FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

InjectionCop

A Pluggable Type Checker for Inferencing Custom Type Qualifiers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Racz, BSc
Matrikelnummer 0627659

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof.Dr. Wolfgang Kastner
Mitwirkung: Dr. Christian Platzer

Wien, 07.09.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

InjectionCop

A Pluggable Type Checker for Inferencing Custom Type Qualifiers

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Michael Racz, BSc
Registration Number 0627659

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner
Assistance: Dr. Christian Platzer

Vienna, 07.09.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael Racz, BSc
Untere Umfahrungsstraße 44, 2432 Schwadorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Die erste Erwähnung gebührt meinen Eltern Karin und Alois Racz. Sie haben mich in jeder Phase meines Lebens voll und ganz unterstützt, und mir den Zugang zu Bildung, insbesondere meinem Studium, ermöglicht. Dieses uneingeschränkte Engagement, dieser Wille zur Förderung ist nicht selbstverständlich. Danke!

Ich möchte mich auch beim Team um *re-motion*, allen voran Michael Ketting und Stefan Wenig, für die Gelegenheit einen Beitrag an einem Open Source Framework zu leisten und den Zugang zu Know-how im Bereich der industriellen Software Entwicklung bedanken.

Besonderer Dank geht an Christian Platzer und das gesamte *Seclab*. Für die Betreuung dieser Arbeit, aber auch für Praktika, Vorlesungen und Erlebnissen wie der Teilnahme am UCSB iCTF. Es wurde mir wertvolles Wissen vermittelt und das Interesse für Security geweckt.

Abstract

The importance of security in the field of software development cannot be emphasized enough. Already known vulnerabilities like all kinds of injections are still very common. InjectionCop is a framework that helps to address cross cutting security concerns by adding custom type qualifiers to the type system of *C#*. Type qualifiers can be used to boost the expressiveness of the source code and define security requirements that are executable. Furthermore, requirements that are not met indicate vulnerabilities and are detected by static analysis of the assembly. InjectionCop is targeted to support and improve the development process by reducing the risk to introduce programming errors that affect security.

Kurzfassung

Die Relevanz von Security auf dem Gebiet der Software Entwicklung kann nicht genug hervorgehoben werden. Bereits bekannte Sicherheitslücken, wie zum Beispiel Injections, sind immer noch weit verbreitet. InjectionCop ist ein Framework um Security-relevante Anforderungen, die nicht nicht einem spezifischem Modul, sondern über das ganze System verteilt sind, zu behandeln. Um das zu bewerkstelligen, wird das Typsystem von *C#* um das Konzept des Custom Type Qualifiers erweitert. Type Qualifier können genutzt werden, um die Ausdrucksstärke von Quellcode zu verstärken und ausführbare Security-Anforderungen zu definieren. Anfoderungen, die nicht erfüllt werden, entsprechen Sicherheitslücken und werden über statische Code Analyse ermittelt. InjectionCop zielt darauf ab, den Entwicklungsprozess zu unterstützen und zu verbessern, indem das Risiko reduziert wird, einen Programmierfehler zu begehen der die Sicherheit des Systems beeinflusst.

Contents

List of Figures

Listings

CHAPTER 1

Introduction

This section gives an overview of the thesis by defining the problem domain InjectionCop is designed for. Additionally, the expected result and the methodology are outlined. The section concludes with a discussion of the relevance to the curriculum.

1.1 Problem

Nowadays, most modern applications are web-based, which reinforces importance of security requirements. With the rise of agile development techniques, leading to requirements changing and evolving throughout the development process, it is essential to address non-functional requirements comprehensively. There are means of testing non-functional requirements, but correlation with corresponding types and methods in production code is difficult. Furthermore, it is essential to extend types and methods so that violations of non-functional requirements are recognized by the type system at compile time.

Security related issues, like all kinds of injections, sending encrypted data only, or tracking immutable objects for ensuring thread safety, can be mapped to an instance of the source sink-problem. In this context, the source sink problem refers to denoting those parts of the code that provide untrusted data, like user input, as well as denoting sinks that demand trusted data to operate securely. Tools like the open source framework *re-motion* [62], which includes an object relational mapper and web controls, need to assemble *SQL* statements and validate user input with exactly this kind of problem. An algorithm addressing this problem has to deal with the complex task to ensure that validation is done on every possible path from any source to any sink and untrusted data is not injected between data-validation and data-processing. Tracking and verifying the data-validation process guarantees that only trusted data is processed. Modern programming languages provide a powerful type system and feature adding meta information to code constructs by annotations, but a mechanism using these features to target the source sink-problem is still missing.

1.2 Expected result

The outcome of this work should be a pluggable type checker called InjectionCop, extending the type system of *C#*, and enforcing its expressiveness by addressing the source-sink problem. InjectionCop should be mature enough to be utilized in a productive environment. The goal is to introduce custom type qualifiers to the *re-motion* open source framework and address cross cutting security concerns reliably in the development process. The framework should provide basic annotations that should be easy to extend and customize for defining type qualifiers which specify sources and sinks according to the problem domain. Type qualifiers are monomorphic, and therefore straight forward to use without extensive declaration but still expressive enough for most problem instances. The core module of the framework is an analyzer engine which is capable of inspecting an assembly and identifying source-sink violations. Additionally, the framework should be standalone and integrate into the build process and integrated development environment seamlessly. Another important feature is an *XML* interface for integrating third party libraries conveniently.

The essence of this master's thesis is building upon existing frameworks and rationale from the field, reducing complexity where possible and reasonable, but also adding features that are crucial for feasibility in an advanced software engineering process. The benefit of InjectionCop is an inference engine that improves a developer's means of dealing with aspects that are non-functional and is easier to use than existing solutions.

1.3 Methodology

The pluggable type checker is implemented as a custom *FxCop* rule which is basically a dynamic link library containing code for analyzing assemblies by means of static code analysis. *FxCop* provides facilities for integrating this rule into *Visual Studio* as well as executing analysis via command-line. Furthermore, these front ends bring along a handy GUI and open several possibilities for applying the analysis during the build process. The rule performs static code analysis and static annotation checking on intermediate language code. By analyzing an object model of the assemblies code, which is built by *FxCop*, a problem specific graph representation is generated. The resulting graph is an instance of the source-sink problem that is analyzed by the core engine. The core engine's task is to identify possibilities of untrusted data consumed by a sink. Therefore, an algorithm for solving the problem instance has to be defined and implemented. The result of the analysis is a collection of problem objects referencing the corresponding parts of the source code. Depending on the front end, the collection is presented in a tab of *Visual Studio* or written to an *XML* output file. For defining and qualifying sources and sinks, a separate dynamic link library containing basic annotations has to be created. To enable third party library integration, an *XML* interface is designed and implemented. The interface complements InjectionCop annotations and adds type qualifiers to code that is not modifiable. The *XML* qualification file can be tested for correctness with the help of an *XML Schema* file.

To enforce high code quality, and to handle the complexity of user input that is inherent when analyzing assemblies, the software is developed in a test-driven way. To gain information regarding suitability in practice a performance test is performed on the core algorithm. [27, 36, 50]

1.4 Structure of the thesis

The thesis begins with a discussion of related work. Static and dynamic code analysis whith an emphasis on security, formal methods that are applied in a security problem domain, and relevant theory of type systems are covered. Furthermore, related tools and frameworks are introduced and evaluated. Specific fields of application are outlined by highlighting use cases where InjectionCop is a perfect fit. Based on this foundation, design decisions and details of the verification engine, which is based on formal methods of the analysis engine, are explained. Another important aspect is the actual usage of InjectionCop. This includes annotations in source code, as well as integration into the build process and the development environment. An overview of the architecture and the overall analyis process provides a big picture of all components that are involved when InjectionCop is used. Additionally, some selected implementation details are presented to give insights about used design patterns. The thesis concludes with a performance anlysis, potential issues that were identified and future work.

CHAPTER 2

Related work

This chapter covers a discussion about detecting malicious code by means of code analysis. Methodologies of different approaches that are applicable for InjectionCop are emphasized. Furthermore, the significance of formal methods in malware- and code-analysis, as well as theoretical foundations for extending a type system are addressed. The chapter concludes with an evaluation of frameworks enabling the extension of a type system and frameworks implementing an extension of a type system.

2.1 Static and dynamic code analysis

Advanced static code analysis operates on a *control flow graph (CFG)*. Each vertex of the graph is a *basic block*, which is a sequence of instructions that is executed entirely if the first instruction is executed. Since a *basic block* cannot contain jump instructions, the edges of the graph model code jumps between *basic blocks*. A *CFG* is a suitable data structure to analyze programs and derive possible execution paths by following paths in the graph. *CFGs* are also heavily used in the field of compiler construction and compiler optimization to tackle problems like dead code elimination and various loop optimizations [18].

Static code analysis has some advantages compared to dynamic analysis. Obviously, it is not necessary to execute potentially malicious code, there is no need to set up an analysis environment. Furthermore, the analyzed program cannot detect and bypass the virtual environment to remain undetected. Another advantage of static analysis is that the code coverage is 100 percent, because the analyzed execution paths are independent from the input and the runtime behavior of the program [45].

Based on the *CFG*, taint analysis can be applied by defining sources of potentially harmful input and sinks, for example vulnerable system calls. By traversing the graph vulnerabilities can be detected when sources and sinks are connected [45]. This approach is prone to produce false positives, which is one of the major problems of static code analysis when the outcome is not filtered. Picking up this idea, algorithms that perform a more fine grained analysis have been proposed. Instead of just seeking for library functions, the focus is on behavior. Patterns of vulnerable functions like `strcpy` are defined and detected in the assembly to identify possible buffer overflows [61].

The effectiveness of this approach, and static analysis approaches in general, is affected by the quality of the graph. Several techniques for obfuscating the control flow of binaries have been proposed [4, 59], which distort the *CFG* and evade detection by malware scanners.

Another graph related approach is a *data flow graph (DFG)*, where initial values of variables and transitions of values by instructions or assignments are modeled. The graph is used to detect malicious variable values that are passed system calls or other attack vectors. Also hybrid approaches using *CFGs* and *DFGs* were proposed [65]. In a security related context, the generation of the control flow is often based on parsing binaries [68]. Many generators of this kind rely on well known disassembler-tools like *IDA Pro* [34], *CodeSurfer* [13] or textitObjDump [56] to build the graph.

To circumvent issues introduced by an obfuscated graph, many researchers focused on dynamic code analysis. Contrary to static analysis, which is a white box approach, a black box approach like dynamic code analysis is harder to bypass with this kind of obfuscation. Dynamic analysis is not immune to obfuscation, since, virtualization is not equivalent to transparency. Many ways to detect a virtual machine have been discovered [9] which also target virtual machine managers (VMM) that are specifically designed for security, like *Malware Analysis Virtual Machine Manager (MAVMM)* [5], provide foundation to circumvent dynamic analysis.

In dynamic analysis, information is gathered by analyzing actually executed code paths and data that is passed to vulnerable sinks like system calls, files or network sockets. Usually, the program is run on a virtual machine to provide an isolated environment and utilize virtual machine introspection [3, 33, 40, 46].

2.2 Formal methods for specification and verification

In theory, many problems are solved, but implementing these algorithms is still a challenge. Providing a theoretic approach does not necessarily induce proper tools and frameworks that are applicable in a software development process. Furthermore, designing and implementing frameworks that utilize these algorithms reveal new problems.

To define the behavior of malicious code, approaches based on formal methods evolved. Behavior and patterns of malicious software can be modeled with a tailored logic, where a set of malicious machine instructions is defined by logical formulas [2, 37]. With the help of *model checking* [22], which is a technique to check if a given specification can be derived from a model, executables represented by a *CFG* can be scanned for those patterns. A benefit of a formal specification of malicious code snippets is precision and readability, because specifying low level code in an algorithm is hard to achieve. Also other formal approaches, for example an approach based on inference on a *tree automata* [26], which is a state machine working on tree structures, have been discovered [16]. Like other static approaches, also techniques using formal methods suffer from code obfuscation [66].

Considering theory and performance of prototypes in an artificial environment, using formal methods in context of static code analysis seems to be a promising approach. Prototypes involving model checking and algorithm improvements [38] lack practical applicability due to limited support of the *x86* instruction set or missing support of procedure calls.

The already discussed approaches are based on analyzing binaries, which is a drawback since binaries are difficult to analyze and tools that work on higher level languages can provide features relying on language characteristics. Code optimization can bias the result, or even introduce new security flaws that cannot be revealed by analyzing source- or intermediate language code. The following sample is taken from [7] and illustrates the issue.

```
memset(password, '\0', len);
free(password);
```

The snippet is taken from a security module, where a password is stored in clear text in a buffer pointed to by *password*. Before the buffer is released, and returned to the heap, *memset* [49] is used to fill the buffer with null values to prevent leakage of sensitive information. From a compiler point of view, the call of *memset* has no effect on the algorithm and may be removed.

Also prototypes that operate on *intermediate language (IL)* [42] are promising but not tested for practicability. These prototypes' focus is on further challenges of static code analysis like *virtualization obfuscation*, where the payload is byte code that is executed by an inline interpreter [41].

2.3 Pluggable type checkers

Pluggable type checkers extend the built-in type system of a language by custom modules that implement features the type system does not provide to reduce bugs or enrich the language. Tools for *Java* like *JustAdd* [25], or the *Checker Framework* [12], which is easier to use, exist to create pluggable type checkers. With the help of the *Checker Framework* modules that check for passed null values or verify that a class is immutable [71] have been implemented. Also more sophisticated language extensions like *Mixins*, which is a mechanism to inject code modules into classes [17], can be added with the help of pluggable type checkers [58]. Theory of pluggable type checking and custom type qualifiers is sophisticated, but we are decades away from adopting it in practice [57].

2.4 Custom type qualifiers

Type qualifiers are used to add semantics to a type and foster expresiveness of a type declaration. A well known example of a built-in type qualifier in *C* is *const*, which indicates that a variable cannot be altered after initialization. Custom type qualifiers extend this concept and introduce type qualifiers that are defined by the developer. Theory of type qualifiers was introduced in [39] and initially applied on *C*, which resulted in the framework *CQual* [15]. Effectiveness of the framework was shown by detecting format string vulnerabilities [70], and practicability was enhanced by implementing a plug-in to visualize type qualifier inference [19] for the development environment *Eclipse* [24]. Furthermore, the concept was adopted to *Java* to implement the framework *JQual* [31]. Applications of the framework include a check for *pure methods*, which are methods without side effects [32]. Additionally, theory of type qualifiers was also applied on *C++* [69]. The discussed frameworks for popular languages indicate that the concept of custom type qualifiers has attracted interest and is powerful enough to have high impact on the development process. Most of these frameworks suffer from a limited feature set or a weak type system of the target language, which is not an indicator for a poor language, since choosing language features carefully is a fundamental facet of language design. *C#* is a modern language with a rich feature set and a good candidate for the implementation of InjectionCop.

2.5 JQual

JQual is built upon *CQual* and analyzes source code written in *Java* 1.4 and prior. It is discussed in greater detail because *Java* and *C#* are popular general purpose languages and have many features in common.

Using *JQual* is quite cumbersome. A programmer has to create a file, containing the description of a lattice, to specify relations between qualifiers. A *lattice* is a partial order where for each pair of elements x and y, the least upper bound and greatest lower bound of x and y both always exist [43]. A lattice is a type system theoretical and mathematical background theory of a type system of an object oriented language. The problem with lattice file is that type system internals are exposed, and for the developer the lattice is usually hidden and irrelevant. The average developer is not even aware of the lattice, therefore, defining a lattice is unintuitive and confusing.

Denoting sources and sinks can be done on various positions in the code via annotations. As annotations are introduced in *Java* 1.5, and therefore not available when *JQual* was implemented, comment-based annotations are used. Furthermore, targets for *JQual* annotations include local variables and return values, where annotating the latter is not even possible with annotations provided by *Java* 1.7, which indicates they are not powerful enough to express *JQual* syntax. Comment-based syntax is error prone and is coupled with quite a number of additional problems. It is isolated from the type system, inconvenient in context of IDE integration, and only available in source code. Another drawback of *JQual* is that integrating third party libraries is solely possible by adding corresponding source code, or generating stubs and annotating them properly. This restriction narrows use cases for *JQual*, since any modern application and most security related modules depend on third party libraries.

2.6 FxCop

FxCop [29] is a powerful tool for static code analysis developed by *Microsoft*. It is shipped with *Visual Studio* [51] and bundled with a default set of rules, which are modules that run static code analysis on a given assembly and report any problems found during the analysis. These default rule sets cover code smells, which are shortcomings in code that indicate poor design [1], regarding aspects like architecture, performance, or naming. Besides the predefined default rules, there is also an *FxCop SDK* which adds the possibility to define custom rules, which is a well suited interface and plug-in mechanism for integrating custom static code analysis modules. *FxCop* has several characteristics making it a good fit for developing a pluggable type checker, as well as an engine dealing with type qualifiers. These characteristics include

- analyzing managed code assemblies
- working on *Common Intermediate Language (CIL)* code
- generating an object model of assembly
- easy integration into software development cycle

The first item seems to be more like a restriction than a benefit. FxCop can handle assemblies that are managed, in other words they run in a Common Language Runtime virtual machine. This includes assemblies written in *C#*, *Visual Basic* as well as *managed C++*. The benefit of this design decision shows when looking at the other characteristics mentioned.

The *Common Language Infrastructure (CLI)* is an environment defined by *Microsoft* to execute code on a computing platform. A software component in *CLI* is an assembly, consisting of *Common Intermediate Language (CIL)* code and meta data. *CLI* is designed to support multiple languages, *C#* and *Visual Basic* are compiled to *CIL*. A benefit of the intermediate language is that not every feature of a higher language needs to have an equivalent in *CIL* [23].

Since *CIL* code is analyzed, a simplified but consistent model of an assembly is generated where entities and connectors are equal and language independent. Additional languages would change the model in a way that language dependent modules need to be added, which enforces complexity of a system that already is highly complex by nature. The simplification has an effect on the generated model, since mapping source code to *CIL* code is not bijective. This affects obvious aspects, like the formatting of the source code, but also code changing aspects, like compiler optimizations. Also, information that would be beneficial for user output, like names of local variables, are lost in *CIL* code. By taking a closer look at the consequences of the simplification, it becomes apparent that only modules that target and analyze source code would be affected in a negative way. When dealing with semantics, those simplifications become beneficial because code details that are not crucial for the control flow are excluded. It is much more effective to analyze a code model reduced to code blocks and jumps, than having a model containing constructs for each type of conditionals and loops. Another implication when working on *CIL* code with *FxCop* is that code modifications are not possible. Therefore, means of automatic problem resolution cannot be implemented. Instead the user himself has to change the source code to handle issues identified by the static code analysis module. Again, this disadvantage hardly influences analyzing semantics since resolving semantic issues is automatically very difficult and error prone.

FxCop parses the *CIL* code of the assembly and builds an object model that is very complex. The model is convenient to analyze, and information about the assembly can be accessed quickly, but the high level of detail can be hindering when analyzing for a specific semantic issue.

FxCop is also very good at integrating into the development process. There are several front ends including a standalone GUI, but *FxCop* is also an inherent part of *Visual Studio*. Additionally, *FxCop* can be used via command-line. Therefore, static code analysis can be started from a build- or powershell script [67] conveniently. Depending on the front end, the output can be displayed in a GUI element that references corresponding parts of the code visually, printed on the console or written to an *XML* file.

CHAPTER 3

InjectionCop's fields of application

While the focus of the last chapter was setting the scene and outlining requirements for a useful static code analysis framework, this chapter is about use cases and specific applications of InjectionCop. By exposing problems and solutions, required features, usage and syntax are shown at a greater level of detail. Also, the benefits of using InjectionCop as a extension of the type system are illustrated.

3.1 Injections

When developing web based software, security-driven requirements often involve cross cutting concerns. Code relating to such requirements hardly correlates to specific modules and is spread all around the code base. This characteristic can complicate testing, because unit and integration tests concentrate on modules and interaction of modules. Additionally, since practices that facilitate writing tests before business logic is written like *test-driven development* [8] and *behavior-driven development* [54] are very popular nowadays, especially in combination with agile principles, design and architecture are also affected. Again, this is because those requirements cannot be put into a module that is part of design and architecture.

So there is a need to annotate the code that correlates with a given requirement. Code injections are omnipresent concerns in security and require proper handling. Considering *Cross-Site Scripting (XXS)*, which is an attack where malicious code is injected into web applications [14], it comes down to sanitizing user input before it can do any harm. From a code perspective, the developer can identify type qualifier receivers, which indicate that any passed data that is not properly processed and verified, violates the requirement not to allow *Cross-Site Scripting*. On the other hand, code that does exactly that kind of processing can also be identified and therefore annotated to be a valid qualifier provider. This is illustrated by Listing 3.1. The method `Render` generates *html* out of parameter `data`. Only sanitized data can be passed to `Render`, which is defined by the attribute `Sanitized`. The producer of sanitzed data is method `Sanitize`, which is specified by the attributes `FragmentGenerator` to introduce producer and `Sanitized` to express the *qualified type* of the return value.

Listing 3.1: Annotations to prevent XSS

```
class XSS
{
  public void Render ([Sanitized] string data, HtmlTextWriter writer)
  {
    writer.Write(data);
  }

  [FragmentGenerator]
  [return: Sanitized]
  public string Sanitize(string tainted)
  {
    return "sanitized";
  }
}
```

Note that these methods are contained in the same class to provide a compact example. The class violates the *Single Responsibility Principle*, which states that a class should have exactly one reason to change [47, Chapter 8], and should definitely be decomposed. The design would be better when the class is separated into modules for rendering and sanitizing. The class regarding rendering covers the part of preventing *XSS*, which is convenient, since writing data to an *HtmlTextWriter* is very likely to be distributed and may occur multiple times. The benefit of the annotation is that it extends the interface, hence, every caller is obliged to pass sanitized data to prevent *XSS*.

3.2 Immutable objects

InjectionCop is not just another mechanism for sanitizing methods that competes with *Aspect Oriented Programming (AOP)*, which is also a programming paradigm that deals with cross cutting concerns [30]. Depending on context there are much more use cases for this tool that cannot be handled by *AOP*. Immutable objects, which are objects that do not change the internal state after initialization, are not directly related to security, but increase code quality and are often part of secure coding guidelines [28]. Listing 3.2 shows classes and type qualifiers to enforce thread safety. The setup is similar to the setup shown in the preceding chapter but deals with another issue.

Listing 3.2: Annotations to enforce thread safety

```
class ThreadSafety
{
  public static void NeedsImmutableToBeThreadSafe(
    [Immutable] ParameterType parameter) {}
}

class ParameterType
{
  public readonly int readonlyField = 0;

  [return: Immutable]
  public ParameterType() { }
}

class Main
{
  public void main()
  {
    ThreadSafety.NeedsImmutableToBeThreadSafe(new ParameterType());
  }
}
```

Again, there is a source and a sink producing and demanding thread safe objects. Additionally, the class `Main` represents an occurrence of code relating to the cross cutting concern of thread safety, which is spread all over the code base. The class simply connects a provider with a receiver and does not care about the interface that holds the thread safety requirements as long as there is no concurrency issue introduced, because detecting any possible violation is InjectionCop's task.

This example also shows another distinction to *aspect oriented programming*. In *AOP*, it is defined when an aspect has to be considered. With this approach the "when" is left unspecified. All that is relevant, is that at the time of making a call to the interface the according parameters are matching.

3.3 Additional type meta data

Another use case is providing additional information to existing types (Listing 3.3). It can be very useful to add units or domain specific meta data to objects.

Listing 3.3: Domain specific meta data

```
class CustomTypeQualifiers
{
    public void Calculate([Millimeter] int distance) { }
    public void TransferMoney([IBAN] string iban, [BIC] string bic) { }
}
```

It could be argued that such issues can be handled by the type system too, but there are situations where this is not practicable. A type, wrapping a distance in millimeter, could easily be implemented, but at the same time cancels support for built-in arithmetic operations. Defining a class `Millimeter` would result in defining lots of operator overloads that redefine functionality already provided by the programming language. A different situation exposes in the second method of the sample. "IBAN" and "BIC" should definitely be modeled in the domain, and there is also hardly any implementation work to do for handling all necessary uses of the type. In this case, type annotations can be useful when leaving the domain e.g. rendering *html* containing a string representation of bank data or using third party libraries. Especially when sensitive data, like bank accounting data, is processed any additional security mechanism is appreciated.

Another noteworthy aspect is that separating type qualification from type definition can reduce complexity of inheritance hierarchies, which enforces better architecture. The methodology of adding annotations can be superior when dealing with, and extending, existing code. Often, it is a lot easier and reasonable to add annotations to existing code than introducing new types. Tests do not break because types can be left unchanged. Existing interfaces and types can be extended with annotations rather than changing code to comply with newly introduced types, which would very likely lead to problems when integrating modules.

3.4 Encrypted data

Encryption is one of the principal concepts in security, again InjectionCop can promote this concern. By now, all discussed use cases were targeted on the respective application's code base, but since almost every application relies on third party libraries it is essential to have proper support of dependencies. Encryption is a good fit for pointing this out which is shown in Listing 3.4.

Listing 3.4: Annotations to handle encryption

```
class Outstream
{
  public void EncryptAndSend(
    [Tainted] byte[] data,
    Socket socket,
    ThirdPartyLibrary.Encryptor encryptor)
  {
    data = encryptor.Encrypt(data);
    socket.Send(data);
  }
}
```

There is a method getting some data contained in a byte array that is marked as tainted by an annotation. The data has to be encrypted by using a third party encryptor class, and sent to a given socket afterwards. The goal is to model the requirement that only encrypted data is sent over the network. There is a need for a mechanism that describes interfaces of libraries where the source code is not available, and cannot be annotated. *XML* is a good fit for defining such structures. It can easily describe an existing method and add type qualifier information needed to solve the issue. Beginning with the assembly name, for which the *XML* definition is targeted, one can define child elements for types, as well as methods and their parameters to handle overloads. The definition shown in Listing 3.5 contains *XML* that models the need of the `Send` method of the `Socket` type to have a parameter that is `Encrypted`.

Listing 3.5: Definition of a sink in *XML*

```
<Assembly name="ProjectName">
  <Type name="System.Net.Sockets.Socket">
    <Method name="Send">
      <Parameter type="System.Byte[]" fragmentType="Encrypted" />
    </Method>
  </Type>
</Assembly>
```

The counterpart is the third party library doing the encryption work and returning a byte array that is qualified to be `Encrypted` (Listing 3.6).

Listing 3.6: Definition of a source in *XML*

```
<Assembly name="ProjectName">
  <Type name="ThirdPartyLibrary.Encryptor">
    <Method name="Encrypt"
      returnFragmentType="Encrypted"
      fragmentGenerator="">
      <Parameter type="System.Byte[]"/>
    </Method>
  </Type>
</Assembly>
```

In this use case, no interface of the project's code base was qualified, and the need for integrating third party libraries in a convenient way becomes obvious. Tools lacking this kind of feature cannot handle large parts of an application's source code. Additionally, they cannot handle important use cases and requirements and reduce applicability immensely.

CHAPTER 4

InjectionCop

InjectionCop is a lightweight pluggable type checker capable of type inference for custom type qualifiers. Usage of the framework is straight forward and easy to learn. This chapter covers principle concepts of defining custom type qualifiers by defining attributes, denoting qualifier providers and receivers by adding those attributes to the business logic and integrating third party libraries by defining annotations via an *XML* interface. Finally, a short guideline for using InjectionCop by integrating into *Visual Studio* or using command-line tools is given.

4.1 High level design decisions

Features of *C/C++*, *Java* and languages of the *.NET Framework* differ significantly. This diversity of language characteristics facilitate a design of the framework that is different to the discussed custom type qualifier frameworks. The design decisions outlined in this section cope with polymorphism and functionals.

Custom type qualifier sub typing

InjectionCop features a monomorphic type system, which means that every value and variable can be interpreted to be of one and only one type. Deciding whether the type system of InjectionCop is monomorphic, or polymorphic, where every value and variable can be interpreted to be of multiple types [10], was taken carefully. *CQual* needs to be polymorphic because of the way type qualifiers are handled in *C/C++*, this issue is shown in Listing 4.1.

Listing 4.1: const problematic

```
int *foo(int *parameter) { return parameter; }
const int* cfoo(const int *parameter) { return parameter; }

void main(int argc, char **argv)
{
  int *i;
  const int *ci;

  ci = (const int*) foo((int*) ci);
  i = (int*) cfoo((const int*) i);
}
```

The first two lines of this listing define functions where the body is exactly the same. Method `foo` expects and returns a pointer to an integer, which is defined by the type `int*`. The signature of method `cfoo` is similar, it expects and returns a pointer to a constant integer, which is defined by the type *const int* *. From a software engineering point of view, code duplication should be eliminated. Applying this refactoring by removing `cfoo` introduces new problems, which is shown by the call of `foo` in the body of the main function. Calling `foo` with a const parameter requires removing the type qualifier by an explicit cast. Furthermore, assigning the return value of `foo` to a const pointer needs manual adding of the type qualifier by an explicit cast. In this case, the whole point of using a const pointer is undermined, because the dereferenced value can be modified in the body of `foo`. Doing the refactoring the other way, by removing `foo` leads to similar cast problems, which is shown by the call of `cfoo` in the body of main. Obviously, both refactorings bypass the type system. As a consequence, *C/C++* programmers either don't use const, or cast function results to non const. This problem arises because *C/C++* lacks `const` polymorphism, so that `const` introduces a new type. In other words, the type qualifier changes the signature of the function. The lattice in Figure 4.1 illustrates this taxonomy since `int` and `const int` are on the same level without any subtype relation. CQual solves this issue by

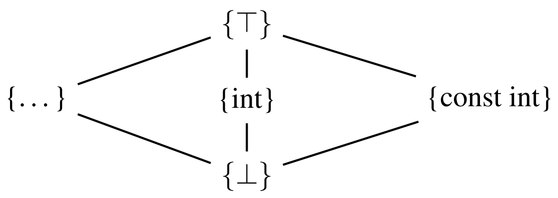


Figure 4.1: Handling of type qualifiers in *C*

adding `const` polymorphism $\tau \preceq$ const τ for all types τ to the type system of *C/C++*, the adapted lattice is shown in Figure 4.2. Adding this subtype relation enables removing function

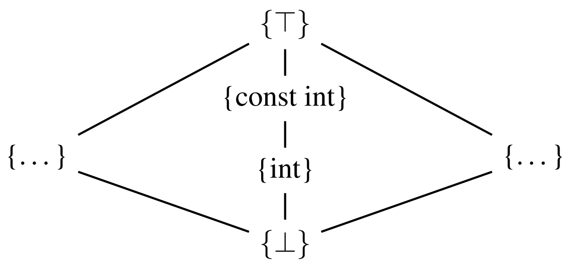


Figure 4.2: Handling of type qualifiers in CQual

foo. An effect of this type system modification is an increase of complexity, since a simple form of inheritance is introduced. In *C* this language extension is easily understandable, *C++* already supports inheritance and using these two mechanisms in parallel may introduce taxonomies which are hardly understandable.

In *C#* the situation is different because the type system does not allow const qualifiers on parameters. Instead, constants are inlined at compile time. Additionally, also *volatile*, which is the only other *C#* qualifier, cannot be defined on a parameter. This is an opportunity to create a type qualifier framework that is not related to the type hierarchy constructed by native language inheritance mechanisms. Furthermore, eliminating coupling of type qualifiers and types enables the construction of a monomorphic type qualifier engine without the need to modify the type system of the programming language. But having monomorpic type qualifiers is not the ultimate goal, since an implicit cast as shown in Listing 4.2, is common in most popular programming languages and convenient to use.

Listing 4.2: Implicit cast to *const*

```
const int i = 3;
```

Enabling this syntactic shortcut is possible by adding the artificial subtype $\perp$ to all user defined custom type qualifiers. To highlight purpose and special use of this construct the type is called "literal". Since defining a custom type qualifier for a parameter type is not mandatory, an implicit super type "Empty", representing an unset qualifier, is introduced. The empty qualifier guarantees that any custom type qualifier can be passed when no type qualifier is specified in the interface, literals enable assigning constant values to any type qualifier.

Combining all design decisions captured in this section leads to a schematic depiction of the type qualifier taxonomy (Figure 4.3). To reduce complexity, subtype relations between user defined custom type qualifiers are prevented, but for convenience literals can be assigned to any custom type qualifier.

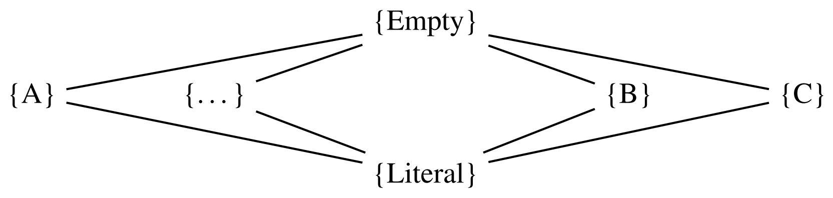


Figure 4.3: Handling of type qualifiers in InjectionCop

Functionals

Another difference between *CQual*, *JQual* and InjectionCop is handling of functionals. Since support for functionals was introduced in *Java 8*, there is no need for this feature in *JQual*. *C/C++* handles functionals by function pointers, which can be annotated like any other type. The type system can easily be bypassed with the help of type casts. *C#* features delegates, lambda functions and anonymous methods, which are basically type safe function pointers.

Since functionals are very well supported, they can be extended with custom type qualifiers very conveniently.

4.2 Verification

Hoare Calculus is the foundation of the verification engine. This section shows how analyzing a program on statement level can be automated. Core tasks of the engine are tracking custom type qualifiers of symbols, finding start and end states and resolving control structures.

Code abstraction

Since the pluggable type checker is isolated from the type system an abstract representation of code, tailored for verifying type safety of custom type qualifiers, is used for simplification. The code shown in Listing 4.3 is translated to pseudo code covering only types and instructions for type manipulation, which is shown in Listing 4.4.

Listing 4.3: Sample that is translated into pseudo code

```
[return: T]
public int Foo([T] int p)
{
  A q;
  q = new A();
  int b = 3;
  b++;

  if(1=1)
    Foo2(q);
    return p;
  else
    return 3;
}
```

Listing 4.4: Pseudo code of the sample

```
T Foo(T p)
{
  q = Empty
  b = Literal

  if(...)
    Foo2(q);
    return1 = p;
  else
    return2 = 3;
}
```

Access modifiers, as well as types, are omitted because there is no need to consider them in the verification algorithm. Return statements are converted to variables with a postfix index to handle each terminating execution path of a method separately during verification. Assignments may change the custom type qualifier of a variable. For simple cases, like the use of literals or a constructor, the new custom type qualifier can be inserted. More complex situations, such as method calls, require proper analysis to deduce the correct type qualifier. Conditional logic of branching code elements and loops can be dropped. Special cases regarding code with commands, using or manipulating custom type qualifiers in the condition, can easily be refactored by introducing an expression before the particular statement that stores the result of the condition evaluation. Also other expressions that do not affect type qualifier inferencing can be ignored. That includes all code constructs except flow control structures, assignments, returns and method calls. Flow control expressions introduce code branches that have to be resolved separately, assignments represent propagation of qualifier types. Method calls require comparing actual types of parameters with the custom type qualifier, defined in the method signature. How these three situations are handled is part of the subsequent verification chapter. T is part of the method signature and represents a custom type qualifier, the signature is used to deduce pre- and post-conditions.

Hoare Calculus

Having a pseudo code representation of a problem instance is the starting point for a static code analysis that verifies correct usage of type qualifiers. This is achieved by applying Hoare Calculus [6]. The reasoning system includes all rules that are needed for verification. The rules are explained by examples in the following sections. Deduction is based on *Hoare Triples*.

$$\{P\}C\{Q\}$$

P is a set of preconditions, Q is a set of postconditions. C is a program segment represented by a sequence of instructions of an imperative language.

Pre- and postconditions

When working with Hoare logic, one of the challenging parts is finding suitable pre-, postconditions and loop invariants. Loop invariants express the change of state that is caused by a loop, a detailed explanation is given later. To enable automation of inference, conditions have to be expressed using predicate logic. The predicates are crucial for manual proving, and even more important in an automated verification process, when adjustment of predicates by human intervention is not possible. When using InjectionCop pre- and postconditions on method level are defined by the user when the method signature is extended by custom type qualifier annotations. The conditions derived from user input are sufficient to perform verification, since loop invariants and further verification steps are deduced by the framework. Based on predicates given by the user, InjectionCop derives additional predicates on demand to perform verification. Listing 4.5 is a code snippet that is used to illustrate how pre- and postconditions are identified.

Listing 4.5: If statement with returns in every option

```
T Foo(T p, Empty q)
{
  if(...)
    return1 = p;
  else
```

```
    return2 = 3;
}
```

Parsing `Foo` results in the following *Hoare Logic* expression containing all pre- and postconditions that can be derived.

$$\frac{\ldots}{\{p = T, q = Empty, Foo = T\}if \ldots \{return1 = T, return2 = T\}}$$

The first two preconditions are determined by iterating method parameters and assigning the corresponding type defined in the signature. Those preconditions can be expected to hold, because the framework assures that the method is only called with parameter values having the same custom type qualifier as defined in the method signature. The third precondition models that any call of `Foo` returns a result of the type `T`. Again, the framework is capable of verifying that the method returns the correct custom type qualifier. Variables, that are part of a postcondition, are found by parsing the method body and seeking for assignment statements where the target is a return variable. After collecting these variables, the required return type, which is also defined in the method signature, is assigned to each of them. The resulting expression is the starting point of the analysis, the next step is breaking down statements of the method body to check for inconsistencies which indicate a type qualifier mismatch.

Deriving all possible predicates is the standard behavior of the framework, because it is not possible to determine which predicates are needed in advance. Furthermore, and higher complexity would be introduced for a minor optimization of the deduction. In contrast, some predicates of manual proofs in this work are omitted to enhance readability.

Assignments

As mentioned in previous sections, assignments are a vehicle for changing and passing qualifier types of variables. This behavior is covered by the *assignment rule* of the *Hoare Calculus*.

$$\frac{}{\{P[E/x]\}x = E\{P\}}$$

$P[E/x]$ is an expression where in predicate P all free occurrences of x are replaced by E. The following examples are some deductions that cover use cases which emerge frequently in the problem domain. When assigning a constant to a variable, the postcondition $x = 3$ can be seen easily, deriving the precondition by replacing x by the source expression results in $3 = 3$.

$$\frac{}{\{3 = 3\}x = 3\{x = 3\}}$$

It is crucial to apply this rule backwards, which means starting from the postcondition to infer the precondition. This approach may seem counter intuitive, but otherwise some bogus predicates could be deduced by a precondition that introduces a contradiction, which is shown in the following incorrect deduction.

$$\frac{}{\{x = 1\}x = 3\{3 = 1\}}$$

Also an assignment from one variable to another follows this pattern.

$$\frac{}{\{y = y\}x = y\{x = y\}}$$

Additionally, the example in this section features *rule of composition* which is used to resolve sequential application of instructions.

$$\frac{\overline{\{P\}S\{Q\}} \quad \overline{\{Q\}T\{R\}}}{\{P\}S;T\{R\}}$$

Commands S and T are split up and a mid-condition Q is introduced. Q has to be constructed in order to prove that the initial postcondition R holds. Applying this rule is straight forward, which can be seen in the following example.

$$\frac{\dfrac{\dots}{\{x=2\}x=x+1\{x=3\}} \quad \dfrac{\dots}{\{x=3\}x=2*x\{x=6\}}}{\{x=2\}x=x+1;x=2*x\{x=6\}}$$

This sample is just used for illustration. As indicated by the dots, the inference is incomplete. To complete the deduction tree, applying the *assignment rule* would be required.

Whenever logical reasoning is needed during verification, the *logical consequence rule* is applied.

$$\frac{P_1 \Rightarrow P_2, \quad \{P_2\}S\{Q_2\}, \quad Q_2 \Rightarrow Q_1}{\{P_1\}S\{Q_1\}}$$

The rule allows to strengthen the precondition or weaken the postcondition. Whenever this rule is applied in the samples of subsequent sections, it marks those parts of the proof that have to be handled by the verification engine. The following example shows a simple application of assignments that are executed sequentially. All previously mentioned rules are covered. The starting point is method `Foo`.

```
T Foo(T p)
{
  q = p;
  return1 = q;
}
```

Evaluating the problem instance by parsing the method signature implies some predicates forming pre- and postconditions, which are the starting point of the verification.

$$\frac{\dots}{\{p=T\}q=p;return1=q;\{return1=T\}}$$

Starting from this basis, the goal is to decompose the method down to instruction level by applying Hoare Calculus. The result is shown in Figure 4.4. The first step is splitting the sequence of assignments with the help of the *sequence rule*, which is indicated by the (sc) label of the induction. On the left branch of the resulting tree, the *logical consequence rule* is applied, which is indicated by the (lc) label, to prepare a proper precondition for resolving the assignment statement. The left part of the premise, marked with (1), is the entry point where the framework has to prove that the implication is correct.

$$(1) \equiv \{p=T\} \Rightarrow \{p=T, p=p\} \equiv \top$$

The implication is trivially true since simply the statement $p = p$ is added, which is obviously always true. The middle part of the premise is just an application of the *assignment rule*. The instruction models passing the type of variable p to variable q. The last part of the premise is an implication that holds because of transitivity.

$$(2) \equiv \{p = T, p = q\} \Rightarrow \{q = T\} \equiv \top$$

The second assignment follows the same pattern with other variables, which is seen in the following equations.

$$(3) \equiv \{q = T\} \Rightarrow \{q = T, q = q\} \equiv \top$$

$$(4) \equiv \{return1 = q, q = T\} \Rightarrow \{return1 = T\} \equiv \top$$

Resolving the deduction tree seen in this section required finding suitable predicates to prove that the postcondition holds. Finding these predicates followed patterns, like introducing tautologies or building transitive closure, which can be automated. Furthermore, much of the example was about tracking types of variables where well-established techniques from the field of compiler generation can help.

$$\dfrac{\dfrac{\{p=T\} \Rightarrow \overset{(1)}{\{}p=T, p=p\} \quad \{p=T,p=p\}[p/q\overset{(as)}{]}q=p\{p=T,q=p\} \quad \{p=T,q=\overset{(2)}{p}\}\Rightarrow\{q=T\}}{\{p=T\}q=p\{q=T\}}\,(lc) \qquad \alpha}{\{p=T\}q=p;return1=q;\{return1=T\}}\,(sc)$$

$$\dfrac{\{q=T\}\Rightarrow\overset{(3)}{\{}q=T,q=q\} \quad \{q=T,q=q\}return1\overset{(as)}{=}q\{return1=q,q=T\} \quad \{return1=q,q=\overset{(4)}{T}\}\Rightarrow\{return1=T\}}{\alpha:\{q=T\}return1=q\{return1=T\}}\,(lc)$$

Figure 4.4: Deduction of two sequential assignments

Method call analysis

Akin to return statements, method calls introduce further possibilities of type qualifier mismatches. Each method call needs verification to ensure that custom type qualifiers of parameter values and the qualifiers defined in the method signature match. Therefore, the framework has to handle parameter values from variables, as well as return types of nested method calls. Additionally, when a method call is the source of an assignment, the custom type qualifier of the target variable may be modified. The code snippet shown in Listing 4.6 covers the latter scenario of a method call in an assignment.

Listing 4.6: Method call in an assignment

```
S Foo (T p) {...}

S MethodCallInAssignment (T p)
{
  return1 = Foo(p);
}
```

The essential piece of code is the body of method `MethodCallInAssignment`. The body of the callee `Foo` is undefined since only the signature is required for analysis. For deduction, a mathematical function is used that models parameter checks and the return type of `Foo`.

$$Foo(x) = \begin{cases} S & x = T \\ abort & otherwise \end{cases}$$

The framework can utilize the semantics of this function simply by comparing types and static code analysis of the custom type qualifiers of the method signature for the return type and the parameters. The evaluation of method `MethodCallInAssignment` is shown in Figure 4.5. The deduction is very similar to the deductions seen in the previous chapter, the only difference is that `Foo` is used in the logical reasoning part of the *(lc)* rule.

$$(1) \equiv \{p = T\} \Rightarrow \{Foo(p) = S, Foo(p) = Foo(p)\} \equiv \top$$

The equation holds because `Foo` is called with a parameter value that matches the signature, which is ensured since the premise contains statement $p = T$.

Figure 4.6 shows a deduction of a code snippet containing nested method calls (Listing 4.7).

Listing 4.7: Nested method calls

```
S Foo (T p) {...}

S NestedMethodCalls (T p)
{
  return1 = Foo(Foo(p));
}
```

In this sample, logical reasoning is more complex because function composition has to be resolved correctly. The equation

$$(1) \equiv \{p = T\} \Rightarrow \{Foo(Foo(p)) = abort, Foo(Foo(p)) = Foo(Foo(p))\} \equiv \top$$

holds because

$$p = T \Rightarrow Foo(p) = S \Rightarrow Foo(Foo(p)) = abort$$

This leads to a return value mismatch that is recognized at the end of the deduction because the postcondition can not be resolved.

$$\{return1 = abort\} \not\Rightarrow \{return1 = S\}$$

In the previous example, a parameter mismatch is identified, because the returned value of $abort$ is propagated throughout the deduction, and a contradiction is encountered when the return value is verified. This approach does not work when the return value of the method call is discarded. In practice, both cases are simplified by aborting inferencing as soon as the parameter mismatch is found. In the manual proof a new placeholder variable and a postcondition is introduced to cover this scenario.

Listing 4.8: Discarded method call return value

```
S Foo (T p) {...}

S DiscardedMethodCallReturnValue (T p)
{
  methodReturn1 = Foo(p);
}
```

Method `DiscardedMethodCallReturnValue` (Listing 4.8) is mapped to the following incomplete deduction tree.

$$\frac{\dots}{\{p = T\}methodReturn1 = Foo(p)\{methodReturn1 \neq abort\}}$$

By adding a placeholder variable "methodReturn1" and the postcondition $methodReturn1 \neq abort$, the error can not slip through the verification engine, and the mismatch is identified correctly.

Another noteworthy aspect of methods are overloads. When looking at the samples covered in this chapter, it is easy to create method overloads that have the same type qualifier signature. In the implementation this is not a problem, because the type system assures that the correct signature is used and referenced in the object model of the method.

$$\frac{\{p = T\} \Rightarrow \{Foo(p) \overset{(1)}{=} S, Foo(p) = Foo(p)\} \quad \beta \quad \{return1 = Foo(p), Foo(p) = S\} \Rightarrow \{return1 = S\}}{\{p = T\} return1 = Foo(p) \{return1 = S\}} \ (lc)$$

$$\beta : \{Foo(p) = S, Foo(p) = Foo(p)\}[Foo(p)/return1] \overset{(as)}{return1} = Foo(p) \{return1 = Foo(p), Foo2(p) = S\}$$

Figure 4.5: Resolution of a method call that is the source of an assignment

$$\frac{\{p = T\} \Rightarrow \{Foo(Foo(p)) = abort, \overset{(1)}{F}oo(Foo(p)) = Foo(Foo(p))\} =: \alpha \quad \beta \quad \gamma}{\{p = T\} return1 = Foo(Foo(p)) \{return1 = S\}} \ (lc)$$

$$\beta : \alpha[Foo(Foo(p))/return1] \overset{(as)}{return1} = Foo(Foo(p))\gamma$$

$$\gamma := \{return1 = Foo(Foo(p)), Foo(Foo(p)) = abort\} \Rightarrow \{return1 = abort\}$$

Figure 4.6: Resolution of nested method calls

Branching statements

Branching statements are among the core concepts of any language. Only `if` statements are covered in this chapter, since `switch` statements or conditional operator can be mapped to equivalent code using only `if` statements. When looking at Intermediate Language, all mentioned manifestations are language features to enhance convenience for the programmer and are ultimately converted to a branch instruction. This instruction checks an intermediate variable that holds the result of the expression of the condition and jumps according to the value. This also implies that instructions of the condition, that could affect inferencing of custom type qualifiers, are not distinguishable from instructions that are executed right before the `if` statement and handled by the framework automatically. Therefore, the conditional statement is of the form $x == true$ which is a statement that is irrelevant for inference of custom type qualifiers.

Certainly, the *conditional rule* contains the condition. The rule resolves an `if` statement by setting up a branch for each code path.

$$\frac{\{B \wedge P\}S\{Q\}, \quad \{\neg B \wedge P\}T\{Q\}}{\{P\}if\ B\ then\ S\ else\ T\ endif\{Q\}}$$

For the *consequent*, which is the path that is executed when the *conditional* statement holds, the precondition is the union of the precondition defined in the *conclusion* and the conditional of the `if` statement. For the *alternative*, which is the path that is executed when the *conditional* statement does not hold, the precondition is the union of the precondition defined in the conclusion and the negated conditional expression of the `if` statement. Since the postcondition has to hold for the *consequent* and the *alternative*, both branches acquire the postcondition from the *consequence*.

Listing 4.9: Valid branching statement

```
public T Foo(T p, S q)
{
  if(...)
    x = p;
  else
    x = q;
  return1 = x;
}
```

Listing 4.9 contains a sample where the *consequent* of an `if` statement is unproblematic, while the *alternative* raises an error. The *conditional rule* resolves this code snippet correctly (Figure 4.7). Evaluation of assignments is handled as described in the preceding chapters. A noteworthy difference is given by an application of the `if` rule, which introduces two branches in the inference tree that need to fulfill the same postcondition. The precondition of the *consequent* contains a placeholder B representing the conditional. As already mentioned, the conditional is irrelevant for inferencing custom type qualifiers, so the predicate is removed by the following application of the *logical consequence* rule. Accordingly, the precondition of the alternative contains $\not B$ which is also removed.

Another aspect of the deduction is that in order to determine the postcondition of the if statement, the branch containing the assignment of the return value is evaluated first. When the verification is executed manually, this approach seems to be natural, but this is a step that is hard to automate. When the analysis of the method follows a top down approach, the impact on the performance needs to be considered since the remainder of the if statement is evaluated multiple times. Nested branching statements would boost this issue even more. A more detailed treatment of this aspect is given in Chapter 6, complexity analysis.

$$\dfrac{\dfrac{\alpha}{\{p=T,q=S\}if\ B\ then\ x=p\ else\ x=q\ endif;\{x=T\}} \quad \dfrac{\beta}{\{x=T\}return1=x\{return1=T\}}}{\{p=T,q=S\}if\ B\ then\ x=p\ else\ x=q\ endif;\ return1=x\{return1=T\}}\ (sc)$$

$$\alpha: \dfrac{\dfrac{\gamma}{\{B,p=T,q=S\}x=p\{x=T\}} \quad \dfrac{\delta}{\{p=T,q=S\}x=q\{x=T\}}}{\{p=T,q=S\}if\ B\ then\ x=p\ else\ x=q\ endif;\{x=T\}}\ (if)$$

$$\gamma: \dfrac{\{B,p=T,q=S\}\Rightarrow\{p=T,p=p\} \quad \{p=T,p=p\}[p/x]x=p\{p=T,x=p\} \quad \{p=T,x=p\}\Rightarrow\{x=T\}}{\{B,p=T,q=S\}x=p\{x=T\}}\ (lc)$$

$$\delta: \dfrac{\{B,p=T,q=S\}\Rightarrow\{q=S,q=q\} \quad \{q=S,q=q\}[q/x]x=q\{q=S,x=q\} \quad \{q=S,x=q\}\not\Rightarrow\{q=T\}}{\{B,p=T,q=S\}x=q\{x=T\}}\ (lc)$$

$$\beta: \dfrac{\{x=T\}\Rightarrow\{x=T,x=x\} \quad \{x=T,x=x\}[x/return1]return1=x\{x=T,return1=x\} \quad \{x=T,return1=x\}\Rightarrow\{return1=T\}}{\{x=T\}return1=x\{return1=T\}}\ (lc)$$

Figure 4.7: Resolution of a branching statement

Loops

Loops are the most complex expressions the framework has to deal with. Complexity is mainly introduced by repetitions of code blocks and optional execution of the body. In this subsection, three cases are examined in detail, to illustrate core aspects of loop handling. Although complexity is high, already known approaches can also be applied for loops.

Hoare calculus provides a rule that handles each of the cases properly.

$$\frac{\{I \wedge B\}S\{I\}}{\{I\}while\ B\ do\ S\ done\{\neg B \wedge I\}}$$

I represents an invariant which is a state that holds before, during, and after the loop. The invariant is the core aspect of the rule, because its predicates express the semantics of the loop. There is no generic approach to obtain the invariant, since the invariant is problem-specific and hard to identify in most cases. This chapter illustrates how the invariant is defined in an automated way, tailored for the problem domain. Just as *if rule*, this rule includes branch condition B. In general, B is often used in combination with the invariant for deriving the final result of the loop. In the problem domain of custom type qualifiers, the condition is not used, since it cannot change a qualifier type.

All samples in this chapter use helper method `Foo`, which requires a parameter of type T and returns type S.

```
S Foo(T p) {...}
```

The behavior of the method is defined by the following function.

$$Foo(x) = \begin{cases} S & x = T \\ abort & otherwise \end{cases}$$

The first sample illustrates a case where the framework can assure type qualifier correctness (Listing 4.10).

Listing 4.10: Correct return type

```
S CorrectReturnType(T p, S q)
{
  x = q

  while(...)
    x = Foo(p);

  return1 = x;
}
```

The matching return type S is assigned to the return variable x. After the assignment, the body of the loop is executed either multiple times or not at all. Both cases leave x in a correct state. If the loop body is not executed, x keeps the value it had before. Otherwise, the type of x is set to the return type of `Foo` which is also a correct return type of method `CorrectReturnType`. The inference tree of this method is shown in Figure 4.8. In this sample, the invariant $I := \{p = T, x = S\}$ expresses that the loop cannot alter the value of x, and therefore the values of x before and after the loop are equivalent. The essential part of the inference tree starts at β, where the *while rule* is applied and invariant I is introduced. Furthermore, in the subsequent application of the `lc rule`, the part $x = S$ of the invariant is dropped at derivation γ. Afterwards, the assignment statement restores the expression and enables deriving the invariant for the postcondition of the `while rule`. This behavior simulates resetting the value of x by a method call, which is the body of the loop in the pseudo code.

$$\dfrac{\dfrac{\{p=T,q=S,x=Empty\} \Rightarrow \{p=T,q=S,q=q\} \quad \overset{(as)}{\{p=T,q=S,q=q\}[q/x]x=q\{p=T,q=S,x=q\}} \quad \alpha}{\{p=T,q=S,x=Empty\}x=q\{p=T,x=S\}}\,(lc) \quad \beta}{\{p=T,q=S,x=Empty\}x=q,\ while\ \ldots;\ return1=x;\{return1=S\}}\,(sc)$$

$$\alpha : \{p=T,q=S,x=q\} \Rightarrow \{p=T,x=S\}$$

$$\beta : \dfrac{\dfrac{\dfrac{\gamma \quad \overset{(as)}{\{p=T,Foo(p)=S,Foo(p)=Foo(p)\}[Foo(p)/x]x=Foo2(p)\{p=T,Foo(p)=S,x=Foo(p)\}} \quad \delta}{\{B,p=T,x=S\}x=Foo2(p)\{p=T,x=S\}}\,(lc)}{\{I\} := \{p=T,x=S\}while\ \ldots\ do\ x=Foo(p)\ done\{\neg B,p=T,x=S\}}\,(while) \quad \epsilon}{\{p=T,x=S\}while\ \ldots\ do\ x=Foo(p)\ done;\ return1=x\{return1=S\}}\,(sc)$$

$$\gamma : \{p=T,x=S\} \Rightarrow \{p=T,Foo(p)=S,Foo(p)=Foo(p)\}$$

$$\delta : \{p=T,Foo(p)=S,x=Foo(p)\} \Rightarrow \{p=T,x=S\}$$

$$\epsilon : \dfrac{\{p=T,x=S\} \Rightarrow \{x=S,x=x\} \quad \overset{(as)}{\{x=S,x=x\}[x/return1]return1=x\{x=S,return1=x\}} \quad \psi}{\{p=T,x=S\}return1=x\{return1=S\}}$$

$$\psi : \{x=S,return1=x\} \Rightarrow \{return1=S\}$$

Figure 4.8: Resolution of a loop

The second sample illustrates a scenario where the framework cannot guarantee that the return type of the method is the correct return type S (Listing 4.11). If the loop body is not executed, x is not set and does not match the required return type.

Listing 4.11: Return type may not match

```
S LoopMayChangeReturnType(T p)
{
  while(...)
    x = Foo(p);

  return1 = x;
}
```

At the top of Figure 4.9, it can be seen that the inference tree basically consists of three branches. These branches cope with the invariant, the actual loop and comparing the state after the loop with the target state. As mentioned before, the invariant has to express the state before, during and after the loop. In this case, the predicates model that the type qualifier of x may change. Implication α builds the invariant by weakening the precondition by adding $x = \mathit{Empty} \not\Leftrightarrow x = S$ to express the only two possible custom type qualifiers for x. The predicate $x = \mathit{Empty}$ holds, if the body is not executed, the second part $x = S$ holds if the loop body is executed at least once. This is crucial for branch β where $x = S$ is needed in the state after the assignment, to derive the invariant which is part of the postcondition of the `while rule`.

To build the invariant in an automated way, the framework would have to compare changes of variables by memorizing the symbol table before the loop and track all possible assignments in any path inside the loop. It would be wrong behavior if the framework reported a problem right after identifying a custom type qualifier mismatch since the variable involved in the mismatch may not be used again. Therefore, variables in a mismatch state must be marked as an ambiguous type, and the problem is raised when the next occurrence is encountered. This behavior is illustrated in branch γ, where the desired custom type qualifier of the return value cannot be derived, because the ambiguity of variable x is propagated to A_2 which is the state after the assignment. Alternatively, the code block could be inferred multiple times with all possible type qualifier constellations. The result would be the same since one inference tree would lead to a type qualifier mismatch.

$$\dfrac{\alpha \quad \dfrac{\beta}{\{I\}while\ \dots\ dox = Foo(p)\ done\{\neg B \wedge I\}} \quad \dfrac{\gamma}{\{\neg B \wedge I\}return1 = x\{return1 = S\}}}{\{p = T, x = Empty\}while\ \dots\ do\ x = Foo(p)\ done;\ return1 = x\{return1 = S\}}\ (sc, lc)$$

$$\begin{aligned} \alpha : \{p = T, x = Empty\} &\Rightarrow \{p = T, x = Empty, x = Empty \not\Leftrightarrow x = S\} && (4.1)\\ &\Rightarrow \{p = T, x = Empty \not\Leftrightarrow x = S\} =: \{I\} && (4.2) \end{aligned}$$

$$\beta : \dfrac{\dfrac{\{I\} \Rightarrow \{p = T, x = Empty \not\Leftrightarrow x = S, Foo(p) = Foo(p), Foo(p) = S\} =: \{A_1[Foo2(p)/x]\} \quad \{A_1[Foo2(p)/x]\}\overset{(as)}{x} = Foo2(p)\{A_1\} \quad \delta}{\{I \wedge B\}x = Foo(p); \{I\}}\ (lc)}{\{I\}while\ \dots\ do\ x = Foo(p)\ done\{\neg B \wedge I\}}\ (while)$$

$$\begin{aligned} \delta : \{A_1\} = \{p = T, x = Empty \not\Leftrightarrow x = S, x = Foo(p), Foo(p) = S\} &\Rightarrow \{p = T, x = Empty \not\Leftrightarrow x = S, x = S\} && (4.3)\\ &\Rightarrow \{p = T, x = Empty \not\Leftrightarrow x = S\} = \{I\} && (4.4) \end{aligned}$$

$$\gamma : \dfrac{\{\neg B \wedge I\} \Rightarrow \{x = Empty \not\Leftrightarrow x = S, x = x\} =: \{A_2[x/return1]\} \quad \{A_2[x/return1]\}\overset{(as)}{return1} = x\{A_2\} \quad \{A_2\} \not\Rightarrow \{return1 = S\}}{\{\neg B \wedge I\}return1 = x\{return1 = S\}}\ (lc)$$

$$\{A_2\} = \{x = Empty \not\Leftrightarrow x = S, return1 = x\} \not\Rightarrow \{return1 = S\}$$

Figure 4.9: Resolution of a loop that causes a mismatch when the body is executed

Loops also introduce some tricky situations that are hard to grasp. In the sample shown in Listing 4.12, a type qualifier mismatch would occur after the second iteration of the loop.

Listing 4.12: Mismatch after second iteration

```
void ParameterMismatch(T p, S q)
{
  while(...)
  {
    Foo(p);
    p = q;
  }
}
```

Since it is not possible to evaluate the number of iterations, the framework has to raise a type qualifier mismatch. This is a good showcase of the limits of static code analysis. Also, commercial and widespread tools like *Resharper* [63] do not perform a detailed condition analysis. Again, the invariant is formed by adding an antivalence $p = T \not\Leftrightarrow p = S$ to the state before the loop to express the ambiguity of p. Furthermore, the invariant lacks an assignment for placeholder variable mr. Variable mr is an auxiliary variable that is used to detect parameter mismatches, which is necessary since the return value is discarded in the sample code snippet. The absence of mr in the invariant expresses that the custom type qualifier value for mr cannot be derived because the type qualifier of parameter p is ambiguous. This can be seen in implication γ, where possible assignments are derived, but discarded afterwards to satisfy the invariant. Ambiguity of p and mr are handled differently because the impact is different. The framework would have to mark p as ambiguous and keep the state to be able to detect potential type qualifier mismatches when the variable is used as the value of a qualified parameter. On the other side tracking mr is not necessary, because it is definitely not used, and a mismatch can be raised as soon as it is detected that method `Foo` is called with an ambiguous parameter.

$$\dfrac{\{p=T,q=S\} \Rightarrow \{p=T \not\Leftrightarrow p=S, q=S\} =: I \quad \dfrac{\dfrac{\dfrac{\alpha}{\{I \wedge B\}mr = Foo(p)\{I\}} \quad \dfrac{\beta}{\{I\}p=q;\{I\}}}{\{I \wedge B\}mr = Foo(p);\ p=q;\{I\}}\,(sc)}{\{I\}while\ \ldots\ do\ mr = Foo(p);\ p=q\ done\{\neg B \wedge I\}}\,(while)}{\{p=T,q=S\}while\ \ldots\ do\ mr = Foo(p);\ p=q\ done\{mr \neq abort\}}\,(lc)$$

$$\alpha : \dfrac{\{I \wedge B\} \Rightarrow \{p=T \not\Leftrightarrow p=S, q=S, Foo(p)=Foo(p)\} =: \{A_1[Foo(p)/mr]\} \quad \overset{(as)}{\{A_1[Foo(p)/mr]\}mr = Foo(p)\{A_1\}} \quad \gamma}{\{I \wedge B\}mr = Foo(p)\{I\}}$$

$$\gamma : \{A_1\} = \{p=T \not\Leftrightarrow p=S, q=S, mr=Foo(p)\} \Rightarrow \{p=T \not\Leftrightarrow p=S, q=S, mr=Foo(T) \not\Leftrightarrow mr=Foo(S)\} \tag{4.5}$$

$$\Leftrightarrow \{p=T \not\Leftrightarrow p=S, q=S, mr=S \not\Leftrightarrow mr=abort\} \tag{4.6}$$

$$\Rightarrow \{p=T \not\Leftrightarrow p=S, q=S\} = \{I\} \tag{4.7}$$

$$\beta : \dfrac{\{I\} \Rightarrow \{p=T \not\Leftrightarrow p=S, q=S, q=q\} =: \{A_2[q/p]\} \quad \{A_2[q/p]\}p=q\{A_2\} \quad \delta}{\{I\}p=q;\{I\}}\,(lc)$$

$$\delta : \{A_2\} = \{p=T \not\Leftrightarrow p=S, q=S, q=p\} \Rightarrow \{p=T \not\Leftrightarrow p=S, q=S\} = \{I\}$$

$$\{I\} = \{p=T \not\Leftrightarrow p=S, q=S\} \not\Rightarrow \{mr \neq abort\}$$

Figure 4.10: Resolution of a loop that causes a mismatch in the second iteration

Further aspects of verification

Previous chapters covered handling of control structures that are supported by most programming languages. Since *C#* is very feature rich, some supported language features deserve a closer look. Since the core architecture is not affected by these features, the topics in the section are covered briefly.

Aliasing

Aliasing occurs when two variables refer to the same data location. If this is the case for x and y the following deduction is wrong because a modification of x modifies y too.

$$\{y = 0\}x := 1\{y = 0\}$$

In *C#*, variables in code can either have value types, such as built in types or structs, or reference types, such as objects (Listing 4.13).

Listing 4.13: Possible types of variables

```
void Foo()
{
  int i = 3;
  var x = new Object();
}
```

Both categories cannot introduce aliasing since variables always refer to distinct memory locations. This does not hold for parameters, which is shown in Listing 4.14.

Listing 4.14: Evaluation strategies

```
void Foo(int a, int b, object c, object d, ref int e, ref int f)
{
}
```

The first four parameters act like variables. Parameters a and b have a value type and are evaluated by a call-by-value strategy. Parameters c and d have a reference type and refer to different memory locations that are allocated when the method is called, they follow a call-by-sharing evaluation strategy. The last two parameters are evaluated in a call-by-reference evaluation strategy. When the method is called, no memory is allocated for parameters e and f. Instead, a pointer to the memory location of the parameter values at the callee is used. Aliasing occurs if the same variable is passed as parameter e and f, which is a case that cannot be handled. Therefore, ref parameters are not supported by the framework. Aliasing can also be introduced by using pointers. Code using pointers is called unsafe code in *C#* terminology and also not supported by the framework. In practice, these restrictions don't impact usability since they relate to features that are rarely used.

Type qualifier vectors

Continuing analysis of parameters reveals scenarios where a single type qualifier is not expressive enough (Listing 4.15).

Listing 4.15: Annotating delegates

```
delegate [S] QualifiedFunction ([T] object a)

void Foo(QualifiedFunction f)
{
}
```

In this case, the parameter f is a *functional* that takes an object as parameter and returns an object. According to *C#* notation this is handled by introducing a custom type qualifier vector containing the types of the signature. The first element of the vector represents the return value and is followed by the parameter qualifiers. Hence, the precondition of the deduction tree would contain $f = (S, T)$. Introducing vectors has little effect on the algorithms and approaches covered so far, because the framework only uses equality of type qualifiers which can be easily extended to support vectors.

Out parameters

Out parameters can be defined in *C#* to return multiple values without being forced to create a container object (Listing 4.16.

Listing 4.16: Out parameters

```
void Foo(out int a, out int b)
{
}
```

There is no special treatment needed for type qualifier verification, because out parameters can be handled like ordinary return values. Every occurrence of a return implies additional checks for the return values.

Fields

A custom type qualifier can also be defined on a field to ensure that only values of a specific qualifier are assigned. For deduction, a predicate expressing the constant value of the field can be added to the precondition. Assignments of fields need to be handled like method calls, since the semantics is the same as calling a method without parameters. Therefore, a mismatch can be detected immediately and no further analysis is required.

Implications on implementation

It was shown that a tool handling *custom type qualifiers* can be implemented with methods of formal verification and compiler construction. *Hoare Calculus* can be used analyze an assembly and handle all control sturctures of *C#* to follow all execution paths. Also inconsistencies can be found, which are revealed type qualifier mismatches. The verification can be automated by means of *static code analysis* and the help of a symbol table to track *qualifier types* of variables. The correct call of methods can be checked by analyzing method signatures and comparing specific *custom type qualifiers* of parameters with the actual type of the variable that is passed and stored in the symbol table.

4.3 Generating custom annotations

The first step to utilize custom type inference is defining custom type qualifiers. To achieve this, InjectionCop provides various possibilities. The fastest way is using the pre defined `FragmentAttribute` contained in "InjectionCopAnnotations.dll" where the custom type can be specified by a string parameter.

```
public void Foo ([Fragment("CustomType")] object qualifiedParameter)
```

This approach can be useful when the framework is used sparsely, or in a strongly restricted scope, like in a test class. On the other hand, this approach is not very robust, since typos can unintentionally introduce new type qualifiers, and unfortunate refactoring can break type inference. Additionally, literals are not part of the type system and therefore not expressive from a programming language theoretic point of view. The following improvement addresses this issue by moving the string literal to a class capable of managing custom type qualifier literals.

```
public void Foo (
  [Fragment(TypeQualifiers.CustomType)] object qualifiedParameter)
```

This code snippet comes along with an obvious drawback: verbosity. Many characters are waste because the identifiers `Fragment` and `TypeQualifiers` are non descriptive. The more elegant way is inheriting from the built in attributes to implement attributes that are recognized by the type system. Those attributes are tailored for a specific context and verbosity can be minimized without losing expressiveness.

```
public void Foo ([CustomType] object qualifiedParameter)
```

Defining such attributes is quite simple and depicted in Listing 4.17. User-defined attributes have to be in namespace `InjectionCop.Attributes` and an extension of the attribute `InjectionCop.FragmentAttribute` has to be defined. Additionally, an overload of the default constructor has to be implemented, that passes a string representation of the custom type qualifier to the base class. This is the preferred approach for defining custom type qualifiers and the best fit in most scenarios.

Listing 4.17: Custom type qualifier definition by inheritance

```
namespace InjectionCop.Attributes
{
  [AttributeUsage(AttributeTargets.Parameter
                  | AttributeTargets.ReturnValue
                  | AttributeTargets.Field
                  | AttributeTargets.Property
                  | AttributeTargets.Constructor)]
  public class CustomTypeAttribute : FragmentAttribute
  {
    public CustomTypeAttribute() : base("CustomType") { }
  }
}
```

4.4 Annotation usage

By using proper annotations, custom type qualifiers can be defined. Those annotations are recognized by the inference engine, which performs static code analysis to reveal possible violations. This section covers all possible positions in the source code where custom type qualifier annotations can be added, as well as some framework design decisions concerning inheritance of custom type qualifiers and handling of literals by InjectionCop.

Annotating methods

As illustrated in the last section, custom type qualifiers can be added as parameter attributes to facilitate expressiveness of methods. Listing 4.18 shows all possible scenarios where custom type attributes can be placed. The parameter of method `Foo` contains an attribute declaring the constraint that on every call of `Foo` the first parameter must be an object of qualifier type `CustomType`. This constraint is violated in method `UnsafeFooCall`, since it calls `EmptyTypeGenerator`, which is not a declared source of `CustomType` qualifier. The return type of `EmptyTypeGenerator` is of the default type, the empty qualifier type, because it does not have any qualifier annotations. Another feature of InjectionCopis annotating return values, which is used in `UnsafeFooReturn`. The concept is similar to the support of parameter qualification, and the implied constraint is violated because the return type of `EmptyTypeGenerator` does not match.

At this point, the question of how to generate custom type qualifier sources to be able to perform safe method calls and safe return statements according to the sample annotations arises. This is where the attribute `FragmentGenerator` comes into play, which is shown in method `CustomTypeGenerator`. The attribute represents the developers commitment that the corresponding method always returns a matching custom type qualifier, and therefore, the inference engine does not need to check the return value. It is essential to use this annotation carefully, since it can break validity of inference. Generator methods should be of high quality, well tested, and reviewed to maximize confidence in the result of the static code analysis.

After setting up a matching generator, giving samples of safe method calls and safe returns is an easy task. Methods `SafeFooCall` and `SafeFooreturn` utilize `CustomGenerator` to provide samples of safe custom type qualifier usage.

Listing 4.18: Example covering method annotations entirely

```
class AnnotatingMethods
{
  public void Foo([Fragment("CustomType")] object parameter) { }

  public void UnsafeFooCall()
  {
    // unsafe: calling Foo with empty type qualifier
    Foo(EmptyTypeGenerator());
  }

  [return: CustomType]
  public object UnsafeFooReturn()
  {
    // unsafe: returning empty type qualifier
    return EmptyTypeGenerator();
  }

  public object EmptyTypeGenerator()
  {
    return new object();
  }

  [FragmentGenerator]
  [return: CustomType]
  public object CustomTypeGenerator()
  {
    return new object();
  }

  public void SafeFooCall()
  {
    Foo(CustomTypeGenerator());
  }

  [return: CustomType]
  public object SafeFooReturn()
  {
    return CustomTypeGenerator();
  }
}
```

Annotating fields

When methods are annotated, there is always a clear and obvious distinction between type qualifier providers and receivers that is given by the `FragmentGenerator` attribute. Fields have a different semantic meaning since an analogical differentiation would introduce irrational behavior. Having a field defined as source-only would enable the developer to assign any type qualifier resulting in an obscure conversion to the field's type. This is an InjectionCop anti pattern, because it breaks the principle of highlighting and declaring code capable of generating qualified data clearly. Turning the scenario around, a field defined as receiving-only would lead to a field where only objects of the specific qualified type can be assigned but inferencing would discard this information when the field is used. Therefore, forcing the developer to assign a certain type automatically turns the corresponding field into a source, too.

Listing 4.19 illustrates the previously described behavior. Field `customTypeField` is qualified as `CustomType`. Extending behavior of type qualifiers consistently according to handling methods, member `Unsafe` assigns an object of the empty type which raises a warning. A safe assignment and a safe call with a qualified field parameter is demonstrated in method `Safe`. Safeness of the assignment is assured by properly annotated `CustomTypeGenerator`, safeness of the call to `Foo` is ascertained by the attribute of `customField`.

Listing 4.19: Safe and unsafe usage of fields

```
class AnnotatingFields
```

```
{
  [CustomType] private object customTypeField;

  public void Unsafe()
  {
    // unsafe: assigning empty type qualifier
    customTypeField = new object();
  }

  public void Safe()
  {
    customTypeField = CustomTypeGenerator();
    Foo(customTypeField);
  }

  public void Foo([CustomType] object parameter) { }

  [FragmentGenerator]
  [return: CustomType]
  public object CustomTypeGenerator()
  {
    return new object();
  }
}
```

Annotating properties

In cases where properties are used like fields, which is illustrated in Listing 4.20 by the property `CustomTypeProperty`, the developer would expect InjectionCop to behave as if the property was a field. Of course, this is the way InjectionCop interprets those properties. Taking these considerations a step further, additional code in the property body could break validity. The getter of `ComplexCustomTypeProperty` causes a warning, since an object holding the empty type qualifier is returned. Also assigning a mismatching qualifier type raises a warning which is shown in method `Foo`.

From a developers' point of view, this behavior may be too restrictive. Since a `getter` could include instructions causing the property's type qualifier to be obsolete, and also the `setter` could do some sanitizing work without the need of passing a value of a specific qualifier type, it would be desirable to annotate getters and setters according to their semantics. Currently, InjectionCop does not support this feature but implementation is planned, further details can be found in the Chapter *Future work* 7.

Listing 4.20: Annotating a property

```
class AnnotatingProperties
{
  [CustomType]
  public object CustomTypeProperty { get; set; }

  [CustomType]
  public object ComplexCustomTypeProperty
  {
    get
    {
      // unsafe: returning empty type qualifier
      return new object();
    }
    set { }
  }

  public void Foo()
  {
    // unsafe: assigning empty type qualifier
    ComplexCustomTypeProperty = new object();
  }
}
```

Annotating constructors

For convenience, InjectionCop provides the feature of annotating constructors to assign a custom type qualifier to a generated object. The advantage of this approach is that initialization and qualification of an object is possible without encapsulation in a class that is in charge of the initialization of the object. This prevents extensive utilization of creational patterns, for example *factory pattern* [60] where initialization is done by a factory class, or *builder pattern* [21] where a class is used to collect meta data before the object is created. Annotating a constructor with a type qualifier attribute, as shown in Listing 4.21, is semantically equivalent to qualifying a method's return value and adding `FragmentGenerator` attribute. This syntax inconsistency is a disadvantage that is introduced by the language definition of *C#*. Annotating the return value of a constructor is not possible, because from a language theoretical point of view, there is no return value. Adding `FragmentGenerator` attribute to the constructor is syntactic sugar because there is no additional semantic meaning present.

Listing 4.21: Annotating a constructor

```
class AnnotatingConstructors
{
  [CustomType]
  public AnnotatingConstructors
    ([CustomType] object parameter) { }
}
```

Annotating interfaces

There is no difference between annotating an interface and a class, as shown in Listing 4.22, which contains the interface that is used for illustrating behavior of InjectionCop when interfaces are inherited.

Listing 4.22: Annotated interface

```
interface AnnotatedInterface
{
  [return: CustomType]
  object Foo([CustomType] object parameter);

  [FragmentGenerator]
  [return: CustomType]
  object CustomTypeGenerator();
}
```

A class implementing the interface, as depicted in Listing 4.23, automatically inherits the interface's annotations. This design decision was taken because type qualifiers are heavily related to interfaces. Intentions for defining custom type qualifiers are very similar to intentions for defining an interface. The difference is that the interface or contract defined by type qualifiers is incorporated into an existing type and no distinct artifact is created. From this point of view, a type that does not adhere to declared custom type qualifiers of an interface does not fully implement the interface, and a warning must be raised by InjectionCop. For better readability and maintainability, matching attributes can be added to classes, which are treated as syntactic sugar and ignored by the inference engine. Method `UnsafeFooCalls` in Listing 4.23 demonstrates this behavior by calling `Foo` with an unsafe parameter, once with a caller of the interface's type and once with a caller of the implementation's type.

Listing 4.23: Implementation of `AnnotatedInterface`

```
class Implementation : AnnotatedInterface
{
  public object Foo(object parameter)
  {
    return new object();
  }

  public object CustomTypeGenerator()
  {
    return new object();
  }

  public void UnsafeFooCalls()
  {
    Implementation implementation = new Implementation();
    implementation.Foo(new object()); // raises warning
    AnnotatedInterface interfaceRef = implementation;
    interfaceRef.Foo(new object());   // raises warning
  }
}
```

Since it is syntactically correct to place attributes on the implementation of an interface, InjectionCop provides a mechanism that detects mismatches of custom type qualifiers. Listing 4.24 contains a sample, where method `Foo` is annotated in contradiction to the implemented interface. In this case, a warning is raised notifying the developer that the attribute of the implementation's method is ignored and the interface's annotations are used for inference.

Listing 4.24: Mismatching implementation of `AnnotatedInterface`

```
class QualifierMismatch : AnnotatedInterface
```

```
{
  public object Foo([Fragment("MismatchingType")] object parameter)
  {
    return new object();
  }

  public object CustomTypeGenerator()
  {
    return new object();
  }
}
```

Inheritance

The behavior of InjectionCop when dealing with inheritance is closely related to behavior when implementing interfaces. However, there are differences between handling dynamic and static binding. Listing 4.25 shows a base class, containing methods that are used for explaining treatment by InjectionCop in context of inheritance.

Listing 4.25: Annotated base class

```
class AnnotatedBaseClass
{
  [return: CustomType]
  public object StaticBinding([CustomType] object parameter)
  {
    return DynamicBinding();
  }

  [return: CustomType]
  public virtual object DynamicBinding()
  {
    return CustomTypeGenerator();
  }

  [FragmentGenerator]
  [return: CustomType]
  public virtual object CustomTypeGenerator()
  {
    return new object();
  }
}
```

A class extending `AnnotatedBaseClass` is depicted in Listing 4.26. `StaticBinding` is a method that is statically bound and defines other *custom type qualifiers* than the base class. This is a completely legitimate use case, because the decision of using either the method defined in the base class or the method defined in the extension of the base class is explicitly stated in the source code. When using static binding, the developer chooses to explicitly define which implementation of the method to use. This behavior is illustrated in method `SomeCalls`.

Dynamic binding consequently pursues design decisions for handling interfaces. Changing custom type qualifiers of dynamically bound methods would encourage bad design. Dynamic binding is used in situations where a method with a well-defined prototype, that is constant across the class hierarchy, should execute a method body dependent on the object instance. Since qualifiers are part of the method prototype, a change in the prototype contradicts the intended usage of a virtual method. Following this principle, definition of the overridden method `DynamicBinding` in class `Inherited`, which introduces an annotation mismatch, raises a warning.

Listing 4.26: Extension of `AnnotatedBaseClass`

```
class Inherited : AnnotatedBaseClass
{
  public new object StaticBinding([Fragment("OtherType")]object parameter)
  {
    return new object();   // no warning
  }

  // warning: type qualifier mismatch
  [return: Fragment("OtherType")]
  public override object DynamicBinding()
  {
    return new object();
  }

  void SomeCalls()
  {
    // warning
    StaticBinding(CustomTypeGenerator());
    // no warning
    base.StaticBinding(CustomTypeGenerator());
  }
}
```

Annotating functionals

C# provides various means for defining functionals, namely delegates, anonymous methods and lambda expressions. Since functionals are first class citizens, it is essential to provide features for adding custom type qualifiers to type definitions of functionals. Listing 4.27 demonstrates annotating a delegate, which is similar to annotating a method, as well as usage of functionals.

The first statement of method `Calls` is default behavior, which was already explained in previous sections. A warning is raised because the parameter object is not qualified properly. The method body becomes more interesting when the same method is assigned to a variable of a delegate type. In this case, the method's qualifiers are obsolete and matching is based on the delegate's type qualifiers. This is demonstrated by the following calls which raise warnings related to the delegate type. This behavior requires explanation, since, at first glance this design decision is in contradiction to previous decisions where custom type qualifier mismatches trigger a warning. Comparing this use case with handling of interfaces, the difference is that a class explicitly declares to implement an interface, while there is no possibility to define that a method `implements` a delegate type. This is not a weakness of *C#*, delegates are simply not intended for this kind of usage. Additionally, this would be a pointless language feature since interfaces are the obvious approach to handle this. Type qualifier mismatches are ruled out when an anonymous method or lambda expressions is assigned to a delegate variable, because *C#* forbids adding attributes to them.

The section about anonymous method behavior of method `Call` demonstrates the ability of InjectionCop to detect type qualifier mismatches of delegate types on return values. A warning is raised, because the delegate type declares to return an object qualified as `CustomType`, but the assigned anonymous method returns the empty type. The following lambda expression satisfies this constraint since it is an implementation of the identity function.

Listing 4.27: Annotating functionals

```
class AnnotatingFunctionals
{
  [return: CustomType]
  public delegate object AnnotatedDelegate([CustomType] object parameter);

  public object Identity([Fragment("OtherType")] object parameter)
  {
    return parameter;
  }

  public void Calls()
  {
    // warning: parameter qualifier type mismatch
    Identity(new object());

    AnnotatedDelegate annotatedDelegate;
    // method assignment behavior
    annotatedDelegate = Identity;
    // no warning: parameter matches delegate’s qualifier type
    annotatedDelegate(CustomTypeGenerator());
    // warning: parameter qualifier type mismatch
    annotatedDelegate(new object());

    // anonymous method behavior
    // warning: return type mismatch
    annotatedDelegate = delegate(object parameter)
    {
      return new object();
    };

    // lambda expression behavior
    annotatedDelegate = x => x;
  }

  [FragmentGenerator]
  [return: CustomType]
  public object CustomTypeGenerator()
  {
    return new object();
  }
}
```

Literals

For reasons of convenience, InjectionCop treats literals as a superior type matching with any custom type qualifier (Listing 4.28). This is a natural assumption because literals always denote a source and forcing the developer to implement a fragment generator wrapper method that returns a literal to introduce an appropriate custom type qualifier would result in producing “boiler plate” code.

Listing 4.28: Handling literals

```
class HandlingOfLiterals
{
  [return: CustomType]
  public string Foo([CustomType] int? parameter)
  {
    return "no warning raised";
  }

  public void NoWarningsRaised()
  {
    Foo(3);
    Foo(null);
  }
}
```

4.5 XML interface

As elaborated in Section 3.4, it is of great importance to define custom type qualifier interfaces at the system’s borders. This is achieved by an *XML* interface that is capable of setting Injection-Cop annotations. Listing 4.29 shows an example *XML* file, covering all annotations provided. Although this mechanism is intended for qualifying third party libraries, it can also be used for setting attributes of your own code base. This can result in obscure conflicts that are hard to identify. Since it is better style to define attributes, mainly because they are recognized by the type system, in case of a conflict, the attributes set in the source code overrule the *XML* settings. To avoid this confusing scenario, it is highly recommended to use *XML* definitions for third party libraries only.

Listing 4.29: Custom type qualifier definition file

```
<?xml version="1.0" encoding="utf-8" ?>
<Blacklist xmlns="http://injectioncop.codeplex.com/">
  <Assembly name="InjectionCopDemo">
    <Type name="TypeName">
      <Method name="Foo" returnFragmentType="CustomType" >
        <Parameter type="System.Object"
    fragmentType="CustomType"
    fragmentGenerator=""/>
      </Method>
    </Type>
  </Assembly>
</Blacklist>
```

Custom type qualifier definition files are identified by the extension ".injectioncop" and there are several possibilities where an *XML* qualifier definition file can be located. When custom type qualifiers are resolved, and no attributes are defined in source code, the first location that is scanned for an *XML* definition file is the assembly directory of InjectionCop. If no definition file is present, or the existing file does not contain a definition for the required entry, InjectionCop starts a search in the directory containing the assembly the type is defined in. Again, this can produce conflicts when multiple type qualifier definition files are present in one directory. Evaluation order of these files is not specified, and it is highly recommended not to spread definitions across multiple files in one directory.

4.6 Build process integration

For reasons of applicability, it is fundamental to integrate InjectionCop into build and development process neatly. For this purpose, the framework can be launched via command line, and configured to run within *Visual Studio* which provides a GUI.

Command line tool - FxCopCmd.exe

Starting static code analysis from shell is pretty simple which is shown in Listing 4.30.

Listing 4.30: Starting InjectionCopfrom console

```
$ cd 'C:\Program Files (x86)\Microsoft Visual Studio 11.0\Team Tools \
  \Static Analysis Tools\FxCop'

$ .\FxCopCmd.exe /file:C:\InjectionCopDemo.dll \
  /rule:C:\InjectionCop.dll /console
```

FxCopCmx.exe is the tool to use, and it is shipped with Visual Studio. Launching the analysis requires parameters "/file" which specifies the assembly to analyze, and "/rule" to state the rule base to use for analysis. Additionally, "/console" is given for activating console output. After firing the command, all warnings identified by InjectionCop are printed.An exemplary part of the output is shown in Listing 4.31.

Listing 4.31: Output of an analysis

```
...\AnnotatingInterfaces.cs(40,1) : warning  : IC0001 : \
InjectionCop.Analysis : Expected fragment of type 'CustomType' \
but got 'Empty'.

...\AnnotatingInterfaces.cs(47,1) : warning  : IC0003 : \
InjectionCop.Usage : Expected fragment of type 'CustomType' \
from implemented interface method, but got 'MismatchingType'.
```

The first warning corresponds to the code snippet for demonstrating implementation of an annotated interface (Listing 4.23), particularly the second warning of method `UnsafeFooCalls`. The format is straight forward and lists the affected file and line number, the "Action" which is set to warning, the id of the rule which is "IC0001", the category "InjectionCop.Analysis" and a customized output message for more detailed context information.

The second warning is related to the sample illustrating the fault of implementing an interface with mismatching custom type qualifiers (Listing 4.24). Some noteworthy differences are the different rule id and category to distinguish problems identified by analysis and wrong usage of InjectionCop.

Visual Studio integration

Visual Studio provides all interfaces needed for extending static code analysis by adding pluggable type checkers. The setup requires just a few steps and starts with defining a rule set for the project, which is just an *XML* file with ".ruleset" extension. An example rule set is given Listing 4.32.

Listing 4.32: Configuration file of a rule set

```
<?xml version="1.0" encoding="utf-8"?>
<RuleSet Name="New Rule Set" Description=" " ToolsVersion="11.0">
  <RuleHintPaths>
    <Path>C:\InjectionCop.dll</Path>
  </RuleHintPaths>
  <Rules AnalyzerId="Microsoft.Analyzers.ManagedCodeAnalysis"
    RuleNamespace="Microsoft.Rules.Managed">
    <Rule Id="IC0001" Action="Warning" />
    <Rule Id="IC0002" Action="Warning" />
    <Rule Id="IC0003" Action="Warning" />
    <Rule Id="IC0004" Action="Warning" />
  </Rules>
</RuleSet>
```

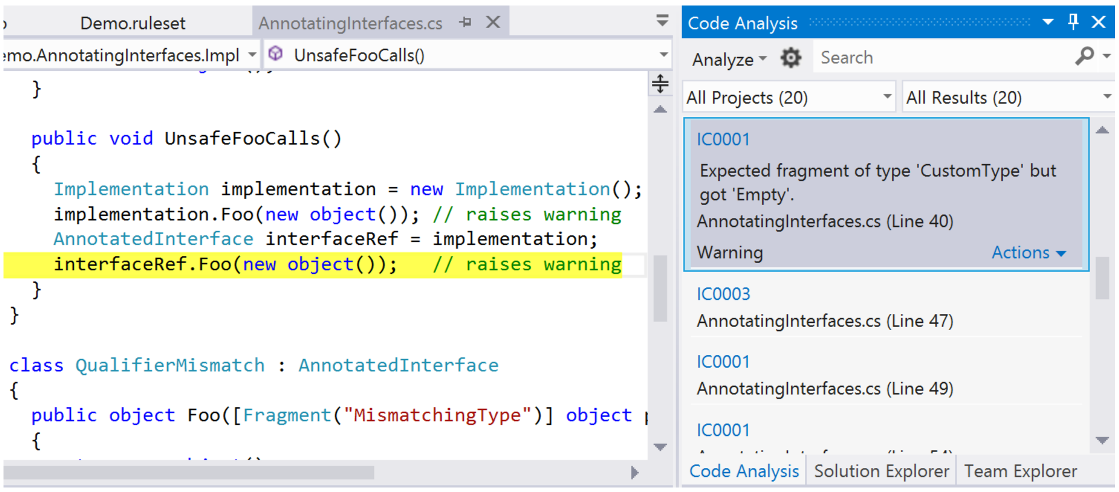
Most elements are self-explanatory. An assembly containing *FxCop* rules that should be considered, can be configured using the "RuleHintPaths" element, which in this example contains the path to the InjectionCop dll. To have fine grained control on which rules to activate, the "Rules" element represents a white list of rules, identified by id, that are executed during analysis process.

The next step is configuring the project to use the previously defined rule set. Those settings are located on the "Code Analysis" tab of the project properties where a rule set can be selected. Additionally, there is the handy option to enable code analysis on build which is recommended for taking full advantage of InjectionCop. This is all configuration work that needs to be done to integrate InjectionCop into *Visual Studio* and the project to analyze.

When a new build of the project is initiated, InjectionCop analyzes the code base and displays warnings on the code analysis view. Source context is resolved and affected code is highlighted. Figure 4.11 shows those features by displaying the same warnings as discussed in subsection about "FxCopCmd.exe" 4.6. For handling false positives, any entry from the code analysis problem list can be suppressed separately. This can be achieved by the "Actions" button that is part of the corresponding problem list element. This is just a shortcut for automatically generating a "SuppressMessage" attribute which is shown in Listing 4.33.

Listing 4.33: "SuppressMessage" attribute

```
[System.Diagnostics.CodeAnalysis.SuppressMessage("InjectionCop.Analysis",
\ "IC0001:TypeParser", MessageId = "IC0001")]
```



```
  }

  public void UnsafeFooCalls()
  {
    Implementation implementation = new Implementation();
    implementation.Foo(new object()); // raises warning
    AnnotatedInterface interfaceRef = implementation;
    interfaceRef.Foo(new object());   // raises warning
  }
}

class QualifierMismatch : AnnotatedInterface
{
  public object Foo([Fragment("MismatchingType")] object
  {
```

IC0001
Expected fragment of type 'CustomType' but got 'Empty'.
AnnotatingInterfaces.cs (Line 40)
Warning

IC0003
AnnotatingInterfaces.cs (Line 47)

IC0001
AnnotatingInterfaces.cs (Line 49)

IC0001

Figure 4.11: Code analysis view

Depending on practicability, those attributes can be added to source code or a separate suppress messages file.

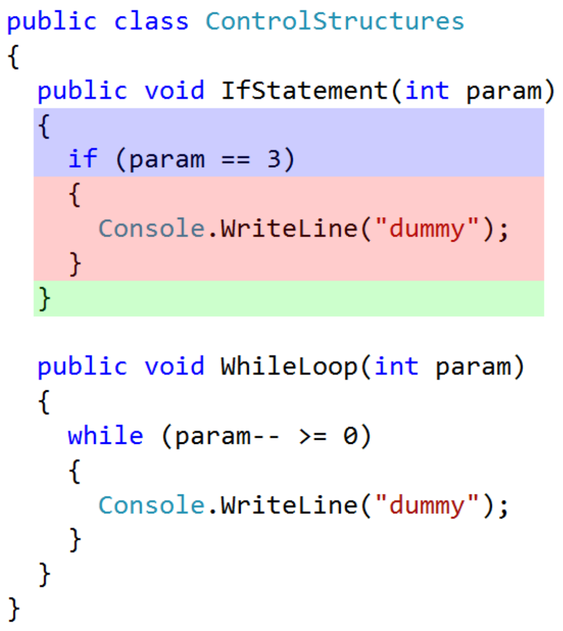
CHAPTER 5

Implementation

Applying the methodology outlined in previous chapters to analyze an executable involves working with code in different formats. The section begins with the life cycle of an analysis artifact, starting from source code that is compiled to an executable, whose *CIL* instructions are converted to an object model that is used by the framework. Another segment deals with algorithmic foundation of key elements of verification. The chapter concludes with performance analysis and findings that emerged in the process of gathering data.

5.1 Object model

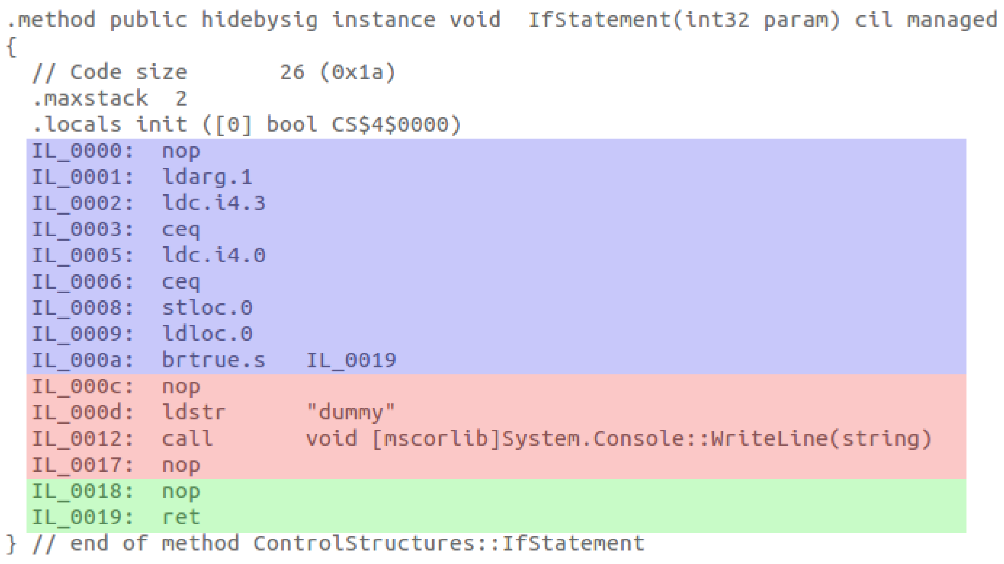
To understand the algorithm behind InjectionCop, it is necessary to take a closer look at the object model generated by *FxCop*. This model is mapped to a custom model representing an instance of the source and sink problem, which is tailored for the core algorithm. As indicated in Chapter 2.6, a model of the assembly is generated where executable members like methods or constructors are transformed to statements, blocks and jumps. To gain deeper understanding of these very abstract entities, several illustrations of some sample methods will be discussed. The purpose of the class in Figure 5.1 is providing the sample methods used to demonstrate mapping of source code, especially control structures, to the *FxCop* object model. The focus is on the method `IfStatement`, which is separated into three areas, recognizable by different colors. Each area represents a basic block. To gain better understanding of the basic blocks, especially the green block that seems to be empty in the source code, the corresponding *CIL* code is examined (Figure 5.2). The blue block contains all commands that evaluate the if conditional, it consists of load instructions, comparisons and a branch instruction. In this case, the branch instruction jumps past the if-statement if the condition is not satisfied. The red block embodies the true branch of the if condition, where a literal is loaded and an output function is called. A look at the green block, which seemed to be non existing due to its absence of statements in the source code, reveals that a return instruction is added by the compiler. Comparing the *CIL* code with the tree view of the object model generated by *FxCop* (Figure 5.4), indicates correlation of the sequence of instructions and the object graph. The method body aggregates a collection of



```
public class ControlStructures
{
  public void IfStatement(int param)
  {
    if (param == 3)
    {
      Console.WriteLine("dummy");
    }
  }

  public void WhileLoop(int param)
  {
    while (param-- >= 0)
    {
      Console.WriteLine("dummy");
    }
  }
}
```

Figure 5.1: Sample class with highlighted basic blocks



```
.method public hidebysig instance void  IfStatement(int32 param) cil managed
{
  // Code size       26 (0x1a)
  .maxstack  2
  .locals init ([0] bool CS$4$0000)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldc.i4.3
  IL_0003:  ceq
  IL_0005:  ldc.i4.0
  IL_0006:  ceq
  IL_0008:  stloc.0
  IL_0009:  ldloc.0
  IL_000a:  brtrue.s   IL_0019
  IL_000c:  nop
  IL_000d:  ldstr      "dummy"
  IL_0012:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0017:  nop
  IL_0018:  nop
  IL_0019:  ret
} // end of method ControlStructures::IfStatement
```

Figure 5.2: *CIL* instructions of the sample class with highlighted basic blocks

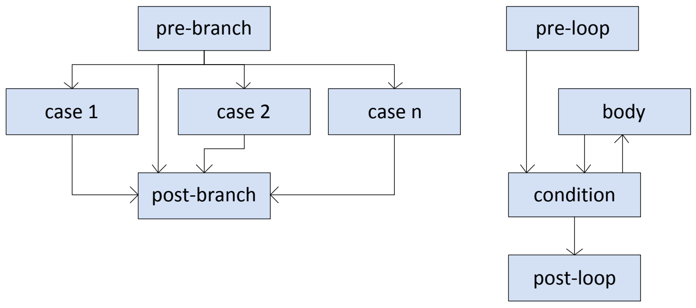


Figure 5.3: Method graph of a branch and loop statement

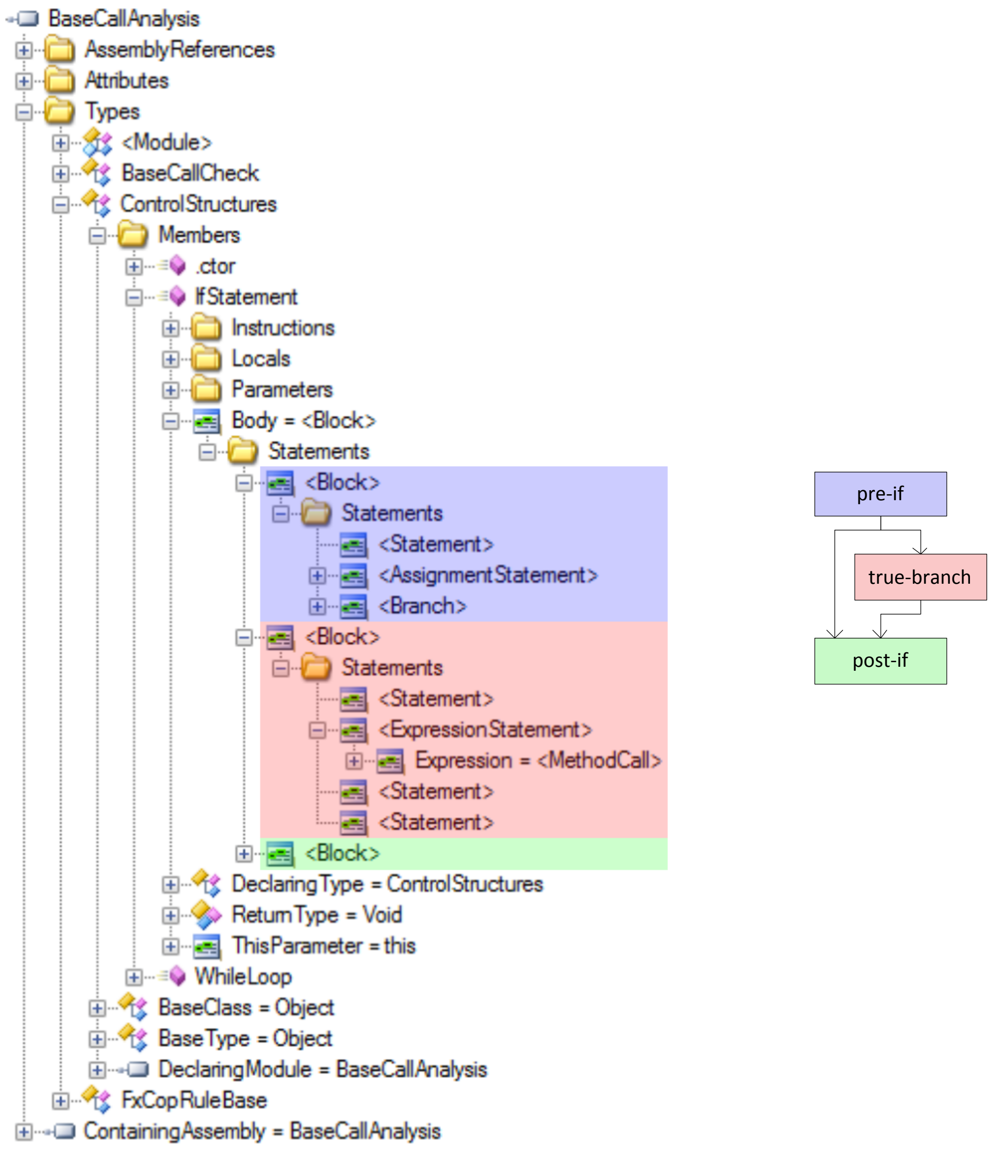
blocks where each block aggregates a collection of statements. The statement objects group several *CIL* instructions which leads to a more handy code model. Assignments, methods, branches and many more statement types, are wrapped by objects that reference each other according to semantics of the code. This implies that all branch statements within a basic block reference the appropriate succeeding block. Considering sequential execution of *CIL* commands, a block without a branch statement at the end introduces a transition to the subsequent block. Following these rules of edge definition, a graph as depicted in Figure 5.4 can be constructed. This graph represents the control flow of the method and is perfectly suitable for analyzing semantics. Another advantage of the graph generated from *CIL* code is that all control structures in the source code are mapped to graphs following general patterns for branch and loop statements (Figure 5.3). If statements, as well as switch statements or ternary if operators are mapped to an object model containing several branches as depicted on the left graph. All kinds of loops result in a graph that is similar to the right one.

The tree view shown in Figure 5.4 of the sample class gives insight into the objects acting as containers that form the model of the given code. It also shows complexity of the object model, since, there are loads of objects aggregating collections of subordinate objects starting from module level leading to types, members, blocks and statements. Additionally, there are collections for attributes, references to base classes of declaring members and lots of other details that are useful for code analysis.

Summarizing these observations, *FxCop* generates an object model by parsing *CIL* code. This object model can be mapped to a custom graph, tailored for static code analysis, of the control flow of any method. The graph consists of sub graphs following two specific patterns. It may contain circles if the corresponding method contains a loop, and the graph may branch when the methods includes a conditional.

5.2 Architecture

High requirements are put on the architecture because an executable is a very complex input to process. Starting from a program, *FxCop* generates the object model that is used for analysis. Building the object model is accomplished with the help of *CCI* [11]. *CCI* is provided by *Mi-*



```
BaseCallAnalysis
  AssemblyReferences
  Attributes
  Types
    <Module>
    BaseCallCheck
    ControlStructures
      Members
        .ctor
        IfStatement
          Instructions
          Locals
          Parameters
          Body = <Block>
            Statements
              <Block>
                Statements
                  <Statement>
                  <AssignmentStatement>
                  <Branch>
              <Block>
                Statements
                  <Statement>
                  <ExpressionStatement>
                    Expression = <MethodCall>
                  <Statement>
                  <Statement>
              <Block>
          DeclaringType = ControlStructures
          ReturnType = Void
          ThisParameter = this
        WhileLoop
      BaseClass = Object
      BaseType = Object
      DeclaringModule = BaseCallAnalysis
    FxCopRuleBase
  ContainingAssembly = BaseCallAnalysis
```

Figure 5.4: Tree view and graph representation of method “IfSample”

crosoft Research and stands for “Common Compiler Infrastructure”. It is a powerful collection of libraries and APIs, targeted to be used by compilers or similar tools. The object model is the foundation of the analysis and embodies different abstraction levels like types, methods and statements of the executable.

Analysis process

An illustration of all components and artifacts of the analysis process is shown in Figure 5.5. An analysis can be started from any of the *FxCop* launchers mentioned in the chapter about build

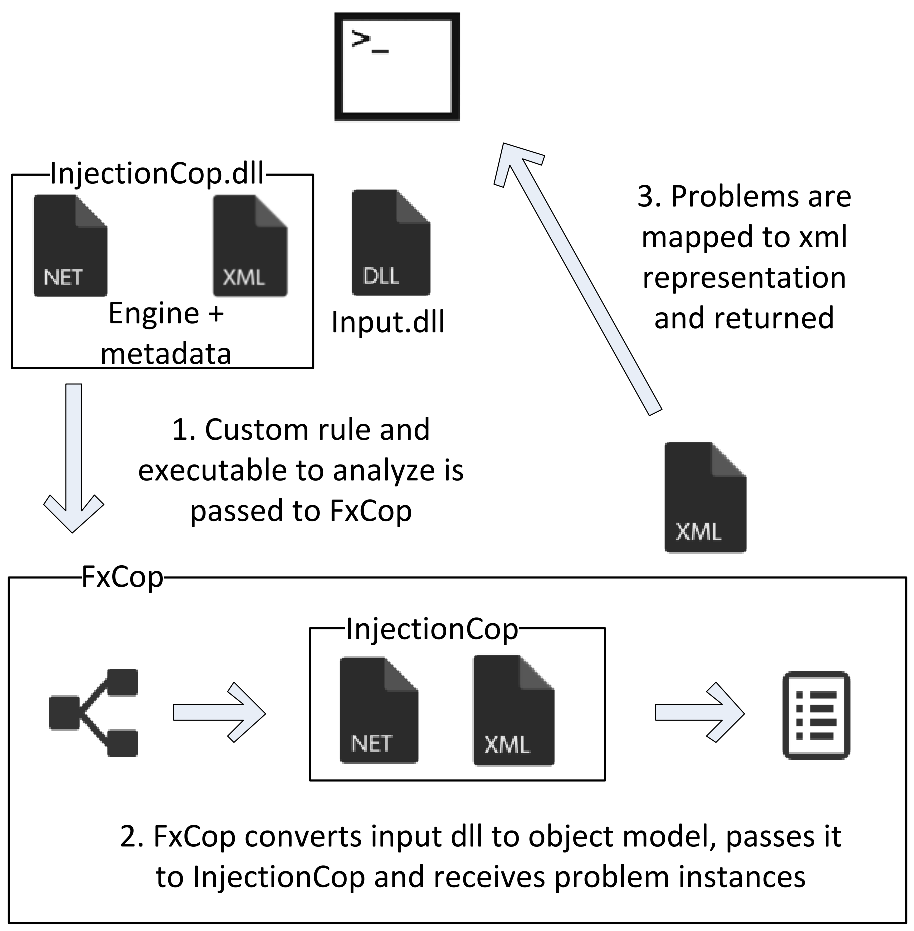

InjectionCop.dll
NET
XML
Engine + metadata
DLL
Input.dll
3. Problems are mapped to xml representation and returned
1. Custom rule and executable to analyze is passed to FxCop
XML
FxCop
InjectionCop
NET
XML
2. FxCop converts input dll to object model, passes it to InjectionCop and receives problem instances

Figure 5.5: InjectionCop's components and artifacts

process integration (Section 4.6). In the diagram, *FxCopCmd.exe* is used to pass the Injection-Cop dynamic link library and the input dll containing the executable to analyze. The dll is a *FxCop* custom rule that contains the engine, as well as a meta data *XML* file. The records of a meta data instance are self explanatory. A sample is shown in Listing 5.1.

Listing 5.1: InjectionCop meta data

```
<?xml version="1.0" encoding="utf-8" ?>
<Rules FriendlyName="InjectionCop">
  <Rule TypeName="TypeParser" Category="InjectionCop.Analysis" CheckId="IC0001">
    <Name>Check for Fragment Violations</Name>
    <Description>See injectincop.codeplex.com for quick introduction</Description>
    <Url>injectioncop.codeplex.com</Url>
    <Resolution>Expected fragment of type '{0}' but got '{1}'.</Resolution>
    <MessageLevel Certainty="95">Warning</MessageLevel>
    <FixCategories>NonBreaking</FixCategories>
    <Email />
    <Owner />
  </Rule>
</Rules>
```

The data structure covers information that is used for selecting and displaying rules before the analysis, like “Name” or “CheckID”, but also for representing and interpreting the output of an analysis. “FixCategories” and “Resolution” are examples for values that support further processing of the result of the analysis. The next step is converting the input dll to an object model that is passed to the custom rule. During analysis, a custom rule can report issues by adding objects to a collection of type “Problem” that is provided by *FxCop*. An instance of `Problem` is basically a container holding a resolution generated from rule meta data, the id of the rule that is violated and a source context. Source context is a convenient way to identify the position of the violation in the code that can be interpreted by *Visual Studio* automatically. When the analysis is finished, *FxCop* processes all problems raised by the rule and generates an output *XML* file that is returned to the launcher of the analysis.

InjectionCop modules

FxCop provides several extension points on different levels of detail that can be utilized by custom rules. This is realized by a base class following *visitor pattern*, which introduces a mechanism to add virtual functions to a set of classes without changing them [21], whose overloads of visit methods take parameters of an object model type from *FxCop*. These visitors enable analysis starting from whole types down to specific statements of an executable. For simple code analysis engines, examining specific statements may be sufficient. For example, easy tasks like verifying that a specific method from an external library is not used, can be accomplished by checking method calls only. In other problem domains, analyzing on method level may be adequate. However, implementing a custom type qualifier engine by examining statements or methods isolated is inconvenient. Analyzing deterministically, which is not guaranteed when using visitors provided by *FxCop*, by parsing types starting with fields and continuing with methods simplifies processing of relations between fields and methods. Nonetheless, introducing different levels of abstraction is an excellent way to handle high complexity of input. The architecture of InjectionCop features layers for analysis on type, method and statement level (Figure 5.6). The input object model is passed to the type parsing module of the top layer which is capable of collecting information about fields and handing off analysis of methods to the next layer. The method parsing layer generates a graph representation of a method with the help of a block parsing layer at the bottom. Any of those layers can detect problems that are returned to *FxCop* by a problem pipe module.

5.3 Algorithm

The chapter about verification (Section 4.2) dealt with the core concepts of the algorithm by analyzing deduction trees. This section focuses more on the implementation by discussing pseudo code of crucial code parts. In this section, modules for walking through the inference tree, analyzing method signatures and modules for applying *Hoare* rules are addressed.

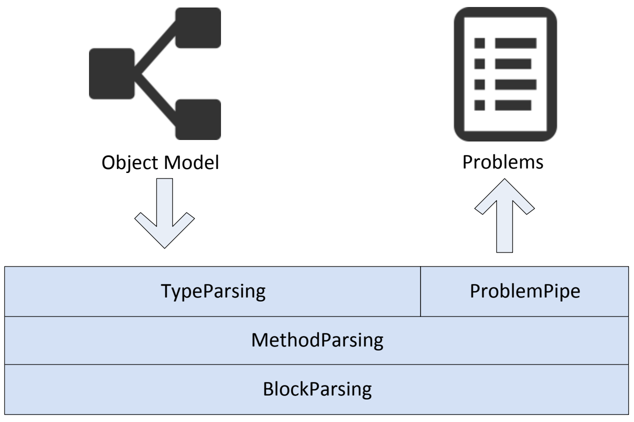


Figure 5.6: InjectionCop's architecture

Parsing the inference tree

As seen in deduction trees, when a method is analyzed, the verification engine has to walk through each statement and maintain the current state of the variables. Depending on the type of the current statement respective actions have to be enforced. These actions are hook points for other modules covered in this section.

Analysis is implemented in the `Inspect` method. A method is represented by an ordered set of code blocks, which are analyzed separately. The entry point for this inspection per code block is given by the first method which takes the block as a parameter. Similar to methods, each block is an ordered set of statements. Each statement of the block is passed to an overload of the `Inspect` method where a distinction of different statement types is performed. For simplification, only assignments and method calls are considered in the outline of the algorithm (Listing 5.2).

Listing 5.2: Handling of different kinds of statements

```
void Inspect (Block block)
{
  foreach(statement in block)
    Inspect(statement);
}

void Inspect (Statement statement)
{
  if(statement is AssignmentStatement)
    Inspect ((AssignmentStatement) statement)
  else if (statement is MethodCall)
    Inspect ((MethodCall) statement)
  else if
    ...
}
```

The `Inspect` method on statement basis is the controller that provides core hook points. These hook points are used by statement handlers that have to cover a wide range of types. Besides assignments and method calls these types include

- Branch
- Switch
- Return
- Indexer

Although there a plenty of handlers covering special cases only the most important ones are addressed here.

Representation of states

The current state in the inference phase is encapsulated in an instance of a `SymbolTable` (Listing 5.3). The current type qualifier of any symbol of the state is stored in a dictionary. The key of the map is the symbol name, the value is an instance of `Fragment` that represents the custom type qualifier. Symbol names include identifier for all possible data sources like variables, fields or parameters. These names are generated by the compiler so that no name clashes can occur.

Listing 5.3: Symbol table

```
public class SymbolTable : ISymbolTable
{
  private Dictionary<string, Fragment> _safenessMap;

  public Fragment InferFragmentType (Expression expression)
  {
    if (expression is Literal){...}
    else if (expression is Local){...}
    else if (expression is Parameter){...}
    ...
  }

  private Fragment InferMethodCallReturnFragmentType (MethodCall methodCall){...}
  public bool IsAssignableTo(string symbolName, Fragment fragmentType){...}
  public Fragment[] InferParameterFragmentTypes (Method method){...}
  public void MakeUnsafe (string symbolName){...}
  public void MakeSafe (string symbolName, Fragment fragmentType){...}
  private Fragment Lookup (string name){}
  ...
}
```

Besides the dictionary for tracking the state of the inference, the symbol table contains many helper methods. Most of them deal with inferring a custom type qualifier for a certain code construct. These methods provide inferring expressions, which is similar to the inspect method, as well extracting the return type of a method call or qualifiers of parameters. Methods `MakeSafe` and MakeUnsafe act like setters and enable the caller to define a custom type qualifier for a specific symbol. Lookup is a getter method that returns the current fragment for a given symbol, IsAssignableTo implements logic to check if a certain symbol can be assigned to a target that requires a specific custom type qualifier. Implementation details of these methods are omitted because the code is straight forward and not relevant to get the idea of the algorithm. The usage of the helpers can be seen in the following pseudo code snippets.

Generating pre- and postconditions

After initial analysis of the input executable by the type parsing layer, conditions are extracted from the signature of a method in the method parsing layer. Generating pre- and postcondition is not a very complex problem that can be solved by reflection and iterating over parameter and return value annotations. Extraction is a phase that has to happen before building of the method graph because postconditions are needed by the block parsing layer below. The block parsing layer needs to know about return values because type qualifier mismatches on block level are added to the problem pipe immediately. When pre- and postconditions are determined, the method graph is built and the framework can continue with examining type qualifier mismatches between code blocks. In order to achieve this, the initial state consisting of preconditions holding the types of the input parameters and the types of fields is passed to a graph analyzer engine.

Parsing a block

Before relations between code blocks are analyzed, each code block is parsed independently. Any problems that can be identified by solely stepping through the commands of the block are raised immediately. In the sample shown in Listing 5.4, both method calls are analyzed by the block parser. There is no context needed to check the second call. Therefore, the framework recognizes that the qualifier of variable k is T and the call is correct.

Listing 5.4: Method with a single basic block

```
[return: T]
int NeedsT([T] int i) {...}

void Inspect (int j)
{
  int k = NeedsT(j);
  NeedsT(k);
}
```

Transitions from one block to another are enforced by branching and looping statements, as well as the beginning of a method. This is an artificial transition from the initial state to the first command of the method. To be able to analyze transitions, meta data has to be generated for the block. In this case, it is necessary to add a precondition to the block that expresses that the symbol j demands the qualifier type T. Additionally, symbol k has a qualifier of type T after the block is executed, which is expressed by a postcondition of the block. When transitions are analyzed, the framework has to match preconditions with the current state, which implies raising a problem because parameter j is not qualified. If j would be of qualifier type T, the framework could continue analysis by merging the postconditions with the current state. In this sample, qualifier type T would be assigned to symbol k in the symbol table.

Assignments

As already mentioned, an isolated examination of expressions or blocks is not sufficient to recognize all type qualifier mismatches. When an assignment is analyzed, two cases have to be considered. If the custom type qualifier of the source can be derived, for example when the source of the assignment is the return value of a method call or a variable with a known type, the symbol table is adjusted accordingly. This is the easy case that can be handled without analyzing a different basic block. When the source of the assignment is defined outside the current block, a mechanism to express this dependency is needed. The problem is shown in Listing 5.5.

Listing 5.5: Method with multiple basic blocks

```
[return: T]
int ReturnsT() {...}

[return: T]
int Foo ()
{
  var x = ReturnsT();
  var y = 0;
  var z = ReturnsT();

  if(...)
  {
    y = z;
    x = y;
  }

  return x;
}
```

The block containing the body of the if statement changes the state of variable x, but the type cannot be derived by solely parsing the block. A history is needed that memorizes sources of variable assignments. The pseudo code in Listing 5.6 handles both cases mentioned in this section and sets the assignment history.

Listing 5.6: Handling of multiple basic blocks

```
void Inspect (AssignmentStatement assignment)
{
  if ("assignment source and targets are variables"
    && "source not assigned inside current block")
  {
    AssignmentHistory.Add(new Assignment(assignment.Source, assignment.Target);
  }
  else
  {
    DeriveType (assignment);
  }
  Inspect (assignment.Source);
}
```

When the method is parsed, two history entries expressing that the type qualifier of x is defined by y and that the type qualifier of y if defined by z are added. The history is implemented as a stack, so the transitivity of the assignments can be resolved by the order of the entries.The call of the inspect method at the last line of the pseudo code also shows the recursive character of the inspect infrastructure since the source expression of the assignment is also inspected.

Resolving method calls

Resolving a method call is a scenario similar to assignment statements. When an operand of a method call is defined outside the current block, this information needs to be accessible to the method parsing layer. Again, the history is used when a method call is analyzed in the respective overload of `Inspect` (Listing 5.7). The method compares operand types with types defined in the method signature. Matching is realized in `MatchOperand` where a precondition is added when the operand is defined outside of the current block. The check is performed by querying the stack of the history. If the type of the operand is defined inside the block, matching can be done on the fly without the need for creating a precondition.

Listing 5.7: Handling of method calls

```
void Inspect (MethodCall methodCall)
{
  for (int i = 0; i < methodCall.Operands.Length; i++)
    MatchOperand(methodcall.Operand[i], methodCall.ParameterType[i])

  foreach(operand in methodCall.Operands)
    Inspect (operand);
}

void MatchOperand (Operand operand, TypeQualifier parameterType)
{
  if("operand is not set inside the block")
    AddPreCondition(operand);
  else
    MatchTypes(operand, parameterType);
}
```

The difference to handling assignments is that a precondition must be added, because in case of a type qualifier mismatch a problem needs to be raised. This is not the case for assignments since they are harmless until the variable is used. Once more, the recursion of the inspection is kept alive by passing each operand to the `Inspect` method.

Analyzing conditionals

In *Hoare logic*, each if statement splits the deduction tree into two paths. The paths ensure separate handling of the cases of the statement and cover the *consequent* and the *alternative*. To

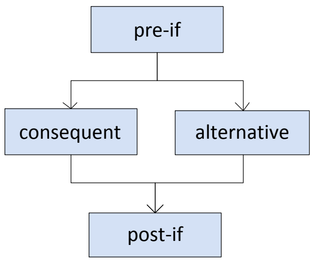


Figure 5.7: Method graph of an if statement

simulate this, the framework has to set up branches according to branching statements of the input code and follow each branch. Listing 5.8 shows a basic code skeleton of an if statement.

Listing 5.8: If statement skeleton

```
void Foo ()
{
  ... // pre if
  if(...)
    ... // consequent
  else
    ... // alternative
  ... // post if
}
```

The graph representing the method is shown in Figure 5.7. Each element of the graph represents a basic block containing a collection of successors. Generation of the graph is challenging, hence it is encapsulated in a separate block parsing layer. To generate the graph, resolution of basic block references is performed by InjectionCop with the help of statement handlers for branch and switch instructions. These are the only two types of instructions that reference successors. Basic blocks are generated by *FxCop* and identified by a unique key which is also used to define a reference to another basic block. Branching of the tree is evaluated by following two paths for the consequent and the alternative (Figure 5.8). The state before branching $\{S1\}$ is cached and and used for the traversal of the sub trees for the *consequent* and the *alternative*. As already mentioned, there are two sub trees covering the instructions for the *consequent* and the *alternative*.

Analyzing loops

One of the core aspects identified in the chapter about verification was generating an invariant that describes the semantics of a loop. In the problem domain of InjectionCop, this comes down to identifying variables with ambiguous types that are introduced by the loop and usage of these variables. An example of a ambiguous variable is given in Listing 5.9.

Listing 5.9: Sample of an ambiguous variable

```
void Foo ()
```

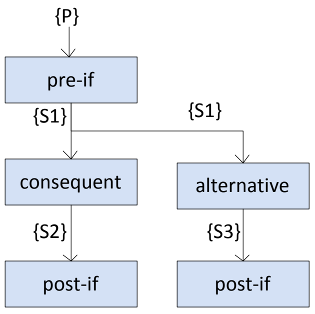


Figure 5.8: Traversal of an if statement

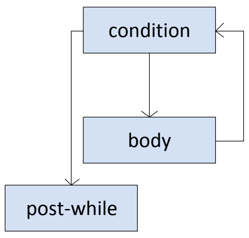


Figure 5.9: Method graph of a loop

```
{
  x = T1
  while(...)
    x = T2
  ... // x = T1 or x = T2 => x is ambiguous
}
```

If variable x is of type $T1$ before the loop and the type of x is changed to $T2$ in the body of the loop, the type of x is ambiguous after the loop because it depends on whether the body is executed or not. Furthermore, the ambiguity of x does not necessarily introduce a type qualifier mismatch, because x may not be used any more.

The basic skeleton of a loop is given in Listing 5.10, the corresponding graph is shown in Figure 5.9.

Listing 5.10: Loop skeleton

```
void Foo ()
{
  while(...)
    ... // body
  ... // post while
}
```

To derive the ambiguities that define the invariant, some paths in the graph must be checked for type qualifier mismatches. For each set of conditions that holds before node *condition* a path to *post-while* and a path to *body* must be evaluated, which covers optional execution of the

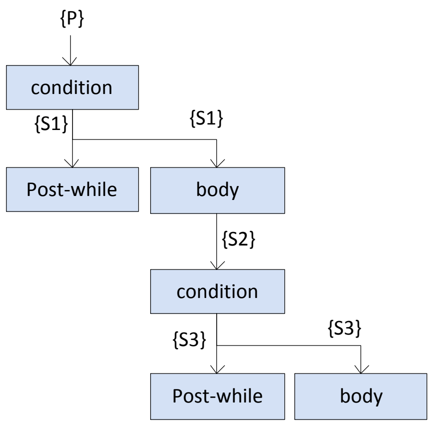


Figure 5.10: Traversal of a loop

body. These sets of conditions that hold before the loop are defined by the ingoing edges of node *condition*. The sources of the ingoing edges are the block before the loop and the node *body*, which cover the states of the initial execution of the loop and the state representing further iterations. Deriving the invariant entirely before the analysis continues with the basic block after the loop is not hard for the given sample, but challenging in case of nested loops.

To overcome this issue, the approach of InjectionCop is mapping the graph to an equivalent tree (Figure 5.10). The initial state of the traversal is $\{P\}$, which contains the conditions that hold before the loop, node *condition* is evaluated which results in state $\{S1\}$. The subsequent paths handle possible results of the condition. The left path evaluates the case when the body is not executed at all, the right path evaluates the first execution of the body. At this point of the analysis, evaluation of the conditions that hold before the loop is finished and evaluation of cases where the body is executed at least once is started. $\{S2\}$ is the state that contains the conditions that hold after the body is executed the first time. Continuing from this state, the same options as for the initial state $\{P\}$ are evaluated again. This time, the left path evaluates the transition from the body to the blocks after the loop, the right path evaluates multiple sequential executions of the body.

CHAPTER 6

Performance

To gain information about applicability of the framework, performance statistics are presented. During creation of the statistics, weaknesses of the implementation could be identified that are discussed in this section.

Performance analysis

To evaluate the performance of the proposed approach, the well-known open source libraries *Nunit* [55], *log4net* [44], *RhinoMocks* [64], *NHibernate* [53] were parsed. Additionally, *mscorlib*, which stands for *Common Object Runtime* and is the core library of a *.NET* language, was also parsed. The sizes of the assemblies are between 71 kB and 4444 kB to examine the impact of the file size on the analysis. In some cases, the framework performed worse than anticipated. Figure 6.1 shows a corner case were runtime of the analysis took very long. What strikes the eye is that the method is bloated with control structures and logical expressions. As observed in the previous chapter, conditionals split an execution path into multiple sub paths that are analyzed seperately. This implies that a sequence of conditionals can broaden the tree of execution paths exponentially. The method in the sample contains many conditionals that are not obvious. Logical operators with *short-circuit* evaluation, which are `&&` and `||` in *C#*, introduce conditionals, because evaluation stops as soon as the result is clear. There are enough logical operators in the sample to increase the number of execution paths to a level where performance impact is observable.

After mitigating performance issues, runtime statistics were gathered (Figure 6.2). The tests were run on a *Parallels Desktop 8* virtual machine with 4 GB RAM and 4 cores assigned, the physical processor was a *Intel Core i7* with 2.6 GHz. Analysis was performed on *Windows 8* with *FxCop 11.0* which is shipped with *Visual Studio 2012*. *FxCop* was configured to utilize 4 threads. The first column of the table holds the name of the assembly, the following columns cover file size and the number of methods of the assembly, which is an indicator of the complexity of the analysis. The remaining columns state duration of compilation and analysis, as well as the relation of compilation duration and analysis duration. The durations were determined by

```
public string GetValidIdentifier (string str)
{
  if (...)
    ...
  else
    ...

  for (...)
  {
    if (...)
      ...


    char c = str[i];
    bool isValid = false;

    if (StringArray != null)
    {
      string replaceString = (string) StringArray[c];
      if (replaceString != null)
        isValid = true;
    }

    if (isValid
      || (allowLanguageSpecificLetters
         && char.IsLetter (c))
      || (! allowLanguageSpecificLetters
         && allowEnglishLetters
         && ((c >= 'a' && c <= 'z')
            || (c >= 'A' && c <= 'Z')))
      || (allowDigits
         && char.IsDigit (c))
      || (allowAdditionalCharacters != null
        && allowAdditionalCharacters.IndexOf (c) >= 0))
    {
      sValid = true;
    }
    ...

  }
  return sb.ToString();
}
```

Figure 6.1: Depiction of a method with high performance impact on analysis

assembly	size (kB)	#methods	compilation (sec)	analysis (sec)	percentage
nunit.framework.dll	144	1458	0.438	1.001	229%
log4net.dll	296	2183	0.799	1.245	156%
Rhino.Mocks.dll	311	2437	0.359	1.400	389%
NHibernate.dll	3260	21463	5.340	11.476	229%
mscorlib.dll*	4444	21485	-	2.850	-

Figure 6.2: Results of the performance analysis

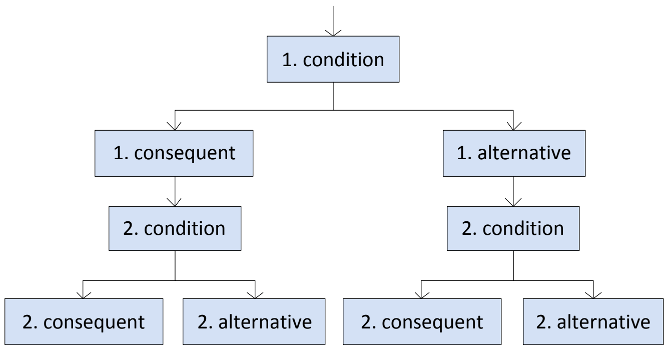
calculating the arithmetic mean of the measurements after 10 runs, calculation of the percentage is based on these mean values. The time of compilation covers a rebuild in *Debug* mode. This is reasonable because *Debug* mode enables *FxCop* to provide source code locations and symbol names to increase the quality of the output of the analysis [20]. *mscorlib.dll* is marked with an asterisk because source code is not available, and compilation duration could not be determined. Nonetheless, the analysis results attract attention. Although *mscorlib.dll* is more than 1 MB bigger than *NHibernate.dll* the number of methods is almost identical. This is probably an effect of a focus on performance, and application of optimization techniques like *method inlining* and reducing method calls by using long methods. But what strikes the eye is that the analysis of *mscorlib.dll* is significantly faster than the analysis of *NHibernate.dll*, which could be caused by the absence of referenced assemblies. Besides the number of referenced assemblies, also the number of methods impact performance. Durations of the open source frameworks show that the duration increases according to the number of methods. The relation of compilation duration and analysis duration does not correlate to any other number, but shows that analysis takes longer than compilation. It is noteworthy that the analysis is performed on a code base without *custom type qualifiers*. This means that InjectionCop generates the graph of a method, but can cancel evaluation because there are no qualified types in the signature. A worst case analysis, where the source code was changed so that every method is analyzed, showed that the duration of the analysis of *log4net.dll* can reach 15 seconds, analysis of *Nhibernate.dll* was stopped after some minutes. On the other hand, analyzing all methods of *Rhino.Mocks.dll* was just 0.3 seconds slower than analyzing no method. When looking at the field of application in the *re-motion* framework, it can be assumed that using the InjectionCop will not lead to extensive usage of *custom type qualifiers*. Instead, the annotations are very specific and spread around the modules handling security, which is expected to mitigate the performance issue. Since the performance impact is potentially high and a quick build is preferred during development, using the framework all the time may be inconvenient. Nonetheless, temporarily activating static code analysis is definitely an acceptable way to go. An alternative use case for the framework is integration into the build process on a build server where duration is less important. Static code analysis fits well into common build server duties like running tests, packaging or generating code metrics.

Complexity analysis

The section covering algorithms showed that branches and circles in the method graph are resolved by following multiple paths. The effect of this evaluation strategy is that code blocks are evaluated multiple times. Sequencing and nesting of control structures has the potential to provoke runtime problems. `Foo` is a method with two if statements in the body.

```
void Foo ()
{
  ... // pre if
  if(...)   // first if
    ...
  else
    ...
  if(...)   // second if
    ...
  else
    ...
}
```

To evaluate the method, the following paths are analyzed.

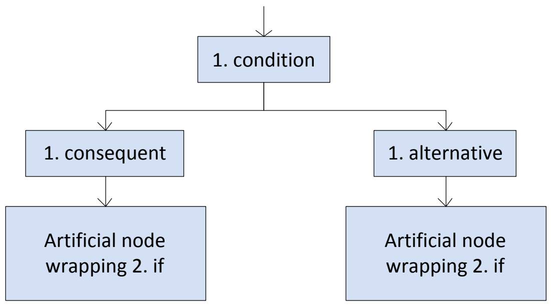


For any possible path introduced by the first if statement, all possible paths introduced by the following if statement are resolved. The complexity grows exponentially depending on the degree of nesting. According to this observation, loops imply even more paths to analyze. As a consequence, the complexity of the analysis of a method is determined by the complexity of the input methods.

In practice, this problem should hardly emerge, since short methods with a low degree of nesting are considered good programming style and are preferred in high quality code [48]. Nonetheless, this issue has to be addressed, because the framework should not rely on guidelines and in scenarios of performance optimization or legacy code such methods may appear.

Performance improvements

This section sketches an algorithm to bypass performance issues introduced by an exploding number of paths in the method graph. The order of analyzing blocks must be changed to avoid the problem, top down evaluation is discarded and the graph has to be simplified. To achieve this, the nodes forming the second if statement are interpreted as an independent tree that is analyzed first. The algorithm needs to generate all necessary preconditions that an initial state for the sub tree has to fulfill. The collection of the preconditions is used to build an artificial block that acts as a placeholder for the if statement. Afterwards, the tree has to be traversed to replace all occurrences of the sub tree by its placeholder. The following figure shows the graph after transformation.


1. condition
1. consequent
1. alternative
Artificial node wrapping 2. if
Artificial node wrapping 2. if

In this approach, also simplification of sequences of blocks and loops need to be considered and the graph has to be simplified until analysis can be performed in reasonable time.

CHAPTER 7

Summary and future work

InjectionCop supports developers on addressing security requirements. To illustrate integration of the tool into the development process, measures to prevent *cross-site scripting* are shown. The scenario is a web application that has to process user input, to render *html* that is returned to user. The application is developed in *ASP.NET MVC*, which is a framework following the *model view controller pattern* [52].

The first step to utilize InjectionCop is, referencing *InjectionCop.dll*. This enables defining the *custom type qualifier* `Sanitized` (Listing 7.1) to express the cross cutting concern of preventing *XSS*.

Listing 7.1: Attribute to define the `Sanitized` type qualifier

```
[AttributeUsage(AttributeTargets.Parameter
                   | AttributeTargets.ReturnValue
                   | AttributeTargets.Field
                   | AttributeTargets.Property
                   | AttributeTargets.Constructor)]
public class SanitizedAttribute : FragmentAttribute
{
  public SanitizedAttribute() : base("Sanitized") { }
}
```

The type qualifier can be used to specify a consumer of sanitized data. In this sample, the *consumer* is the `Render` method (Listing 7.2), which is not fully implemented because the actual code in not relevant for illustration. The purpose of the method is wrapping the user input into *html* that is part of the response.

Listing 7.2: Consumer class of sanitized data

```
public class Renderer
{
  public Output Render([Sanitized] string data)
  {
    // wrap input data into html that included in the response
    return new Output { Data = data };
  }
}
```

The counterpart of the *consumer* is the *producer*. The *producer* is method `Sanitize` (Listing 7.3), which ensures *XSS* prevention by encoding passed data. The *custom type qualifier*, the *producer* is capable of is defined by annotating the return value with the attribute `Sanitized`. Additionally the attribute `FragmentAttribute` is set on the method to mark a *producer* method and turn off verification. If `FragmentAttribute` would be missing, the verification engine would detect a *type qualifier mismatch* because method `HtmlEncode` returns an unqualified string but method `Sanitize` is obliged to return a string of the qualified type `Sanitized`.

Listing 7.3: Producer class of sanitized data

```
public class Sanitizer
{
  [FragmentGenerator]
  [return: Sanitized]
  public string Sanitize (string input)
  {
    return System.Web.HttpUtility.HtmlEncode(input);
  }
}
```

Listing 7.4 shows handlers of *http* requests that demonstrate secure and insecure usage of the method `Render`. Method `Secure` sanitizes correctly and does not provoke a *type qualifier* mismatch. On the other hand, method `Insecure` passes input data directly to the renderer, which is detected by InjectionCop.

Listing 7.4: Usages of `Renderer`

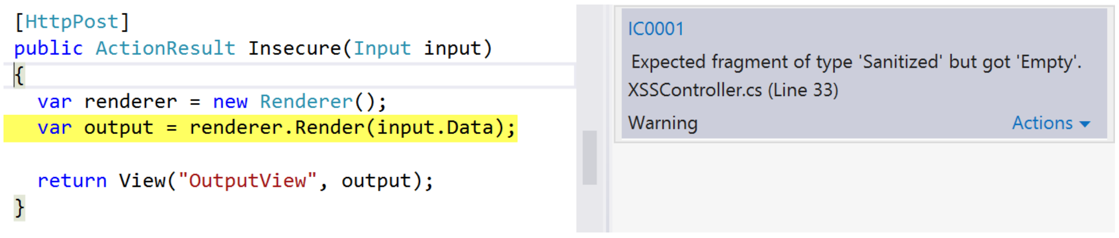
```
[HttpPost]
public ActionResult Secure(Input input)
{
  var sanitizer = new Sanitizer();
  var renderer = new Renderer();
  var sanitizedData = sanitizer.Sanitize(input.Data);
  var output = renderer.Render(sanitizedData);

  return View("OutputView", output);
}

[HttpPost]
public ActionResult Insecure(Input input)
{
  var renderer = new Renderer();
  var output = renderer.Render(input.Data);

  return View("OutputView", output);
}
```

Enabling *Visual Studio*, as described in Section 4.6, would result in warning that references the location in the source code where the mismatch was detected (Figure 7.1).



```
[HttpPost]
public ActionResult Insecure(Input input)
{
  var renderer = new Renderer();
  var output = renderer.Render(input.Data);

  return View("OutputView", output);
}
```

IC0001
Expected fragment of type 'Sanitized' but got 'Empty'.
XSSController.cs (Line 33)
Warning

Figure 7.1: *Type qualifier* mismatch

InjectionCop [35] is a tool that introduces *custom type qualifiers* to *C#*. *Custom type qualifiers* enable tackling of cross cutting security concerns in the development process. These security relevant concerns include all kinds of injections like *SQL* injection and input validation in general. The core engine is reliable and based on *Hoare Calculus*, which was shown by deducing type qualifiers in scenarios covering all relevant language constructs. The implementation is based *FxCop*, which is an engine for static analysis of *.NET* languages, and can be integrated into the build process without any problems. Statistics regarding performance were gathered by analyzing some well known open source frameworks. Potential performance issues were identified, examined and an improvement of the algorithm was proposed.

In the future, InjectionCop will be integrated into the *re-motion* framework to cope with cross cutting security concerns. Application in a productive environment will give insights into the demand of performance improvements that will be addressed if necessary.

Bibliography

[1] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] J. Kinder A. Holzer and H. Veith. Using verification technology to specify and detect malware. In *Proceedings of the 11th International Conference on Computer Aided Systems Theory*, EUROCAST'07, pages 497–504, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] C. Kruegel A. Moser and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 231–245, May 2007.

[4] C. Kruegel A. Moser and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[5] J. HeeDong A. Godiyal S. T. King AM. Nguyen, N. Schear and H.D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 441–450, Dec 2009.

[6] K. R. Apt. Ten years of hoare's logic: A survey&mdash;part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981.

[7] G. Balakrishnan and T. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[8] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] M. Huntley C. Thompson and C. Link. Virtualization detection: New strategies and their effectiveness.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[11] Microsoft Research CCI. `http://research.microsoft.com/en-us/projects/cci/`. Accessed: 2014-09-07.

[12] Checker Framework, Computer Science and Engineering at University of Washington. `http://types.cs.washington.edu/checker-framework/`. Accessed: 2014-09-07.

[13] CodeSurfer, GrammaTech. `http://www.grammatech.com/research/technologies/codesurfer`. Accessed: 2014-09-07.

[14] S. Cook. A web developer's guide to cross-site scripting. Technical report, SANS Institute, 2003.

[15] CQual, J. S. Foster. `http://www.cs.umd.edu/~jfoster/cqual/`. Accessed: 2014-09-07.

[16] D. Reynaud D. Babić and D. Song. Recognizing malicious software behaviors with tree automata inference. *Form. Methods Syst. Des.*, 41(1):107–128, August 2012.

[17] C. Techaubol D. Duggan. Modular mixin-based inheritance for application frameworks. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 223–240, New York, NY, USA, 2001. ACM.

[18] S. L. Graham D. F. Bacon and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[19] J. S. Foster D. Greenfieldboyce. Visualizing type qualifier inference with eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '04, pages 57–61, New York, NY, USA, 2004. ACM.

[20] Debug and MSDN Release Configurations. `http://msdn.microsoft.com/en-us/library/wx0123s5.aspx`. Accessed: 2014-09-07.

[21] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] E. A. Emerson E. M. Clarke and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, November 2009.

[23] R. Wa E. Meijer and J. Gough. Technical overview of the common language runtime, 2000.

[24] Eclipse, Eclipse Foundation. `https://www.eclipse.org`. Accessed: 2014-09-07.

[25] T. Ekman and G. Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, December 2007.

[26] E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 328–337, Oct 1988.

[27] M. Fähndrich. Static verification for code contracts. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 2–5, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Secure Coding Guidelines for Java SE. `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`. Accessed: 2014-09-28.

[29] FxCop, Microsoft. `http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx`. Accessed: 2014-09-07.

[30] A. Mendhekar C. Maeda C. Lopes J. Loingtier G. Kiczales, J. Lamping and J. Irwin. Aspect-oriented programming. In S. Matsuoka M. Akşit, editor, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[31] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 321–336, New York, NY, USA, 2007. ACM.

[32] W. Dietl H. Wei, A. Milanova and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2012)*, pages 879–896, Tucson, AZ, USA, October 23-25, 2012.

[33] M. Egele C. Kruegel H. Yin, D. Song and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[34] IdaPro, Hex-Rays. `https://www.hex-rays.com/products/ida/`. Accessed: 2014-09-07.

[35] InjectionCop. `http://injectioncop.codeplex.com`. Accessed: 2014-09-07.

[36] K. Leino M. Rustan P. Müller J. Hatcliff, G. T. Leavens and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[37] C. Schallhart J. Kinder, S. Katzenbeisser and H. Veith. Detecting malicious code by model checking. In *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'05, pages 174–187, Berlin, Heidelberg, 2005. Springer-Verlag.

[38] C. Schallhart J. Kinder, S. Katzenbeisser and H. Veith. Proactive detection of computer worms using model checking. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):424–438, Oct 2010.

[39] M. Fähndrich J. S. Foster and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 192–203, New York, NY, USA, 1999. ACM.

[40] U. Kargen and N. Shahmehri. Inputtracer: A data-flow analysis tool for manual program comprehension of x86 binaries. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 138–143, Sept 2012.

[41] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 61–70, Washington, DC, USA, 2012. IEEE Computer Society.

[42] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 43–50, Oct 2010.

[43] Lattice, Wolfram Math World. `http://mathworld.wolfram.com/Lattice.html`. Accessed: 2014-09-07.

[44] Apache Software Foundation log4net. `http://logging.apache.org/log4net/`. Accessed: 2014-09-07.

[45] G. Banks M. Cova, V. Felmetsger and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 269–278, Dec 2006.

[46] E. Kirda M. Egele, T. Scholte and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[47] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[48] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[49] Memset, Linux MAN Pages. `http://linux.die.net/man/3/memset`. Accessed: 2014-09-07.

[50] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[51] Microsoft Visual Studio, Microsoft. `http://www.visualstudio.com`. Accessed: 2014-09-07.

[52] ASP.NET MVC. `http://www.asp.net/mvc`. Accessed: 2014-09-28.

[53] NHibernate. `http://nhforge.org`. Accessed: 2014-09-07.

[54] D. North. Introducing bdd. *Better Software magazine*, 2006.

[55] NUnit.org NUnit. `http://www.nunit.org`. Accessed: 2014-09-07.

[56] ObjDump, Linux MAN Pages. `http://linux.die.net/man/1/objdump`. Accessed: 2014-09-07.

[57] Oracle Open World: Build Your Own Type System for Fun and Profit. `https://checker-framework.googlecode.com/hg-history/7c7413415ae92310c74039943f2a2efab8c537d5/demos/2012-JavaOne/FunAndProfit.pdf`. Accessed: 2014-09-07.

[58] A. Biboudis P. Gerakios and Y. Smaragdakis. Reified type parameters using java annotations. In *Proceedings of the 12th International Conference on Generative Programming: Concepts &#38; Experiences*, GPCE '13, pages 61–64, New York, NY, USA, 2013. ACM.

[59] S. Sezer P. O'Kane and K. McLaughlin. Obfuscation: The hidden malware. *Security Privacy, IEEE*, 9(5):41–47, Sept 2011.

[60] Factory Pattern. `http://www.oodesign.com/factory-pattern.html`. Accessed: 2014-09-07.

[61] S. Rawat and L. Mounier. Finding buffer overflow inducing loops in binary executables. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 177–186, June 2012.

[62] re motion. `https://www.re-motion.org/web/`. Accessed: 2014-09-07.

[63] JetBrains Resharper. `http://www.jetbrains.com/resharper/`. Accessed: 2014-09-07.

[64] Hibernating Rhinos LTD RhinoMocks. `http://www.hibernatingrhinos.com/oss/rhino-mocks`. Accessed: 2014-09-07.

[65] A. Chaturvedi S. Bhatkar and R. Sekar. Dataflow anomaly detection. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15 pp.–62, May 2006.

[66] P. Kieseberg M. Huber M. Leithner M. Mulazzani S. Schrittwieser, S. Katzenbeisser and E. Weippl. Covert computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 529–534, New York, NY, USA, 2013. ACM.

[67] H. Schwichtenberg. *Windows Scripting: automatisierte Systemadministration mit dem Windows Script Host und der Windows PowerShell.* Pearson Deutschland GmbH, 2010.

[68] M. M. Seeger. Using control-flow techniques in a security context: A survey on common prototypes and their common weakness. In *Network Computing and Information Security (NCIS), 2011 International Conference on*, volume 2, pages 133–137, May 2011.

[69] Y. Solodkyy and J.Järvi. Extending type systems in a library: Type-safe xml processing in c++. *Sci. Comput. Program.*, 76(4):290–306, April 2011.

[70] J. S. Foster U. Shankar, K. Talwar and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[71] M. D. Ernst K. Muşlu W. Dietl, S. Dietzel and T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.