MSc thesis

Borsik Gábor Bence

2019

Inspection the use of static code analysis to automatically detect security issues

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS

Author:

Borsik Gábor Bence

MSc in Computer Science

2. year

Supervisor:

Gera Zoltán

assistant professor

Budapest, 2019

Contents

Acknowledgments

I would first like to thank my thesis advisor Zoltán Gera of the Faculty of Informatics at ELTE for his support in my research project, his patience, motivation and insight to the field. I would also like to thank Zoltán Porkoláb and Dániel Krupp, my colleagues, who helped me in the time of this research with they approach to the topic, along with my colleague Krisóf Umann, for offering their counterarguments and questions during my work.

My sincere thanks go to Ericsson Hungary, Ltd. for the opportunity for an internship, and access to their facilities.

Abstract

Security is one of the most important non-functional requirement of an application. Computers surround us everywhere, and our life depends on them in many ways. Most of these devices contain potential vulnerabilities as a result of poor programming. Taint analysis is a technique which can catch potential security leaks with static code analysis. It checks uses of data from external which data may have any value in a specific domain. However, a lot of function expect values from the subset of the domain then, so they expects sanitized data. The Clang Static Analyzer has a checker which can perform taint analysis on C or C++ code. Our goal is to refine the checker's architecture and increase its efficiency. In order to achieve this, we need to make the checker configurable. Hence, the user can set his own taint sources, propagation rules, sanitizers, and sinks. In addition, we added an aggressive propagation mode to the checker, where all unknown functions behave as a taint propagator. This greatly increases the checker's hit rate, while it only increases the false positives rate by a little. Another key point to remember is that we added support for some C++ language features and built-in types.

Absztrakt

A biztonság az egyik legfontosabb nem funkcionális követelmény egy alkalmazás számára. Az életünk sok szempontból függ a minket körülvevő számítógépektől. A legtöbb eszköz potenciális biztonsági réseket tartalmaz a nem megfelelő programozás miatt. A *taint* analízis egy olyan technika, ami statikus kódanalízis használatával képes felismerni potenciális biztonsági hibákat. Az analízis feladata olyan – külső forrásból származó – adatok ellenőrzése, amelyek bármilyen értéket felvehetnek egy bizonyos doménből, azonban sok függvény ennek a doménnek csak egy részéből várja az adatokat. A Clang Static Analyzer tartalmaz egy *checker*-t, amely *taint* analízist tud végrehajtani C vagy C++ kódon. A célunk, hogy javítsuk a *checker* belső architektúráját és növeljük annak hatékonyságát. Ennek elérése érdekében konfigurálhatóvá kell tennünk a *checker*-t, így a felhasználó be tudja állítani a saját *taint* forrásait, terjesztőit, tisztítóit és nyelőit. Ezen kívül hozzáadtunk egy agresszív *taint* terjesztési lehetőséget, ahol az összes ismeretlen függvény *taint* terjesztőként viselkedik. Ez nagymértékben növeli a *checker* hatékonyságát, viszont egy kissé nő a hibás találatok aránya is.

Chapter 1

Introduction

1.1 Motivation

The CodeChecker is an open source tool which can identify potentially wrong code constructions at an early stage of the development with static code analysis. It increases the software development speed, therefore, it makes the software cheaper and produces better quality code. The CodeChecker is built on the Clang Static Analyzer (Clang SA) which is one of the most advanced static analyzers. It can detect potential bugs in C and C++ source code using symbolic execution. The Clang SA evolves with the support of Apple, Google, Sony, and Ericsson.[1]

My thesis's purpose is to compare various CERT(Computer Emergency Response Team) organization's suggestions with the static analyzer's ability and to determine which inspections are algorithmic and automated with high reliability.

The security-related issues could have a much bigger impact than other bugs. Usually, a bug can cause bad behavior or crash, but for security issues the best case is when the system crashes. Otherwise, a hacker can steal passwords or other sensitive data, and run malicious code on our system or on our client's system. There are many known attacks for instance XSS, SQL injection, and buffer overflow.[2]

The thesis's main direction is to check the inadequate uses of user inputs. When the program reads some data from an untrustworthy source (standard input, file, or socket) it will be tainted. Those data could be literally anything. A division should not be evaluated with an unknown integer, because it could be zero which causes undefined behavior in C++. An array should not index with a tainted value either. In modern systems, the 32 bit signed integer's maximum value is 2147483647 which is almost always bigger than an array's size. There are a number of problems with strings from an external source. Its size and content are not known. The not sanitized string could cause several problems, because the user can run his own commands in our system.

1.2 Results

To deal with this issue, I described an internal representation for a checker working on taint analysis, which contains source, propagator, sanitizer and sink functions in a configurable way. Moreover, I outlined how to handle C++ language features, for instance, reference, extraction operator, assignment operator, namespaces, and member functions. To support input I modeled built-in classes such as std::string, and std::istream.

I implemented this as the part of the Clang Static Analyzer - an open source code analysis tool built on LLVM - which already has an existing implementation. I refined the internal architecture, implemented configuration, and added support for C++.

I tested my implementation with CodeChecker - an open source static analysis infrastructure built on LLVM/Clang Static Analyzer toolchain - which can help store and view defects. I performed analysis on several projects and I found four defects with taint analysis in *curl*.[3] One of them was a false positive, and three of them was true positive. Unfortunately, these defects are in the test code, but the results were important despite this because these defects cannot be found by the original implementation. I summarize one of the defects:

```
char* ptr;
FILE *stream;
stream = fopen(filename, "rb"); // stream is marked as tainted
char *cmd = NULL;
int error = getpart(&cmd, &cmdsize, "reply", "servercmd", stream); //
    cmd and cmdsize are marked as tainted
```

Source code 1.1: Defect's summary 1

The first tainted symbol is the *stream* pointer, because *fopen* is a taint source. Next, the analyzer parse the *getpart* function. Its definition is unknown, however, the checker considers it as a taint propagator, which is one of the new features. Consequently, the previous version lost the taint here. The checker's assumption is correct, the function reads data from the stream and writes it to the *cmd* buffer.

```c
int rtp_size = 0;
ptr = cmd; // ptr is tainted
if (3 == sscanf(ptr, "rtp: part %d channel %d size %d",
                &rtp_partno, &rtp_channel, &rtp_size)) {
  rtp_scratch = malloc(rtp_size + 4 + RTP_DATA_SIZE); // Untrusted data
      is used to specify the buffer size
}
```

Source code 1.2: Defect's summary 2

The *sscanf* is an unknown function, and it is not in the checker's built-in propagation rules list, but it is correct to mark *rtp_size* as tainted. In conclusion, the allocated amounts of memory depends on the file's content without any sanitization.

Nonetheless, it is not perfect, since the aggressive propagation increases the rate of false positives. Furthermore, C++ support is not complete, but it may offer a good start for further investigation and development.

Chapter 2

Basics

2.1 Static code analysis

Static program analysis is performed without executing the program, but analyzing the source code. Programmers make mistakes all the time, however, most of them are caught by the compiler. The longer a bug lies in the code, the more expensive it can be to fix, and more likely to cause financial or data loss. It is very important to find bugs at the earliest phase of software development. There are many common coding problems, which can be identified by a static analyzer tool. On the other hand, it is not as good as a manual review, but it is much cheaper and faster. It can improve the review procedure very well.

Static analysis cannot find all bugs in the code. It looks for a fixed set of patterns, or rules. Static analyzers will not fix the problem, they just emit some suspicious pattern. They are not perfect. Moreover, it is programmers' responsibility to decide whether it found a real bug or just a false positive (the tool reports bugs that do not exist). It is aimed to preserve the balance between false negatives (the program contains bugs that the tool do not report) and false positives because programmers will stop using the tool if it generates too many false positives.[4]

There are many security-related checks already implemented in the static analyzer tools, which are essential for safety critical application. For instance, Clang-Tidy contains several CERT checks, in particular, cert-msc51-cpp that was implemented by me. It ensures that the user seeds his random generator properly.

```
int main(int argc, char *argv[]) {
  std::mt19937 engine1; // Diagnose, always generate the same sequence
  std::mt19937 engine2(1); // Diagnose
  engine1.seed(); // Diagnose
  engine2.seed(1); // Diagnose

  std::time_t t;
  engine1.seed(std::time(&t)); // Diagnose, system time might be
      controlled by user

  int x = atoi(argv[1]);
  std::mt19937 engine3(x);  // Will not warn
}
```

Source code 2.1: Properly seeded random generators

2.1.1 Taint analysis

External sources (from the user, socket, shared memory, command line) return values where we cannot assume any limitation of the given values. That value is called tainted and its origin a tainted source. Certain functions have preconditions for their actual parameters. Violations of those preconditions can cause undefined behavior or crash.

Functions and operations that have preconditions for their actual parameters are called sinks. The standard library and the C/C++ language contain many sinks, for instance, subscript operator or modulo operator. Moreover, there are many functions which expect a null-terminated string. However, in many cases, the string's content is important too. For instance SQL injection, system calls or XSS are the most common cases.

Furthermore, tainted values are propagated through either functions or operations. Most of the arithmetic operations propagate taintedness. On the other hand, functions do not always behave as propagators. With this in mind, if the function's behavior is unknown, the propagation cannot be modeled precisely.

In order to make the taint analysis usable, there must be a way to remove the taint from a value. Functions which ensure the value meets the preconditions are called sanitizers.

They can sanitize the value in several ways, for instance, change the value, or terminate the program.[5]

2.2 Existing implementations

2.2.1 Clang Static Analyzer

The Clang Static Analyzer (Clang SA) is built on the LLVM infrastructure and is part of the Clang C/C++ compiler. The analyzer's core performs symbolic executions of the given program. It represents unknown input values as symbolic values and deduces all expressions in the program. The execution is path sensitive, hence, every possible path will be explored.[6, 7]

The Clang SA has a checker which is responsible for the taint analysis (GenericTaintChecker). Currently, it supports only C functions, but it works on C++ code as well. It has some common predefined taint sources, for instance, scanf, socket, getch, and fopen. It implements taint propagation in two ways. Firstly taint can be propagated through an expression. If a value is tainted in the expression, then the whole expression will become tainted. Secondly, the checker contains many common predefined functions which propagate taintedness through their parameters and return value. The checker also contains predetermined sinks:

- Uncontrolled format string: untrusted data is used as a format string (CWE-134)[8]
- Tainted buffer size: untrusted data is used to specify the buffer size (CERT/STR31-C)[9]
- System call: untrusted data is passed to a system call (CERT/STR02-C)[10]
- Array out of bound: untrusted data is used to index an array
- VLA size checker: has tainted size
- Division by zero: division by a tainted value, possibly zero

Despite this, it has many limitations. The user is unable to configure and add their own known taint sources or propagators. Furthermore, there are no sanitizer functions, because there is no way to define that. Additionally, today the applications are developed in C++ rather than C. This deficiency significantly reduces its usability.

2.2.2 Custom Taint Checker

CustomTaintChecker is a Clang Static Analyzer plugin, which can be loaded into the analyzer. The project had been forked from the GenericTaintChecker, unfortunately it was not committed into the Clang SA. Not only it has all the features that the original has, but also it is configurable and can handle sanitizer functions. However, it has limited support for C++, for instance the configuration does not handle namespaces or member functions. What's more, it can not work with common C++ I/O functions, for instance, *std::cin*.[11, 12]

2.2.3 Facebook Infer

Infer is an open source static analysis tool developed by Facebook. It has an experimental checker named Quandary, which performs static taint analysis. It has a small list of built-in sources and sinks, and can be used for Java. Sources, sinks, and sanitizers are configurable by the user. In comparison to the Clang SA based checkers, it does not have an ability to propagate taints through functions. In addition, the lack of built-in C/C++ sources and sinks makes it difficult to use on existing projects.[13]

Chapter 3

Outline of the solution

3.1 Internal working of the checker

In order to implement taint detection successfully, the limitations of the framework must be known. Clang Static Analyzer works on a compiling unit. Firstly, the analyzer core creates a call graph, then it starts the analysis at the top of it. Secondly, the checker is called, when the analyzer hits a function call. If the function's name matches with the predefined list, then it will mark the return value or the output parameter(s) as tainted.

The analyzer framework can track back the operations performed on a specific variable. To give an illustration, it can tell us when that variable was created by multiplying another variable with an integral constant. Therefore, the tainted flag can be tracked down. In conclusion, when a value is created by simple operations, the taint can be propagated through them.

```c
void foo(int n) {
  int x;
  scanf("%d", &x); // x is tainted
  int y = x + 5; // y consist of a binary operation between x and a
      constans
  int z = y * n; // z consist of a binary operation between y and n
  // z is tainted, if any of its ascendants are tainted
  // x can be tracked back from z
}
```

Source code 3.1: Taint propagation with symbols

Although it works only with operators, functions could be called with taint values. If the function is defined in the same compilation unit, then it will be inlined. Accordingly, those functions will work as there was not any function call from the taint value's point of view.

```
void func(int*);

void bar(int* x) {
  scanf("%d", x);
}

void foo(int n) {
  int x;
  bar(&x); // x is tainted, because the analyzer know bar's definition
  func(&x); // x is untainted, because the analyzer cannot assume
      anything about func
}
```

Source code 3.2: Taint propagation with functions

If the function is not defined in the same compilation unit, then the taint propagation will depend on the built-in functions of the checker. Obviously, only common functions could be defined there. Otherwise, the function is unknown, so the checker should handle it as propagator of taintedness, if any of its parameters is tainted. Importantly, this is a potential source of false positives. On the other hand, it would help to find as many true positives as possible. Above all, the correct solution is to make this behavior configurable.

In summary, the framework offer the following features:

- It works in one compilation unit
- The checker is called on every function call

- Store the operations which are performed on symbolic values

3.2 Internal architecture

The checker works with function names and classifies them to four groups:

- **Sources**: mark their return value or output parameter(s) unconditionally tainted
- **Propagators**: mark their return value or output parameter(s) if at least one of its input parameters are tainted
- **Sanitizers**: remove the taint from the specified arguments
- **Sinks**: emit bug report if the given argument is tainted

```c
void foo() {
  int x;
  // Reading from user returns tainted value
  scanf("%d", &x); // x is tainted

  // Unknown function propagate taintedness
  int z = func(x); // z is tainted

  // Filters remove taintedness
  myFilter(&x);

  // Sinks emit warning, if it get tainted value
  mySink(x); // No warning
  mySink(z); // Warning
}
```

Source code 3.3: The checker's expected behavior

The checker contains the most common functions in a predefined list to improve its efficiency. Unfortunately, sanitizer functions cannot be defined, because they always depend on the current environment. Consequently, the configuration of the checker is an indispensable feature. Without configuration, the chance of a true positive is inversely proportional with the project size, as well as the number of the used third-party libraries. For instance, if a third-party library is used to handle I/O, taintedness will never be initiated.

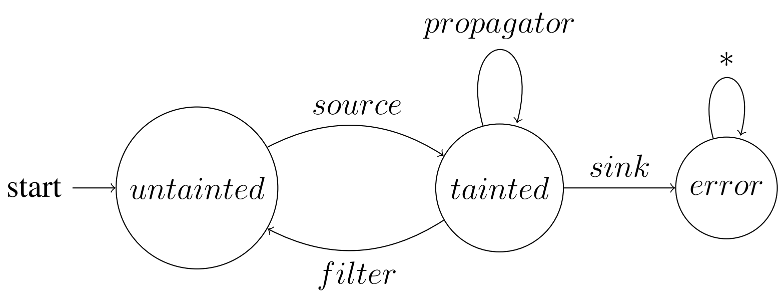

Figure 3.1: State transition system for taint analysis[11]

3.3 Support of C++ language features

C++ is one of the most popular programming languages in the world, therefore it is very important to support its features. This significantly increases the checkers' complexity due to the references, namespaces, templates, and objects.

Currently, the checker compares the function's name with a list of names. Use of namespaces in the function's name greatly decrease the number of false positives caused by name collision. For instance, there is a C library function *read*, which name is commonly used.

```
struct Foo {
  ssize_t read(int, void*, size_t);
};

namespace bar {
  ssize_t read(int, void*, size_t);
}

void func() {
  int fd; // Tainted file descriptor
  constexpr size_t size = 128;
  char buffer[size];

  Foo foo;
  foo.read(fd, buffer, size); // No match

  bar::read(fd, buffer, size); // No match

  // Posix read function
  read(fd, buffer, size); // Match -> buffer will become tainted
}
```

Source code 3.4: Functioning of scopes

The templates and inheritance make this model more complicated. The user has to be able to configure functions for all instantiations or just specific ones. However, normal strings and pattern matching does not offer flexibility for it. By contrast, regular expressions are much more appropriate for this use case.

The object-oriented programming paradigm is widely used in C++, therefore, taint propagation through object should be available. Even if the analyzer knows the structure of the object, it should bind the taintedness to the whole object instead of its fields. Usually, the whole implementation is not available for the analyzer. Consequently, the checker should adapt to the implementation details, which is not a scalable solution. In conclusion, the optional way is to treat objects like a black box. For objects, the taintedness should originate from assignments, constructors, member functions or free functions. Assignments always have to propagate taintedness. On the other hand, constructors and functions should be configurable.

To sum up, there are three essential features for a taint checker:

- Namespaces and member functions for configuration
- Work with templates and inheritance
- Tainted *this*

3.4 Support of C++ I/O

In order to model taint propagation properly, the taint sources should be defined very carefully. C++ greatly increases the possible ways to handle I/O, in particular, streams and stream buffers.

To begin, the most straightforward way to read data from the user is the *std::cin* (or *std::wcin*). This is a global object of class *std::istream* which controls input. Under the hood, it is associated with the C input stream *stdin*. These objects have to be marked as tainted from the beginning of the program.

Notably, uses of *std::cin* adduce the problem of C++'s overloaded operators. The most common way to read formatted data from the standard input is the extraction operator. It is fundamental to consider extraction operator in the same way as functions, which means if the object is tainted, the returned value will become tainted. Moreover, it has to work with all types of objects.

```
void foo() {
  int x;
  std::cin >> x; // x is tainted

  char title[256];
  std::cin.getline(title, 256); // title is tainted
}
```

Source code 3.5: Read from *std::cin*

Subsequently, one can gather unformatted input from *std::basic_istream objects*. Those functions should be handled as propagators. Furthermore, it does not have to depend on the template parameter and it has to work on its derived classes.

Then, the data can be read from files. C++ provide *std::basic_ifstream* for it. Fortunately, it is derived from *std::basic_istream*, so the reading operations are already solved. Another key thing to remember, all objects of this type has to be tainted after construction.

```
void foo() {
  int x;
  std::ifstream file("example.txt");
  file >> x; // y is tainted
  std::string str;
  file >> str; // str is tainted
}
```

Source code 3.6: Read from *std::ifstream*

Next, *std::basic_istringstream* is slightly different. It gets a string and considers it as a stream. As a result, it behaves as a taint propagator instead of a source. Hence, its constructor and *str* method should mark the objects as tainted. The propagation is solved by the inherited functions.

```
void foo() {
  int x, y;
  std::string str1;
  std::cin >> str1;
  std::string str2{"123 Sample string."};
  std::istringstream is{str2};
  iss >> x; // x is not tainted
  iss.str(str1); // str1 is tainted, so iss become tainted
  iss >> y; // y is tainted
}
```

Source code 3.7: Use of *std::istringstream*

Finally, I/O can happen through iterators. The *std::istream_iterator*'s constructor expects a *std::basic_istream* object. If the parameter is tainted, the iterator should become tainted. Accordingly, its dereference operator produce a tainted value.

```
void foo() {
  std::istream_iterator<std::string> iit(std::cin);
  std::string str = *iit; // str is tainted
}
```

Source code 3.8: Use of *std::istream_iterator*

All things considered, the checker has to support various C++ I/O features:

- *std::cin* should be tainted
- Extraction operator should propagate taint
- *std::(i)fstream* should be tainted
- *std::(i)stringstream* should propagate taint
- *std::istream_iterator* should propagate taint

Chapter 4

Implementation

4.1 GenericTaintChecker

In Clang SA there is an existing, built-in checker for taint analysis called GenericTaintChecker. It is responsible to initiate and propagate taintedness and it also emits a warning in specific cases. Besides, other checkers use taintedness, for instance, DivideZero, VLASize, and ArrayBoundV2.

```c
void foo() {
  int x;
  scanf("%d", x);

  int y = 1/x; // Division by a tainted value, possibly zero
  int buffer[x]; // Declared variable-length array (VLA) has tainted
      size
  int buf[10];
  buf[x] = 1; // Out of bound memory access (index is tainted)
}
```

Source code 4.1: Suspicious patterns I

The checker is called every time when the analyzer processes a function call. Firstly, it checks the function against the built-in list of suspicious patterns:

```c
void foo() {
  char s[80];
  fscanf(stdin, "%s", s);
  char buf[128];
  sprintf(buf,s); // Uncontrolled format string

  char addr[128];
  scanf("%s", addr);
  system(addr); // Untrusted data is passed to a system call

  size_t ts;
  scanf("%zd", &ts);
  int *buf1 = (int*)malloc(ts*sizeof(int)); // Untrusted data is used
      to specify the buffer size
}
```

Source code 4.2: Suspicious patterns II

Next, the checker tries to propagate taint through the predefined functions. It contains many taint propagation rules associated with the function's names. These rules describe if one of the specified argument is tainted, and in such cases it will mark other arguments as tainted. Finally, the checker tries to initiate taint. There are a bunch of functions which always return a tainted value, in particular, *scanf* and *socket*.

```c
void foo() {
  char buffer[100];
  int sock = socket(AF_INET, SOCK_STREAM, 0); // sock is become tainted
  read(sock, buffer, 100); // Because sock is tainted, buffer will be
      tainted
}
```

Source code 4.3: Propagation

4.2 Revision of the checker

Firstly, I simplified the taint propagation rules and made it more expressive. There wasn't a clean way to describe variadic functions, however, a lot of C library I/O functions are variadic. After that, I could fix propagation rules that were not correct, resulting from the shortcomings of the implementation. The final representation of taint propagation rules are the following:

- **SrcArgs** (source arguments): A list of indexes. If one of the actual parameters are tainted, the expression is tainted.
- **DstArgs** (destination arguments): A list of indexes. Those arguments, which will get the taintedness, if the expression is tainted.
- **VarType** (variadic type): An enum with three element:
  - **None**: Default value, do nothing.
  - **Src**: The variadic arguments act as taint source. If any of it tainted, the expression is tainted.
  - **Dst**: The variadic arguments act as taint destination. If the expression is tainted, the variadic arguments will get taint.
- **VarIndex** (variadic index): The index of the first variadic argument, if there is any.

I did another simplification on the implementation. The taint initiation used to be a separated step, in turn, it was possible to do it in the same step as the propagation. As a result, I consider the source functions as if they propagate from nothing.

My following patches are already merged into the analyzer:

- I revised GenericTaintChecker's internal representation.[14]
- I fixed taint propagation for source functions.[15]
- I considered source functions as propagate from nothing.[16]

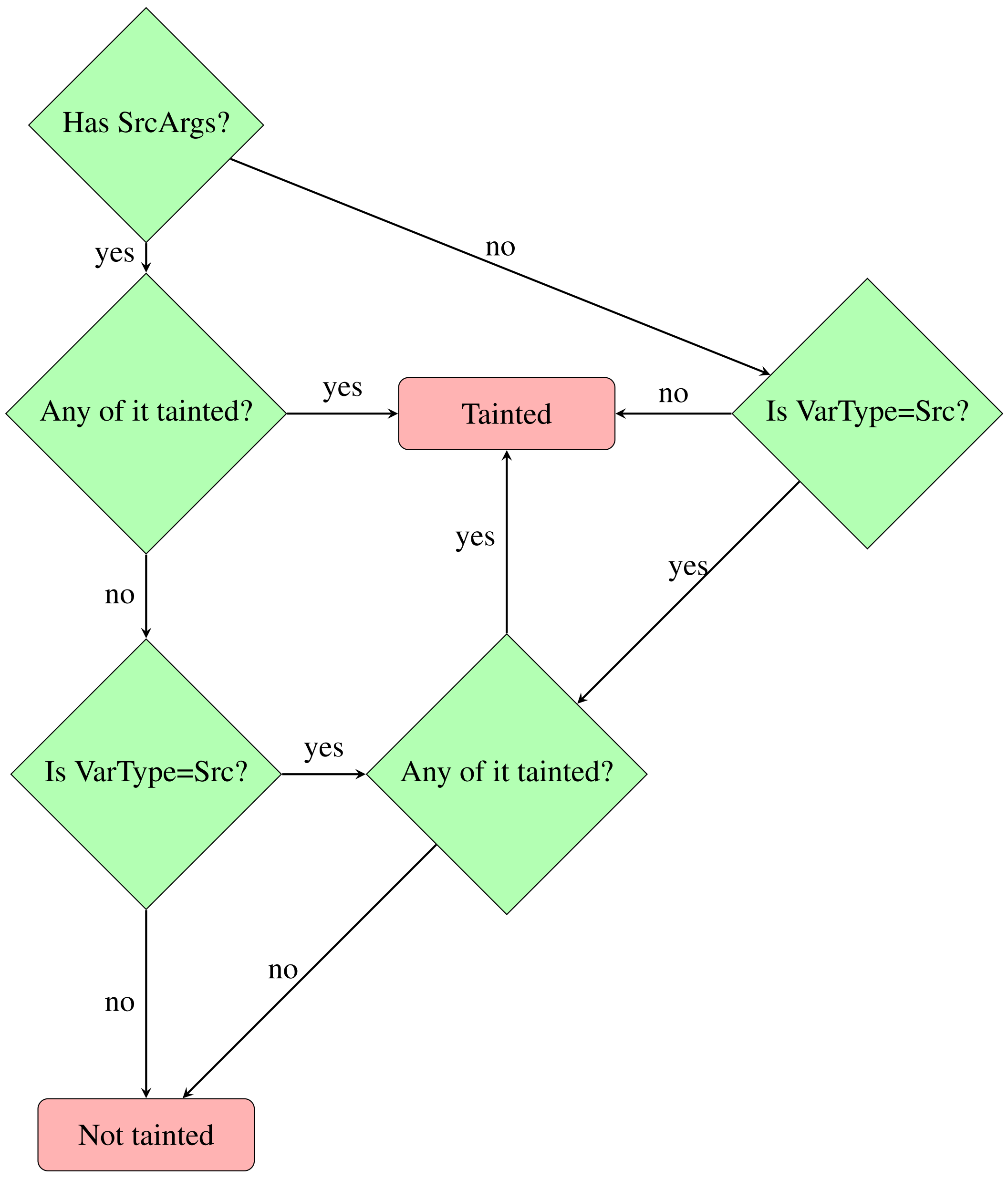

Figure 4.1: Taint propagation

4.3 Configuration

The configuration is a totally new feature of the checker, moreover, there was not any similar in the Clang SA to this either. I chose yaml as the configuration file format because it is a human-friendly data serialization language. Furthermore, the LLVM has an API for parsing yaml files. The checker gets the file's name as a command line parameter.

The file has four optional fields. Firstly, the checker's aggressiveness can be chosen. By default, it is not aggressive, because the Clang SA's policy is to reduce the number of false positives as much as possible. When it is set to false the taint propagation consider all unknown function as they do not propagate taintedness. In aggressive mode when the analyzer hits such a function it will propagate taint to all possible arguments if any of its argument is tainted.[17]

Secondly, the propagation rules are entries in a dictionary. Each rule is identified by the function's name. The scope, source and destination arguments, the variadic type, and index can be configured. The scope is a regular expression which is matched against the function's full name (with namespaces). If the checker runs on non-aggressive mode, then the propagation rules will increase the chance to find a true positive. Otherwise, the propagation rules will decrease the number of false positives. There are cases when the user uses a third party library to manage I/O for the application. At this time the configuration is indispensable because the analyzer is unable to find any bug without taint sources.[18]

As a result of the configuration, sanitizer functions could be introduced thus the user can define functions which remove the taintedness. With this feature, the false positives can be removed easily it without any tooling. To implement this I had to revise the taintedness representation. It was represented as an unsigned integer. I added an extremal value to represent the lack of taintedness. Therefore, if the tainted value is not present or it is zero the value is not tainted, otherwise it is. As a result, sanitizer functions can be considered as propagators which set values to not tainted.[19]

Finally, a custom sink can be configured. In security critical application sometimes there are functions which expect sanitized data. For instance, it can be a third party library where the implementation is unknown, therefore, the checker will not recognize the potential security issue. It can greatly complement the built-in patterns.[18]

```yaml
Aggressive: false

Propagations:
  - Name:     mySource
    DstArgs:  [-1, 0] # Index for return value
  - Name:     myPropagator
    SrcArgs:  [0]
    DstArgs:   [1]
  - Name:      myScanf
    Scope:     "myNamespace::"
    VarType:   Dst
    VarIndex:  1

Filters:
  - Name:   myFilter
    Scope:  "Foo::"
    Args:   [0]

Sinks:
  - Name:   mySink
    Scope:  "myNamespace::Bar::"
    Args:   [0, 2]
```

Source code 4.4: Example configuration

There are some other technical details for the configuration:

- The return value is represented by -1
- The arguments are numbered from 0
- Filters and sinks accept multiple arguments

As a result, users can considerably can reduce the number of false positive and increase the number of true positives. The yaml format offers a clear, human-readable configuration file without being too verbose. In order to model taint propagation properly, the taint sources should be defined very carefully. C++ greatly increases the possible ways to handle I/O, in particular, streams and stream buffers, therefore, it is important for the configuration to support the namespaces.

```
void foo() {
  int x, y;
  x = mySource(&y); // x and y is tainted

  int z = myPropagator(x); // z is tainted

  myFilter(y); // y is no longer tainted

  mySource(y, 1, 2); // No warning
  mySource(z, 1, 2); // Warning
}
```

Source code 4.5: The impact of configuration

I did the following patches related to the configuration:

- I added a yaml parser to GenericTaintChecker.[17]
- I implemented the uses of custom source, propagation and sink functions.[18]
- I implemented the filtering functions.[19]

4.4 C++ support

4.4.1 Language features

The original taint checker was mostly built on C. In this particular case, this means taint can propagate through arithmetic operations and functions where the parameters are pointers. Moreover, the structs do not contain any method. Therefore, the analyzer can trace the taint's path easily. By contrast, in C++ the classes usually have private members and the fields can be manipulated through member functions. Consequently, the implementation is not always known.

Firstly, I added support for references. The users prefer them over pointers, because they cannot be null, so their support is essential. Without it, the configuration will not work properly, because parameter pass by reference was simply ignored.

Then, I completed the configuration with a new field called *Scope*. The *Scope* is a prefix for the function's full name, for instance *myNamespace::myClass::*. It helps to refine the checker's accuracy. The configuration is stored in a map. The function's name is the key, the scope and the other data are the value. First, it gets the value via the function's name for each map. If it is not present, it continues to another check. Second, if the scope is present, it will be compared with the start of the function's full name. Above all, a function is a match with a configuration entry, if the name is equal and the scope is not present or matched.

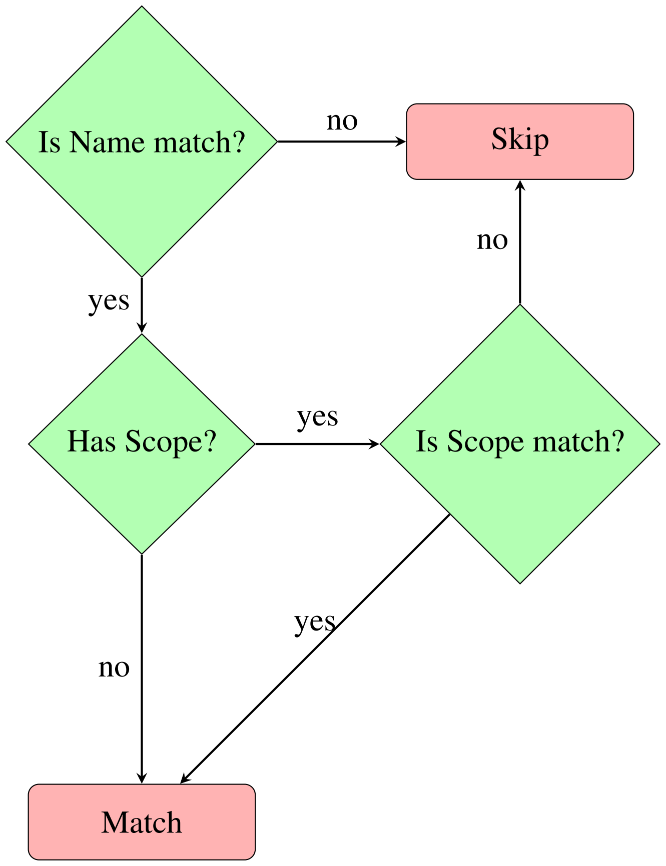

Figure 4.2: Function name matching

Next, the member functions have to be evaluated properly. The analyzer uses a different way for the implicit *this* and the other parameters. I had to create a new abstraction layer to handle them in the same way. The implicit parameter is the zeroth and the explicit parameters start at one if it is a member function.

The Clang SA handle the assignment in a different way for primitive and complex types. For objects, the assignment is a call for the overloaded operator. Therefore, I had to add an entry for overloaded operators and create a taint propagation rule for the assignment operator. This rule propagates taint to the first argument and the return value if the second argument is tainted.

The most common way to read from any stream is with the extraction operator. It is an overloaded operator such as the assignment operator, so it requires a custom propagation rule too. It propagates taint to the second argument and the return value if the first argument is tainted.

```cpp
struct Foo {
    int getInt() const; // Configured to propagate taint
};
std::istream& operator>>(std::istream&, Foo&);

void mySink(int&);

namespace myNamespace {
  void mySink(int&); // Configured to sink
}

void bar() {
  Foo foo;
  std::cin >> foo; // foo is tainted

  int x = foo.getInt(); // x is tainted
  mySink(x); // No warning
  myNamespace::mySink(x); // Warning
}
```

Source code 4.6: Supported C++ language features

4.4.2 Strings

There are several vulnerabilities due to the use of unsanitized strings, for instance, SQL injection, and XSS. What is more, they can be converted to integral types which can lead to other security issues. The *std::basic_string* class is a template and it has many instantiations. My goal is to support all of them. Strings (and other objects) can get taint with assignment or extraction operator.

The standard string's representation is not known, so it behaves like a black box. I hard coded the commonly used string operations: *c_str*, *data*, *size*, *length* and the non-member *getline*. These functions are enough for common use-cases.

```cpp
void mySink(const char*);

void foo() {
  std::string str1, str2;
  std::cin >> str1; // str1 is tainted
  std::getline(std::cin, str2); // str2 is tainted

  mySink(str1.c_str()); // Warning
  int buffer[10];
  buffer[str2.size()] = 1; // Warning
}
```

Source code 4.7: Strings

4.4.3 Streams

When one would like to read data in C++, usually a stream will be used. If a stream reads data from an external source, the data will be tainted. There is one exception, the *stringstream*, where the taintedness should depend on the actual parameters.

Firstly, I had to mark *std::cin* (and *std::wcin*) as tainted. When the analyzer checks if an actual parameter is tainted, it returns true if the object's name is *cin* (or *wcin*). It also must be in the standard namespace. The C *stdin* is recognized in the same way.

Next, the data can be read from files. Unfortunately, I got a problem when I was trying to mark *ifstream* objects as tainted. The *open* function is implemented in the header file, so I cannot model it. I chose a simple but efficient way to solve it. The checker considers all of the descendants of *std::basic_istream* as tainted. Therefore, the *std::cin* doesn't have to be handled separately. It works fine for files too, because a file stream cannot be sanitized only the content read from it.

Although it works properly most of the cases, but it has its own limitations. When a *stringstream* is used for I/O the output's taintedness depends on the input string. Accordingly, this solution can cause potential false positives.

```cpp
void myFilter(std::istream&); // Sanitizer functions

void foo() {
  std::string str1, str2, str3, str4;
  std::cin >> str1; // str1 is tainted
  myFilter(std::cin); // std::istream object cannot be sanitized

  std::ifstream file("example.txt");
  file >> str2; // str2 is tainted
  myFilter(file); // std::istream object cannot be sanitized
  file >> str3; // str3 is tainted

  std::string sample{"123 Sample string."};
  std::istringstream is{sample};
  is >> str4; // str4 is tainted (False positive)
}
```

Source code 4.8: Streams

Chapter 5

Summary

5.1 Future work

5.1.1 Commit to Clang Static Analyzer

As I started the work on an existing implementation it is obvious that I should commit my changes to the analyzer. Since the review, the review is a time-consuming process I have not committed all of my changes yet. The revision of the checker's internal implementation is part of the analyzer because that was a non-functional change. I have three open revisions about the configuration and many other changes in my local trunk.

5.1.2 Make the checker default

The GenericTaintChecker (which accomplishes taint analysis) is currently an experimental checker in the Clang SA, which means it is merely compiled but disabled by default. Our goal is to produce a reliable checker with a low false positive rate. To achieve this, I have to finalize its internal architecture, because it is much more difficult to commit changes into default checkers.

The checker offers a framework for taint analysis and the results are used to emit warnings. The framework is currently used by three other checkers: DivideZero, VLASize, ArrayBoundV2. The last one in an experimental checker, therefore, it is worth revising to move it to the default checkers. Moreover, new checkers should be implemented which rely on taint analysis.

5.1.3 C++ related features

The constructors are not supported yet, because the analyzer considers them differently than functions. They are essential for the taint propagation between objects. The assignment operator propagates taint, although it does not cover all the cases. Secondly, unknown objects can be modeled as taint propagators. If the constructor gets a tainted value, it should mark the whole object as tainted.

```cpp
void foo() {
  std::string str1, str2;
  std::cin >> str1; // str1 is tainted
  str2 = str1; // str2 is tainted
  std::string str3 = str1; // str3 is not tainted, because it is a copy
      constructor call

  const char* cstr = str3.c_str(); // cstr is tainted;
  std::string str4(cstr); // str4 is not tainted
}
```

Source code 5.1: Constructors

The *std::istream* objects are always implemented as tainted, which is not always true. My plan was to find as many defects as possible and then refine the model. Unfortunately, the C++ language features' coverage was not enough to find any defect, or I chose inadequate projects. Accordingly, it should be fine-tuned.

It is possible to read data through *std::istream_iterator*, but for the most common use case of the constructor's taint propagation should be supported. Besides that, all containers' *begin*, *end* (and their variant) have to propagate taint, as well as *std::begin*, and *std::end*. Not to mention, the overloaded dereference and arrow operator have to propagate taint.

The configuration can be complemented with function attributes. Then the checker will be configured without a configuration file. Although it has its own drawback it can work together with the yaml configuration very well.

5.1.4 Other miscellaneous features

The most straightforward source of a tainted value is the main function's parameters. The number of command line parameters are not limited, and their content is unknown. Therefore, they should be marked as taint, if they are presented.

During the tests, I ran my checker under the CodeChecker to analyze the projects with my checker. It has a bug track visitor, which can find the defect's origin. It works fine for the taint checking until the taint is propagated through anything other than a function. Unfortunately, the analyzer loses the path, if the taint originates from a function with an unknown definition. It does not increase the analyzer's performance, however, it greatly increases the user experience especially in big projects.

```cpp
void foo() {
  int x, y;
  std::cin >> x; // real origin of the taint

  propagator(x, y); // analyzer thinks the taint originated from here
  mySink(y); // y is tainted, warning
}
```

Source code 5.2: Bug track visitor

5.2 Conclusion

In this thesis, I optimized the model of the taint analysis for C/C++ programs, and implemented it in Clang Static Analyzer. Taint analysis is essential for security-critical applications because it can find vulnerabilities without executing the code.

After I tested the implementation against several projects and evaluated the defects, I made some conclusions. The original checker's notion was false since it made taint propagation with only compile-time known functions. In theory, it will cause very few false positives, but it works in just few cases, which are rare in large code bases. My greatest result is aggressive propagation, where unknown functions always propagate taint. This was indispensable to find the defects in *curl*, which prove this concept's viability.

The configuration is useful to define taint sources and to reduce the number of false positives. It can improve the quality of the analysis, however, it requires extra time from the developers.

The C++ support needs further investigation to achieve high enough coverage. Despite this, it will be one of the most important parts of the checker, because is C++ more common than C.

	Infer	Clang Static Analyzer		
		old	plugin	new
C sources	No	Yes		
C++ sources	No			Yes
Propagation with operators	Yes			
Propagation with functions	No	Yes		
Sinks	No	Yes		
Configure sources	Yes	No	Yes	
Configure propagators	No		Yes	
Configure sanitizers	No		Yes	
Configure sinks	Yes	No	Yes	
Namespaces	Yes	No		Yes
Member functions	No			Yes

Table 5.1: The comparison of analyzers' taint analysis

Finally, I compare the Facebook Infer, the CustomTaintChecker (the plugin, which was forked from Clang SA), the previous Clang SA, and my own version. My implementation contains every relevant feature of the others, moreover, it has C++ support and aggressive propagation as extra features.

Bibliography

[1] CodeChecker. https://github.com/Ericsson/codechecker. Accessed: 2019-05-03.

[2] G. McGraw. Software security. *IEEE Security & Privacy*, 2:80–83, 2004.

[3] curl. https://github.com/curl/curl. Accessed: 2019-05-09.

[4] G. McGraw B. Chess. Static analysis for security. *IEEE Security & Privacy*, 2:76–79, 2004.

[5] Taint Analysis. https://wiki.sei.cmu.edu/confluence/display/c/Taint+Analysis. Accessed: 2019-04-28.

[6] LLVM. https://llvm.org/. Accessed: 2019-05-01.

[7] Clang. https://clang.llvm.org/. Accessed: 2019-05-01.

[8] CWE-134. https://cwe.mitre.org/data/definitions/134.html. Accessed: 2019-05-09.

[9] CERT/STR31-C. https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator. Accessed: 2019-05-09.

[10] CERT/STR02-C. https://wiki.sei.cmu.edu/confluence/display/c/STR02-C.+Sanitize+data+passed+to+complex+subsystems. Accessed: 2019-05-09.

[11] F. Bavera M. Arroyo, F. Chiotta. A user configurable clang static analyzer taint checker. IEEE, 10 2016.

[12] Custom Taint Checker. `https://github.com/franchiotta/taintchecker`. Accessed: 2019-04-28.

[13] Facebook Infer. `https://fbinfer.com`. Accessed: 2019-04-28.

[14] Revise GenericTaintChecker's internal representation. `https://reviews.llvm.org/D55734`. Accessed: 2019-04-28.

[15] Fix taint propagation in GenericTaintChecker. `https://reviews.llvm.org/D58828`. Accessed: 2019-04-28.

[16] Prepare generic taint checker for new sources. `https://reviews.llvm.org/D59055`. Accessed: 2019-04-28.

[17] Add yaml parser to GenericTaintChecker. `https://reviews.llvm.org/D59555`. Accessed: 2019-04-28.

[18] Use the custom propagation rules and sinks in GenericTaintChecker. `https://reviews.llvm.org/D59637`. Accessed: 2019-04-28.

[19] Add custom filter functions for GenericTaintChecker. `https://reviews.llvm.org/D59516`. Accessed: 2019-04-28.

Figures

Tables

Source codes