking[leftAnnotatedType.ResolvedTypeName] >
        ↪ typeNameRanking[rightAnnotatedType.ResolvedTypeName]
    )
    || leftAnnotatedType.TypeSignedness != rightAnnotatedType.
        ↪ TypeSignedness)
   {
    result.Add(assign.Clone());
   }
  }
  catch (Exception e) {
   result.Add(assign.Clone());
  }
 }
}
```

- Explore new statically typed languages that are farther from C. As an example, Java or TypeScript are probable candidates.
- Attempt type annotation for dynamically typed languages, like Python, Perl, JavaScript or Ruby. We expect that many expressions will be annotated with a generic *any* type, but that some others might be possible to annotate properly. For some of these languages, like Python, code hints can be used in the source code to specify types. This information can be used for the type annotation process.
- Upgrade the proof-of-concept to handle more complex data types, record-like (classes, struct), arrays, pointers or objects. While some of these will prove to be a challenge, others will be simpler to support.

References

1 Acunetix. What is integer overflow? https://www.acunetix.com/blog/web-security-zone/what-is-integer-overflow. Accessed on May 26, 2023.
2 Motor Industry Software Reliability Association. *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems*. HORIBA MIRA, 2019.
3 Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), May 2016. doi:10.1145/2896499.
4 Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. Static type inference for foreign functions of python. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 423–433, 2021. doi:10.1109/ISSRE52982.2021.00051.
5 Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020. doi:10.1145/3421472.
6 MITRE. Cwe top 25 list (2022). https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. Accessed on May 26, 2023.
7 David Alexander Molnar and David Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical Report UCB/EECS-2007-23, University of California at Berkeley, February 2007.
8 Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. doi:10.1145/1273442.1250746.
9 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1999. doi:10.1007/978-3-662-03811-6.
10 Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.
11 The LLVM Project. LLVM compiler infrastructure and tools. https://llvm.org/, accessed 2023.
12 Stuart M. Shieber. *Constraint-Based Grammar Formalisms: Parsing and Type Inference for Natural and Computer Languages*. MIT Press, Cambridge, MA, USA, 1992.