Imperial College London

BEng Individual Project Report

Department of Computing

Imperial College of Science, Technology and Medicine

JsTainter: Dynamic Taint Analysis on JavaScript Program

Author:
Huanyao Rong

Supervisor:
Dr Sergio Maffeis

June 17, 2019

Submitted in partial fulfillment of the requirements for the Computing BEng of Imperial College London

Abstract

In this project, I have implemented a JavaScript dynamic taint analysis tool in JavaScript, called JsTainter. The analysis is based on Jalangi2, a dynamic instrumentation framework on JavaScript. In analysis codes, I have designed a structure that is used to describe the taint state of different types of JavaScript variables, including complex structures such as object and array. In addition, various operations of JavaScript are considered and their taint propagation rules have been implemented, such as binary operator and native function call. A recorder is also implemented to record information like taint flow of the JavaScript program as it executes, which is shown as the result of analysis. When the analysis is run in browser, sources and sinks are also handled.

Acknowledgments

Thanks for Dr Sergio Maffeis who has proposed such great project and provided many valuable instructions.
Thanks for Dr Anandha Gopalan who provides many useful comments on my Interim Report.
Thanks for all staff in Imperial College London who provided such abundant education resources.
Thanks for my family who provided support for me to attend Imperial College.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays the website becomes one of the most commonly used application, and JavaScript is also the most common front-end language, which provides various functionalities on the front-end. Along with the functionalities it provides, JavaScript also poses potential threats to users.

The threats can result from both web JavaScript codes itself (e.g DOM-based XSS) and web browser engine that execute the JavaScript (e.g use-after-free in browser engine). Both of them requires analysis of JavaScript codes: we need to audit the codes from website to find vulnerability from JavaScript; we also need to read the exploit of browser vulnerability to locate and fix the bug.

However, these are not easy work to do: the client-side web application contains complicated logic and symbols are usually stripped out, while we only want to focus on how inputs are processed since that is where the vulnerabilities come from; malicious developers may obfuscate their JavaScript exploit to prevent researchers from reversing it, but we want to focus on where the payloads are passed into JavaScript API and how vulnerability is triggered.

Since it is crucial to trace the user input in order to make our analysis more efficient, we can employ technique called JavaScript Taint Analysis. However, compared to similar works being done in native programs such as binary executables written in C, the research in JavaScript Taint Analysis is comparatively rudimentary. Therefore, something that performs taint analysis on JavaScript program is required.

1.2 Objective

My final objective is to develop a software that performs dynamic taint analysis on target JavaScript codes. Analyzer who uses this software to analyze JavaScript program is able to inspect how user input would be used and processed by the program, which can make analysis more efficient.

1.3 Challenges

JavaScript is a dynamically-type and weakly-typed language, so many edge cages about type must be considered in analysis. For example, number can be concatenated to string, which does not hold for strongly-typed language like Python. Therefore, when string concatenation is processed, we cannot consider the case when both operands are string only.
In addition, unlike binary program which is usually executed on relatively small instruction set, the instruction of JavaScript that need to be consider is much larger than binary assembly. For example, common functions in `String.prototype` should be considered in analysis, but in binary program, these functions will all be compiled to assembly instructions so taint analysis does not need to consider them.

1.4 Contribution

I have considered different approaches to implement a JavaScript taint analysis and their pros and cons. I have investigated different dynamic behaviors of JavaScript and taint propagation rules that should be used for these behaviors when they are being traced by dynamic taint analysis. I have also investigated different types of sources and sinks in modern browser that should be considered when browser-side JavaScript analysis is performed.

Chapter 2

Background

2.1 Overview of Taint Analysis

The taint analysis is a technique that is commonly used by computer science researchers to trace the effect of user input to the program. It is an automatic process that analyze the how input is distributed over the program. The most common usage is in the field of information security, especially automatic reverse engineering and vulnerability detection.

2.1.1 Vulnerability Detection

The taint analysis can be used to find vulnerabilities since every vulnerability is caused by improper processing of the user input. If the user input is not correctly handled, the vulnerability may arise: if the special characters of the user input are not filtered and converted to secure ones, the injection vulnerabilities may arise, such as SQL injection, XSS and command line injection; if the length of the input is not well considered, the overflow vulnerability may occur, such as stack overflow and heap overflow; if there problems within the logic of the program and some edge cases are ignored by developers, which can be triggered by special input, the vulnerability can also emerge, such as use-after-free and race condition. There are also many other kinds of vulnerabilities ranging from binary ones such as format string injection and uninitialized variable, to web ones such as arbitrary file.
With such various kinds of vulnerabilities, people have tried to find efficient way to discover them. However, this is a hard thing to do because the actual amount of codes in the real-world application are huge and the logic is complicated. In addition in most cases the source code of the application is not even accessible by security researcher: the source codes of server-side web applications from companies are usually not publicized except some of the open source library they have used; and the client-side applications are also not always open-source so security researchers may need to spend time on reverse engineering. Therefore, although indeed vulnerabilities can be found in this way effectively, it is still time-consuming and inefficient to analyze the potential vulnerable program manually.

Over the past decades people have spent much time in finding automatic and effective way to detect vulnerability. People came up with the idea of fuzzing, which is to randomly generate input that feeds into the program, and inspect which input may cause the program to crash. However, this might be ineffective because completely random input is not target specific enough. For example, when fuzzing file format parser, the first 4 bytes of file format is always a constant and we must satisfy it to pass the first check. However, if we generate the file in completely random way, it is almost impossible to pass the first check and will lead to very low efficiency. The modern approach to solve this problem is to combine symbolic execution and taint analysis with random input generation. In other words, the generation is not completely random but is generated according to the program to cover the maximum path exploration using constraint solving.

2.1.2 Reverse Engineering

Besides automatic vulnerability detection, taint analysis can also be used to do automatic reverse engineering. In fact, these two are not independent and are highly correlated to each other. As I said above, vulnerability detection may require reverse engineering if the target application is not open-source. Since when doing reverse engineering, we only want to focus on the information that we are interested in, taint analysis can help researchers to efficiently focus on the specific trace of the input propagation, ignoring other unhelpful codes and largely improve the efficiency.
In some cases, there might be large amount of unhelpful codes that reverse engineer people are not interested, which can be produced both intentionally and unintentionally. For the previous case, the developer may add intentional obfuscation to prevent people from reverse engineering. For the second case, some compiler may generate edge case checking and handling codes that reverse engineer people might not want to focus on, such as the executable generated by Rust compiler.
The taint analysis can also be used to reverse engineering wild 0-day sample specifically, since we need to trace the input from malicious data and figure out where the vulnerability is, taint analysis is a common way being applied to improve efficiency of analysis significantly.

2.2 Overview of JavaScript

JavaScript is a kind of script language that is commonly used in front-end of the web application. Since it is a highly dynamic script language, it is harder to analyze compared to native programs.

2.2.1 Dynamic Feature of JavaScript

The JavaScript is a kind of dynamic script language.
Firstly it is dynamically-typed, which means that the type of variable cannot be decided statically and can vary as the program runs, as shown in Listing 1.

```javascript
var a = 1;
// initially type of variable `a` is integer
a = "hello";
// JavaScript is dynamically-typed,
// so this would be fine
// and the type of variable `a` will become string
```

Listing 2.1: dynamically-typed JavaScript

By contrast, the language like C++ is statically-typed, as shown in Listing 2.

```cpp
auto a = 1;
// type of variable `a` is always integer
// this cannot be changed anymore
a = "hello";
// C++ is statically-typed,
// so this would generate compilation error
```

Listing 2.2: statically-typed C++

The JavaScript is a dynamically-typed language, so from the perspective of program analysis, the specific type of a variable cannot be statically determined. This means that less information is provided when we analyze the codes statically. The way to solve this problem is to collect dynamic type information when executing the program and use them for static analysis [14], which I will detail later.

Secondly, JavaScript allows *eval* expression, which regard the string parameter being passed to it as JavaScript code and dynamically execute it. For example:

```javascript
eval("var a = 1;");
// this is equivalent to `var a = 1;`
alert(a);
```

Listing 2.3: example of *eval*

This feature cause problem to static analysis because unlike my example above, when developers use *eval*, usually the string parameter passed into it is usually not a constant and cannot be decided statically. For example: [1]

```javascript
window.onload = () => {
  const form = document.getElementById("flag-form");

  function xor(key, str) {
    let out = "";
    for (let i = 0; i < str.length; i++) {
      out += String.fromCharCode(str.charCodeAt(i) ^ key[i %
          key.length]);
    }
    return out;
  }

  form.addEventListener("submit", event => {
    eval(xor([1,3,3,7],"gob`/ubktf>:<fufm+{hs+X33/3\x7f51^+#L7L
       /_{73c_\x7f1g$@3vf[y32$_.Km]{3c]{2?s_{69A#kzY\"[y2a|X q[
       y2;Ad_{72c_\x7f10!.("))
    || event.preventDefault();
  });
};
```

Listing 2.4: when parameter of *eval* expression is not a constant

In this example, the string passed into *eval* expression is dynamically generated by *xor* function, which uses first parameter as *xor* key to decode the string from second parameter and returns the result of decoding.
This is actually a common way that developer employs as an anti-reverse-engineering approach, and it is very usual for malware author to use such obfuscation technique. The way to solve this is also to collect dynamic information while executing the JavaScript program.[14] To be specific, we can hook each *eval* expression and record the specific string parameter passed into each *eval* expression, and use that information for static analysis.
Although in the second example the string is decoded first before passing into *eval*, the actual parameter passed into *eval* is always identical in both of the two examples above. In other words, the string parameter does not depend on other kinds of variant such as user input or result of network request. However, this is not the case, and this is where the problem usually arises. For example:[2]

```
eval('alert("Your query string was ' + unescape(document.
   location.search) + '");');
```

Listing 2.5: XSS caused by *eval* function

In this case, since *document.location.search* can be controlled by attacker who makes the URL, a XSS exists in this code.
Thirdly, JavaScript support Function Variadicity, which means the same function can be called with variant number of variables, for example, here is one example given by one paper[14] in Figure 4:

```
function a(){
    if(arguments.length===1){
        evt=null;
        L=arguments[0]
    }else{
        evt=arguments[0];
        L=arguments[1]
    }
    J(evt,L)
}
```

Listing 2.6: Function Variadicity if JavaScript

In this case, even if function *a* seems to be declared as function with no argument, it indeed can be called with any number of arguments. In other word, all of the function calls below are valid.

```
a()
//J(undefined, undefined) will be called
a(1)
//J(null, 1) will be called
a(2,3)
//J(2, 3) will be called
a(4,5,6)
//J(4, 5) will be called, while 6 will be ignored
a.apply(null, arr)
//cannot decide length of `arr` statically!
//so we don't know how function `a` will be called!
```

Listing 2.7: *a* can be called with any number of arguments

This cause problem for normal static analysis, especially for the last case: if the second argument is some array variable (in fact you even cannot decide this is an array variable statically!), we cannot decide the length of array thus cannot decide how many number of arguments of this call by merely static analysis.

2.3 Dynamic Taint Analysis

In this section I am going to discuss the methodology covered in [5]. In this paper the formal description and some other aspects of dynamic taint analysis and forward symbolic execution are discussed in detail using a language called *SIMPIL* and its operation semantics. *SIMPIL* is a simple language defined by author, which can be translated from other static languages like C and extended with new features. However, due to its various dynamic feature, JavaScript is harder to be transformed into this language compared to language like C. Nonetheless, we can still refer to the techniques of dynamic taint analysis and forward symbolic execution that are used to analyze this language, since the essences of these technique is actually same.

2.3.1 Discussion over Dynamic Taint Analysis

One of the most important part of this paper is that it discusses the formal algorithm of dynamic taint analysis and forward symbolic execution. In this sub-sub-section I will describe the algorithm in my own words and discuss some of its drawbacks. I will also relate it to JavaScript and point out the difference and similarity between *SIMPIL* language and JavaScript. Some of the paragraph is extracted and modified from my previous writing when preparing this project. [3]
The definition of this language is shown below[5].
In order to discuss the operation semantics, we need to define the context symbols first, as shown in *Figure 2*. [5] Then here is the operational sematics defined in the paper[5] as shown, I will discuss each one in detail in later.

T-INPUT evaluate input from *get_input* as tainted if $P_{\text{input}}(\text{src})$ returns true. In this paper P_{input} is always true but we can vary this and let P_{input} returns true only for

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*) | goto *exp* | assert *exp* | if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | get_input(*src*) | v
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Figure 2.1: definition of **SIMPIL**[5]

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Figure 2.2

taint t	::=	**T** | **F**
value	::=	$\langle v, t\rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

Figure 2.3: definition of SIMPIL contexts

some of the source that we want to trace. In JavaScript context, the sources like DOM object access, network request, and user input can be regarded as similar stuff, especially user input.
T-CONST evaluates the taint state of constant according to return value of P_{const}, which is usually true. In JavaScript context, any constants not limited to integer can be regarded as the similar thing, including string, array, and even object constants.
T-VAR evaluates any variable, and use the taint state of that variable as the result taint state of the expression. In the JavaScript context, since the variables are not all global unlike SIMPIL, we need to consider that 2 variables may have same name but they refer to different variables in different context. For example, in one function we declared a variable *i* but there is also a global variable named *i*. When using this name in this function, *i* refers to that local variable; but when using name outside the funtion, *i* refers to the global variable. It is one factor to consider when doing the dynamic taint analysis for JavaScript.

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\mathbf{input}}(\text{src})\rangle}\ \text{T-INPUT} \qquad \frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\mathbf{const}}()\rangle}\ \text{T-CONST}$$

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \mathit{var} \Downarrow \langle \Delta[\mathit{var}], \tau_\Delta[\mathit{var}]\rangle}\ \text{T-VAR} \qquad \frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\mathbf{mem}}(t, \tau_\mu[v])\rangle}\ \text{T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \lozenge_u e \Downarrow \langle \lozenge_u v, P_{\mathbf{unop}}(t)\rangle}\ \text{T-UNOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \lozenge_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow \langle v_1 \lozenge_b v_2, P_{\mathbf{binop}}(t_1, t_2)\rangle}\ \text{T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle \quad \Delta' = \Delta[\mathit{var} \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[\mathit{var} \leftarrow P_{\mathbf{assign}}(t)] \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \mathit{var} := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc+1, \iota}\ \text{T-ASSIGN}$$

$$\frac{\begin{array}{c}\iota = \Sigma[pc+1] \quad P_{\mathbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\mathbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc+1, \iota}\ \text{T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t\rangle \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc+1, \iota}\ \text{T-ASSERT}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-TCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota}\ \text{T-FCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t\rangle \quad P_{\mathbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-GOTO}$$

Figure 2.4

T-LOAD is the instruction to load the data from memory. This will cause *tainted address* problem. When doing JavaScript analysis, the story is a little bit different, because JavaScript is a script language without direct memory accessing, and its byte-codes generated by interpreter do not have direct memory accessing either[7]. Nonetheless, there is still array accessing in the JavaScript, and if the array index is tainted, we need to be careful about the taint state of the result of array access. In the paper, initially it is suggested to consider the taint state of the data in memory only, ignoring the taint state of address. In the context of JavaScript, this is saying we are not going to consider the taint state of the array index but only consider the taint state of array element inside the array, and the result of array access is tainted if and only if the element being accessed is tainted.
However, this cause problem when we use a ”map table” to convert the form of data. For example, in *base64* encode, we may need a utility like this

```
function map_to_base64_char(val)
{//pre:0<=val<64, and `val` is int
    var tab = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
    "abcdefghijklmnopqrstuvwxyz0123456789+/";
    //the string in JavaScript acts similarly to array
    return tab[val];
}
```

Since the a constant string is assigned to *tab*, every element in tab must also be untainted. According to the rule that I mentioned above, the *tab[val]* is also untainted, which is clearly not the case: *tab[val]* is *base64* form for *val*, so if *val* is tainted, *tab[val]* should also be tainted.

Then, the paper also suggested to also consider the address, the result is tainted as long as the address to access memory or the content in memory is tainted. In the context of JavaScript, this is saying are also considering the taint state of the index that is used to access the array. The result of array access is tainted as long as the index to access the array or the array element to be accessed is tainted. However, we can still let it produce wrong result. Consider this code

```
function return_A(some_tainted_data)
{//pre: `some_tainted_data` is `int`
    var a = 'A'.repeat(0x100);
    var still_tainted = some_tainted_data % 0x100;
    //binary operation inherit the taint state,
    //this is to be discussed later
    return a[still_tainted];
}
```

For this function, whatever the *some_tainted_data* is, the function always returns a constant 'A'. However, since the index is tainted, the *a[still_tainted]* is tainted, which is not what we want. Fortunately this kind of codes rarely occurs unless the developer has intentionally added some anti reverse engineering obfuscation codes.

Now I am going to describe the JavaScript context only. Another way to tackle this issue is to record various mapping tables in taint analysis engine, and regard it as a mapping "function" that will inherit taint state like unary operator(to be discussed later). When array access is executed, only if the array matches one of the tables recorded do we consider the index as a "taint contributor"; if not, we only consider the taint state of array element being accessed. For example, assume that we have already got some common mapping tables including *base64*, *digit conversion*, *alphabet conversion*, etc. table being recorded in the JavaScript dynamic taint analysis engine. When array access with tainted index is executed, we can use the common table recorded to judge if the we should consider the result as tainted. In the case of *map_to_base64_char*, we will find that the content of the array matches the one of the table we stored, thus marked the result as tainted; in the case of *return_A*, since there is no array like `'A'.repeat(0x100)` being stored in dynamic taint analysis engine, we do not consider it as tainted. The accuracy of this approach depends on the mapping tables recorded in taint analysis engine, which can be configured manually.

Back to semantics of SIMPIL.

T-UNOP and **T-BINOP** are operators, and they work in the same way in the context of JavaScript integer operation(in other word, obviously it is a bit different when we apply *plus* to string in JavaScript). In the paper, it is suggested to mark the result of operation as tainted as long as one of the operands are tainted, except some special case like var a = b xor b;. This might be non-problematic by doing some simple check against the operator and operands, but what about this case?

```
var tainted1; // integer
var tainted_cont1 = tainted1;
var tainted_cont2 = tainted1;
var wrongly_tainted = tainted_cont1 ^ tainted_cont2;
```

As shown above, the operands of *xor* seems to be different but actually they are same, and this is hard to identify using solely dynamic taint analysis unless we do static control flow analysis.
What if we check against value of the operands? For example, as long as the result of the operation is *0*, we mark the result as untainted. This turns out to be a terrible idea:

```
var tainted;
//...
var still_tainted = tainted + 0xdeadbeef;
var tainted_wrong = still_tainted - tainted;
```

This should return *0xdeadbeef* unless in the case where the overflow occurs and should have been marked as untainted. But since the result is not *0*, it will still be marked as tainted.
And there is also cases where the 2 input values are not necessarily always identical but we have them equal accidentally in some cases when this binary operation is executed. In this case we should not have simply classified it as untainted but the algorithm will. But, if we use forward symbolic execution, this problem might be solved by evaluating and simplifying the equation.
There are also bit-wise operation that may make the scenario more complex:

```
var tainted; // integer
tainted <<= 0x10;
var wrongly_tainted = (uint16_t)tainted;
```

In this case, *wrongly_tainted* will be marked as tainted, but it should always be *0* since everything has been shift to high 16 bits and low 16 bits are always *0*.
The way to solve this is to use bit-wise taint analysis to track the taint flow, and fortunately many frameworks provide this functionality.
Besides integer operation, there are also other type in JavaScript that shares the same operator in JavaScript. The most common one is *string*. We can simply regard string as an immutable array, so the taint state of each element is recorded individually. Therefore, the *plus* semantic being used for integer should not be used in same way as *string*, since taint status of *string* variable should not be recorded as a whole, unlike *integer*.
T-TCOND and **T-FCOND** are hardest problem in taint analysis and symbolic execution, in which people have investigated much time to study. In this paper, for these conditional jumps of SIMPIL, only the taint status of destination address is considered and abort failure when it is tainted(that means the control flow might be hijacked by attacker). However, the taint status of condition expression is not considered, this will cause problem.

```
var tainted; //boolean variable
var wrongly_untainted;
if (tainted)
    wrongly_untainted = 0xdeadbeef;
else
    wrongly_untainted = 0xcafebabe;
```

As shown, it is obvious that *wrongly_untainted* depends on boolean variable *tainted*, but it will simply be marked as untainted according to the algorithm. The way to solve this is to use static analysis about control flow dependency, but this is hard and not necessarily accurate especially when the complexity of the program arises. Especially in JavaScript, the various dynamic aspects make the static analysis even harder.

2.3.2 Forward Symbolic Execution

In [5], the forward symbolic execution is also discussed. Since my main focus is taint analysis, I will only mention this part briefly. Forward symbolic execution does not trace the input simply as a boolean taint state, but instead assign a symbolic expression for each variable that may be affected by input. Different from taint analysis, in which program is executed using concrete input value but with a taint state boolean state, forward symbolic execution executes the program using user input that is represented as a symbol rather than concrete value. When the conditional branch is executed, it is possible for the input to go through both branches, for each different possible branches, a *constraint* that satisfies or does not satisfy the condition is added. In JavaScript analysis, symbolic execution can be used to generate test cases using *constraint solving* in order to maximize *path coverage* as much as possible. SymJS[6] is a framework to do this.

2.4 Static Taint Analysis

Different from dynamic taint analysis, static taint analysis will not execute the program but analyze it statically by processing its source code.

2.4.1 Control Flow Graph (CFG)

This is a common approach used in compiler design, but it can also be used in the field of information security. CFG is a abstract way to represent a particular function in a program. For example:[9]

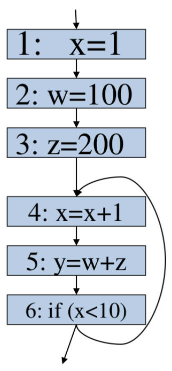


Figure 2.5: CFG example

Each node should represent a basic block in which the control flow will not change. It is the best to use each intermediate representation (e.i. byte-code) as the basic

block instead of one line of codes. Then we are going to use such graph to perform static taint analysis.
the **definition** is the node with assignment operation. **Reaching definitions** of a edge in CFG is all nodes with LHS which value can be propagated to this position without being rewritten. "A definition *d* reach some point *p* iff there is a path from *d* to *p* and *d* is not killed."[9] The **relevant reaching definitions** of a edge are subset of reaching definitions such that the values produced by the reaching definition is used in the node immediately after this edge. In the words of taint analysis, if *d* is relevant reaching definition of *p* and the LHS of *d* is the taint that we want to trace (e.g. *d* is something like a = *get_input()* and we want to trace a), LHS of *p* should be marked as tainted, and algorithm should continue to work with the new tainted variables. I will not detail the specific algorithm to calculate the reaching definitions here, since that is not the main focus of the report. It is not hard and can be found online.
When the tainted variable is passed into a user defined function, we should recursively build a CFG for that new function and perform static taint analysis.
This approach has the advantage that it only considers the instruction that is relevant to the taint flow, and ignores any other instructions. Therefore static taint analysis might be faster than dynamic taint analysis. However, the drawbacks of this approach is clear: it does not work when the codes becomes more complicated, and many problems arise. For example: what if the LHS is a array and the index cannot be determined statically? what if the the tainted variable is passed into a built-in API function like *substring*? what if the taint source is a string, do we regard it as tainted variable as a whole or regard each character separately as taint variable(but the length is unknown statically!) ? Whatever the design choice is, the static taint analysis will produce high number of false positives. (e.i. variables that should have been marked as tainted fail to be identified)
In addition, such static analysis algorithm fails to identify the implicit taint flow caused by conditional branch, which should be one of the main focus. Also, dynamic features of JavaScript cannot be resolved by this method.

2.4.2 WALA

WALA stands for *T. J. Watson Libraries for Analysis*, which is a static analysis framework that can be used to analyze the JavaScript. Author of JSBAF[14] has used this framework in the staic phase. It basically employs similar approach that I have discussed above.

2.5 Blended Taint Analysis

This approach has been discussed in the paper *Practical Blended Taint Analysis for JavaScript*[14]. The key idea of this approach is to record dynamic information like argument passed into *eval* when the program is run using the test cases, and such information is used in static analysis in the second step.

2.5.1 JSBAF

JSBAF stands for JavaScript Blended Analysis Framework, which is the framework being developed using the concept covered in [14]. The good test suites that offer good method coverage are assumed to be provided. These test cases are the ones that are used to provide the dynamic information for subsequent static analysis to use. The framework will analyze each test case individually and then combine their results to generate the final results.
The overall structure of JSBAF is shown below.[14]

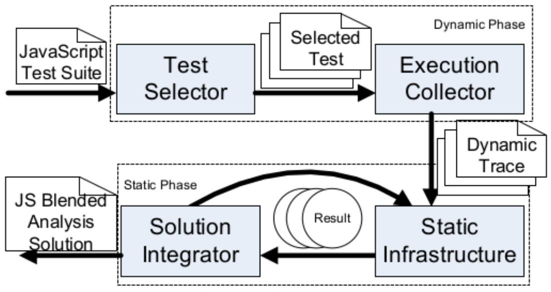


Figure 2.6: JSBAF

The framework is also flexible: user can replace a component without need to consider modifying other components.

2.5.2 Dynamic Phase

In this phase, **Execution Collector** gathers various dynamic information that cannot be determined statically, and then **Trace Selector** selects subsets of good page traces. In the solution of the paper, *TracingSafari* is used as the tool to trace the dynamic execution of JavaScript program. During the dynamic collection, *page trace* of each page will be recorded, which is sequence of instructions from the same page. *Page trace* contains a *dynamic call tree* for static analysis.
To optimize the analysis, *Trace Selector* will try to minimize the number of traces analyzed by calculating information of each page trace.

2.5.3 Static Phase

The author has used WALA in this phase. Firstly the **Call Graph Builder** is performed to transform the call graph of each page trace collected in static phase to WALA data structure. Then **Code Collector** is used to collect dynamic information such as dynamic generated codes and functions for WALA to analyze.
In addition, the call graph node of WALA will have a new *context* field to represent number of arguments passed to the call. If the length of arguments can be confirmed, unexecuted branch will also be pruned to improve efficiency.
Finally the **Static Taint Algorithm** is performed to obtain the result of possible vulnerabilities, the algorithm is similar to the one discussed in static taint analysis but with some optimization.

2.5.4 Reflection

This approach may be better than static taint analysis on the aspect that dynamic information of JavaScript can be considered. However, most of the problems discussed in static taint analysis such as implicit taint flow are still not figured out.

2.6 Dynamic Analysis on JavaScript

To perform dynamic taint analysis, we must be able to track and analyze every possible JavaScript operation, otherwise it is possible to miss some critical operations and produce false positive or false negative results. It is also required to pass information between operations, otherwise taint flow cannot be traced at all. When I was preparing the project, I found several possible ways that dynamic analysis can be performed.

2.6.1 Debug Protocol

The debug protocol is designed for remote debugger: for example, JavaScript IDE uses debug protocol to interact with JavaScript codes being executed in browser, and IDE can use debug protocol to step in, step out and continue until hitting breakpoint. We can also use this debug protocol to trace the program being analyzed and perform dynamic taint analysis. However, there are several problems.
Firstly, the on-line resource of debug protocol development is rare. The only resource seems to be the official document, which is a API documentation instead of a step-by-step tutorial, thus a bit hard to understand.
Secondly, I am not sure if debug protocol supports tracing every operation. For example, in step-in command in IDE, a statement `a = b * c + d` will be jumped over directly rather than be separated as multiplication and addition, but this is what we need for accurate dynamic taint analysis. Thus if debug protocol does not support such operation and I have spent too much time on it, it will be wasteful.
Therefore, using debug protocol is too risky and not appropriate for my project.

2.6.2 Modifying JavaScript Engine for Analysis

Since JavaScript is always executed by JavaScript engine, we can modify the code of JavaScript engine for dynamic analysis. However, here are several problems.
Firstly, if analysis is performed in this way, user must use modified version of browser, which is too heavy compared to a proxy or browser extension. In addition, the product will heavily bond to specific browser, so there would be no portability at all.
Secondly, such implementation is hard. I need to understand JavaScript engine in advance before modifying its code, which might take long. Also, JavaScript is not only executed by interpreter, but it will also be compiled just-in-time when a section of codes is executed very frequently. This could make things very complicated since

I would also need to modify the JIT compiler so that I can still analyze codes even if they are compiled in JIT.
Therefore, the cumbersomeness and difficulty of this approach suggests this is not a good way.

2.6.3 Instrumentation

Instrumentation is a common way to perform dynamic analysis. It modifies codes to be analyzed and insert the codes to perform analysis. For example, Intel Pin Tool [8] is a dynamic code instrumentation tool for binary program. AFL Fuzzer [15] also applies instrumentation technique to generate high code coverage for binary program. Although instrumentation for binary program is common, counterpart resources in field of JavaScript analysis are quite rare. Here are possible ways that JavaScript program can be instrumented.

Byte Code Level Instrumentation

JavaScript byte code is a intermediate representation of JavaScript, and it varies among different browser engine. Since byte code is more like assembly than a high level language, the instrumentation in this level is a bit similar to binary instrumentation that I have discussed above.
However, unlike binary instrumentation, in which there are many resources on-line, JavaScript byte code instrumentation has rarely been investigated before. Also such low-level instrumentation is too dependent on specific browser: for example, the instrumentation on V8 byte code, the JavaScript engine used by Chrome, cannot work at SpiderMonkey, the JavaScript engine used by FireFox. Therefore, this might not be a good approach.

Source Code Level Instrumentation

Source code instrumentation is to instrument through modifying JavaScript file, usually by proxy. Fortunately, there is one framework that provides such functionality, Jalangi2, which is a big advantage. Also, this does not depend on particular JavaScript engine in specific browser, so it is more portable. Therefore, this becomes my final choice of dynamic analysis, and I will explain this in detail in next section.

2.7 Jalangi2

2.7.1 Overview

`Jalangi2`[11] is a dynamic analysis framework for JavaScript that supports dynamic analysis based on instrumentation. This framework can modify the source code of the program being analyzed and provide interface that enables developer to instrument callback function before and after each JavaScript operation. For example, before and after any JavaScript binary operators (e.g. +), user of Jalangi2 can instrument his own functions to inspect and modify the behavior of such operation. Since this framework has already implemented language-level preprocessing such parsing and instrumentation, developer who uses this framework does not have to worry about it. Instead, according to its tutorial[12], it is very easy to use this framework. Here is an example that instrument on the `binary` operator in JavaScript.

```
//Analysis.js
(function (sandbox)
{
function Analysis(rule)
{
    this.binaryPre = function (...) {...}
    this.binary = function (...) {...}
}
sandbox.analysis = new Analysis();
})(J$);
```

By setting these fields to the function we want, when binary operator such as + is executed, our functions `binaryPre` and `binary` will also be executed. The relative argument such as operands will be passed as argument, and we can also use return value of these functions to modify the dynamic behavior of the binary operator such as changing the result.

Instrumentation

As I just suggested, Jalangi2 works by source code level instrumentation. The input JavaScript source will be converted to instrumented JavaScript, which will then be run by the JavaScript interpreter. Since the instrumented JavaScript will call the callback functions that we have defined (e.g. `binaryPre` shown above), analysis can be performed by every possible JavaScript operation.
In Jalangi2, every operation will be wrapped by a member function of `J$`, and this class can be regarded as the *main class* of Jalangi2. For example, binary operation would be wrapped by `J$.B`, in which our instrumentation callback function is called and actual binary operation are performed. These functions not only wrap the operation but also are placed before or after some JavaScript statement. For example, `J$.Se` is placed before JavaScript file, and `J$.Fe` is placed before function body. In these functions our corresponding callbacks will also be called, if any.
Here is an example that demonstrate behavior of instrumentation.

```
//before instrumentation
function add(a, b)
{
    return a+b;
}
```

```
//after instrumentation
function add(a, b) {
jalangiLabel0:
while (true) {
    try {
        J$.Fe(113, arguments.callee, this, arguments);
        // J£.Fe for function entry
        arguments = J$.N(121, 121, arguments, 4);
        a = J$.N(129, 129, a, 4);
        b = J$.N(137, 137, b, 4);
        return J$.X1(105, J$.Rt(97,
            J$.B(10, '+', J$.R(81, 'a', a, 0),
            J$.R(89, 'b', b, 0), 0)));
            // J£.B for binary operator
    } catch (J$e) {
        J$.Ex(921, J$e);
    } finally {
        if (J$.Fr(929))
            continue jalangiLabel0;
        else
            return J$.Ra();
    }
}
}
```

J$

As I suggested above, this is the main class of Jalangi2. Just as the example above shown, the `analysis` property need to be assigned by us to instrument callback functions. In addition, this class could be shared when files are chained together using `ChainedAnalyses.js` provided by Jalangi2, and this feature can be used to export class. For example, I have implemented a `Utils` class that contains some utility functions. Since this is implemented in a separate file, we cannot use this class directly in another file. However, by using `J$`, we can easily export this class, as illustrated below.

```
//Utils.js
(function (sandbox)
{
    Utils.prototype.func2 = function() {...};
    Utils.prototype.func1 = function() {...};
    //define func1 and func2
    sandbox.myUtils = new Utils();
    //store instance into `myUtils` field of J£
})(J$);
```

```
//Analysis.js
(function (sandbox)
{
    const utils = sandbox.myUtils;
    // fetch `myUntils` field from J£
    function Analysis()
    {
        // ...
        utils.func1(...); // use func1 somewhere
        utils.func2(...); // use func2 somewhere
        // ...
    }
    sandbox.analysis = new Analysis();
})(J$);
```

Run

Here is the bash command that run the analysis. `ChainedAnalyses.js` is critical since it is the script that chains everything together.

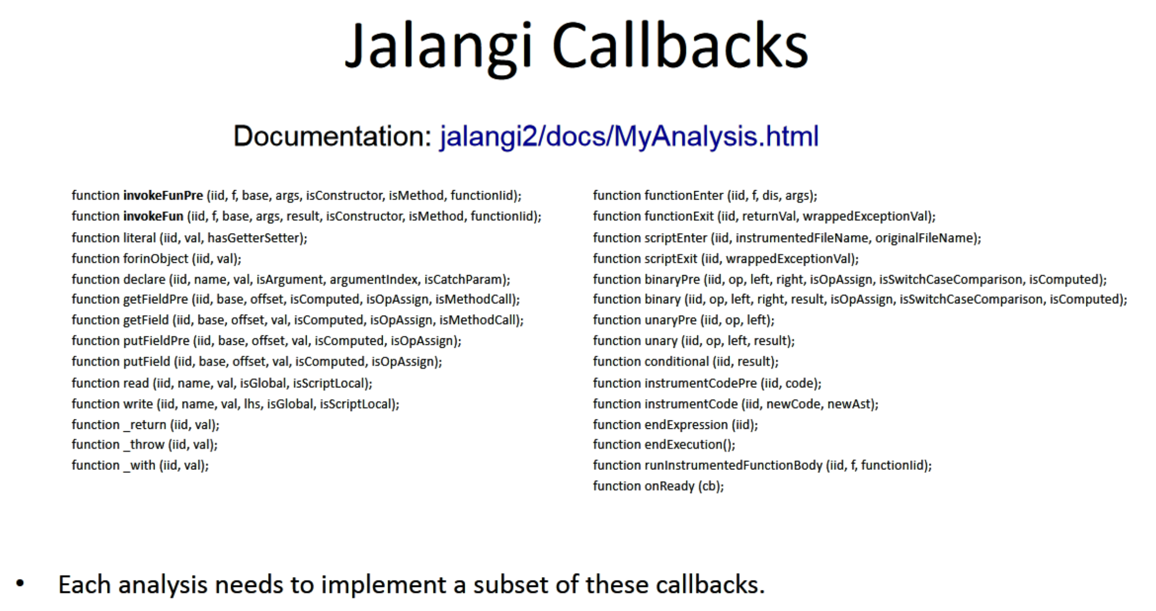
```
node jalangi2/src/js/commands/jalangi.js --inlineIID --inlineSource \
--analysis jalangi2/src/js/sample_analyses/ChainedAnalyses.js \
--analysis Utils.js \
--analysis Analysis.js \
file_to_be_analyzed.js
```

Operations

Here is the full list of the JavaScript operations that can be instrumented, which already cover all possible JavaScript program behaviors. [12]

2.7.2 Shadow Value[4]

`Shadow Value` is a concept formulated in [10]. The key point is that there could be another shadow value associated with a variable. In the paper, `AnnotatedValue` class is used to denote the variable that has any shadow value along with it. The `value` field of this class is the original value of this variable, while `shadow` field is the shadow value associated with this variable. For example, if an integer variable `1337` has shadow value `true`, the variable will be an `AnnotatedValue` object with field `value` being `1337` and field `shadow` being `true`, denoted as `AnnotatedValue(1337, true)` (I will use this notation in the following report). This shadow value concept is important because `JsTainter` will use shadow value to record the taint state about a variable, which is necessary in dynamic taint analysis.
However, I found that mechanism of shadow value of `Jalangi2`, works differently from the one mentioned in [10]. The reason might be that the version is different: the paper covers `Jalangi1` while I am using `Jalangi2`. Of course, `JsTainter` can



Jalangi Callbacks

Documentation: jalangi2/docs/MyAnalysis.html

```
function invokeFunPre (iid, f, base, args, isConstructor, isMethod, functionIid);
function invokeFun (iid, f, base, args, result, isConstructor, isMethod, functionIid);
function literal (iid, val, hasGetterSetter);
function forinObject (iid, val);
function declare (iid, name, val, isArgument, argumentIndex, isCatchParam);
function getFieldPre (iid, base, offset, isComputed, isOpAssign, isMethodCall);
function getField (iid, base, offset, val, isComputed, isOpAssign, isMethodCall);
function putFieldPre (iid, base, offset, val, isComputed, isOpAssign);
function putField (iid, base, offset, val, isComputed, isOpAssign);
function read (iid, name, val, isGlobal, isScriptLocal);
function write (iid, name, val, lhs, isGlobal, isScriptLocal);
function _return (iid, val);
function _throw (iid, val);
function _with (iid, val);
function functionEnter (iid, f, dis, args);
function functionExit (iid, returnVal, wrappedExceptionVal);
function scriptEnter (iid, instrumentedFileName, originalFileName);
function scriptExit (iid, wrappedExceptionVal);
function binaryPre (iid, op, left, right, isOpAssign, isSwitchCaseComparison, isComputed);
function binary (iid, op, left, right, result, isOpAssign, isSwitchCaseComparison, isComputed);
function unaryPre (iid, op, left);
function unary (iid, op, left, result);
function conditional (iid, result);
function instrumentCodePre (iid, code);
function instrumentCode (iid, newCode, newAst);
function endExpression (iid);
function endExecution();
function runInstrumentedFunctionBody (iid, f, functionIid);
function onReady (cb);
```

- Each analysis needs to implement a subset of these callbacks.

Figure 2.7

use `Jalangi1` instead of `Jalangi2`, but `Jalangi1` has not been maintained for many years, so using `Jalangi1` may exist more risk of encountering bugs in the framework. In [10] (`Jalangi1`), it is suggested to use a `AnnotatedValue` to replace some variables, just as I discussed above, and as long as the variable is used for some operation, `actual(value)` is used to convert it to actual value for operation. For example, `actual(AnnotatedValue(1337, true)) === 1337`. Then when our analysis callback function `analysis.binary` is called, `value` is passed as the arguments instead of `actual(value)`. The logic is shown as below, according to the pseudo-codes in the paper.

```
//definition of AnnotatedValue
function AnnotatedValue(val, shadow)
{
    this.val = val;
    this.shadow = shadow;
}
function actual(val)
{
    return val instanceof AnnotatedValue ? val.val : val;
}
//when executing instrumented code of binary operator
var result = actual(left) op actual(right)
//call `actual` before used as operands
if (analysis && analysis.binary)
    //original `left` and `right` are passed
    analysis.binary(op, left, right, result)
```

However, in Jalangi2, things works differently: here is the pseudo-code representing the source code logic of `Jalangi2` when binary operator is handled.

```
function B(iid, op, left, right, flags) {
var result, aret, skip = false;

if (sandbox.analysis && sandbox.analysis.binaryPre) {
    aret = sandbox.analysis.binaryPre(iid, op, left, right);
    if (aret) {
        op,left,right,skip =
            aret.op,aret.left,aret.right,aret.skip;
    }//a `binaryPre` is added
}
if (!skip) {
    result = left op right;
}//no `actual()` being applied before using as operands

if (sandbox.analysis && sandbox.analysis.binary) {
    /*
    `left` and `right` being passed to
    our `analysis.binary` handler
    are same as ones used as operands,
    which is different from the approach mentioned in paper
    */
    aret = sandbox.analysis.binary(iid, op, left, right, result);
    if (aret) {
        result = aret.result;
    }
}
return (lastComputedValue = result);
}
```

Therefore, it seems that `AnnotatedValue` class is not supported in `Jalangi2`, but instead, shadow value is associated with a object reference. `SMemory` is a mechanism that support shadow value feature in `Jalangi2`. However, the drawback of this approach is that we cannot have a shadow value associated with primitive value, including `string`. Therefore, since the approach of `Jalangi1` mentioned in the paper is better for me to use, I will define `AnotatedValue` by myself, and then define `analysis.binaryPre` to let `skip === true`, and perform calculation inside `analysis.binary` instead. In addition, I will do this for all operations, not only binary operator.
Here is the pseudo-codes that describe what I am thinking about.

```
this.binaryPre = function(iid, op, left, right)
{
    return {op:op,left:left,right:right,skip:true}//skip
}
```

```
this.binary = function(iid, op, left, right, result)
{
    var result;
    var aleft = actual(left);
    var aright = actual(right);

    result = left op right;
    //use left and right to perform analysis

    return {result : result}
}
```

In this way we can use the shadow value in the same way as `Jalangi1`.

2.7.3 Location

Jalangi2 has provided an `iid` argument for each instrumentation callback function, which is the `Static Unique Instruction Identifier` that is used to specify the specific instruction that is being analyzed currently. It can also be used to specify the position of the instruction: `J$.iidToLocation(J$.getGlobalIID(iid))` is the code that can be used to get the position of current instruction being analyzed in the original codes, where `J$` is the global variable used by `Jalangi2` mentioned above. The returned position is a string, with format `([absolute path of js file]:[starting line number]:[starting column number]:[end line number]:[end column number])` in `node.js`. For example, if there is an assignment `a = b + c` at line 17, and at the callback instrumentation function of writing the value into `a` (e.i. `write`), the position being obtained will be `/path/to/file.js:17:1:17:2`, while `1` and `2` stand for `a` starts at column 1 and ends at column 2. We can use such position string to identify the position of a particular instruction in original codes.
However, when instrumentation is run in browser, position with different format appears, after reading the source codes of Jalangi2, I found this piece of code that generates the position string in `iidToLocation.js`.

```
arr = ret[iid];
if (arr) {
    if (sandbox.Results) {
        return
            "<a href=\"javascript:iidToDisplayCodeLocation('"+gid+ \
            "');\">(" + fname + ":" + arr[0] + ":" + arr[1] + \
            ":" + arr[2] + ":" + arr[3] + ")</a>";
    } else {
        return "(" + fname + ":" + \
            arr[0] + ":" + arr[1] + ":" + \
            arr[2] + ":" + arr[3] + ")";
    }
} else {
    return "(" + fname + ":iid" + iid + ")";
}
```

Therefore, as shown above, there are 3 kinds of position string format: the first one gives `gid`, file name and information about line and column; the second one does not gives `gid` but still gives file name and information about line and column, which is the format that I appears in `node.js` illustrated above; and the third one only gives file name and `iid`, because `arr` that should contains information about line and column now is `undefined`. During my project I have not encountered the third case, so I think that one is a bit unrelated to the project. As for the other 2 cases, although what Jalangi2 chooses to provide is a string, what I need is actually these variables used to generate this string. Thus, instead of obtaining such string and parse it back by my myself, I would choose to modify the code of Jalangi2 to make `iidToLocation` return a JSON instead of a string. Here is the modified version of code.

```
arr = ret[iid];
if (arr) {
    if (sandbox.Results) {
        return {gid:gid, fname:fname, pos:arr};
    } else {
        return {fname:fname, pos:arr};
    }
} else {
    return {fname: fname, iid:iid};
}
```

The information being returned is exactly same, except now the form is in JSON instead of a string.

2.7.4 Bug

Even if Jalangi2 is an great framework, some bugs still exist since it has not been maintained for 3 years. There is one bug that is triggered very frequently, so I have fixed this bug by modifying some codes in Jalangi2.

Constant Declaration

Cause
In JavaScript, developer can declare a constant that cannot be modified once being assigned by `const a = something`. However, we cannot access the variable before this statement.

```
alert(a);
//cause ReferenceError instead of returning \texttt{undefined}
const a = 1 + f(a); // this also causes ReferenceError
```

In the code, variable a are used before declaration and assignment, which causes ReferenceError. This JavaScript feature leads to a bug in Jalangi2, which is caused by improper instrumentation. The bug is triggered when any constant declaration is instrumented by Jalangi2, for example const x = "1". When this declaration is instrumented, following instrumented JavaScript will be generated.

```
J$.N(41, 'x', x, 0); // x is used before statement `const x`
const x = J$.X1(25, J$.W(17, 'x', J$.T(9, "1", 21, false),
    x, 3)); // x is also used here
```

As I mentioned in last section, J$ is the main class of Jalangi2, and its member functions are called. However, the problem is that variable x is used before it is actually declared, which causes ReferenceError when the instrumented version is run.
By the way, declaration like var a = a (where a has not been declared before) will not cause this problem and a will equal to undefined after this statement, and this is the reason why var declaration does not trigger this error.
Fix
My approach to fix this error is very simple. The reason why instrumented version of JavaScript file access the variable before declaration is that original value of the variable need to be passed as argument, as the documentation of Jalangi2 suggests. However, this functionality is not actually very useful, at least not useful for my project. Therefore, if we can prevent such early access of variable (e.g. change that argument to something else), the ReferenceError can be prevented. To be specific, we need to change x passed as argument to both J$.N and J$.W.
In Jalangi2, file src/js/instrument/esnstrument.js implements JavaScript source code instrumentation, and we need to find the piece of codes that generates the corresponding instrumented JavaScript, and modify the arguments that cause error. The idea is that when the corresponding instrumented code in string form is generated, method name ".N" and ".W" must be referenced, so I decided search such string in that file, and these 2 lines are interesting and possibly relate to instrumented code generation.

```
var logInitFunName = JALANGI_VAR + ".N";
var logWriteFunName = JALANGI_VAR + ".W";
```

We may need to debug the Jalangi2 to make things more clear, so I have used a shortest code that generate the ReferenceError, namely const x = "1", to be instrumented. By setting the breakpoint, we can easily find that JALANGI_VAR equals to "J$". Then after looking for the cross reference, these variables are used in this way, which is very likely to be instrumented code generation. There are several pieces of codes like this, but they are essentially same, so I will only show and explain one of them here.

```
logInitFunName + "(" + RP + "1, " + RP + "2, " + RP + "3, " + ...
```

However, if I set a breakpoint here, I found that value of `RP` equals to `"J$_"`, so the result of string concatenation is something like `"J$.N(J$_1, J$_2, J$_3, 0)"` which is not same as result instrumented code. However, this result will be passed to function `replaceInStatement`, which I think is the function that replaces the `J$_X` by the real argument. However, this is not important, the arguments can already be modified now although a bit hack: `J$_3` corresponds to the third argument `x` that causes `ReferenceError`, so we modify it to something else such as `J$_2`, so the third argument will become `'x'` now. The same thing applies for all other references of `logInitFunName` and `logWriteFunName`. This is the instrumented codes after modifying Jalangi2 code, in which all `x`s are replaced by `'x'`.

```
J$.N(41, 'x', 'x', 0);
const x = J$.X1(25, J$.W(17, 'x', J$.T(9, "1", 21, false), 'x', 3));
```

Chapter 3

Design

3.1 Overview

3.1.1 Choice

In the background chapter, I have covered 3 approach: dynamic taint analysis, static taint analysis and blended taint analysis. The blended taint analysis is simply improved version of static analysis, so the key point is to compare blended taint analysis with dynamic taint analysis.
Although blended analysis gather the dynamic information firstly, it still applies static analysis to get the final result. By contrast, dynamic analysis traces the program and perform analysis as it runs, which should give higher accuracy. The reason is that the dynamic information that can be gathered by blended analysis in first stage is not informative enough: since the analysis and dynamic tracing are done separately, the dynamic information that can be used by static analysis is limited. By contrast, dynamic analysis can provides almost all dynamic information because the analysis is done along with the dynamic tracing.
For example, when array and object are used to store tainted user input, the blended taint analysis might not function well.

```
const hash = window.location.hash; // hash is tainted
const obj = {hash: hash};
function get(obj, prop)
{
    return obj[prop];
}
const r = get(obj, "hash"); // equivalent to obj.hash
```

If blended analysis is used here, it is hard to identify `r` is actually fetched from `obj.hash` unless using complicated program analysis techniques; however, using dynamic taint analysis, since analysis is performed along with program tracing, it is easy to figure out `obj.hash` is returned and assigned to `r`, which should be tainted.
However, there is an advantage of blended analysis: the implicit flow can be detected through program analysis techniques. Since dynamic analysis can only perform analysis for every JavaScript operation separately, it is hard to view and analyze

the JavaScript codes as a whole to figure out implicit flow. By contrast, in static phase of blended analysis, if good program analysis technique is applied, it is possible to detect implicit flow. Nonetheless, as the program logic becomes complicated, program analysis can also get wrong. Therefore, considering the difficulty of performing program analysis on JavaScript and the uncertain effectiveness of automatic implicit flow detection, I would still favor dynamic analysis rather than blended analysis.

3.1.2 Overall Structure

Here is the UML graph of my project.

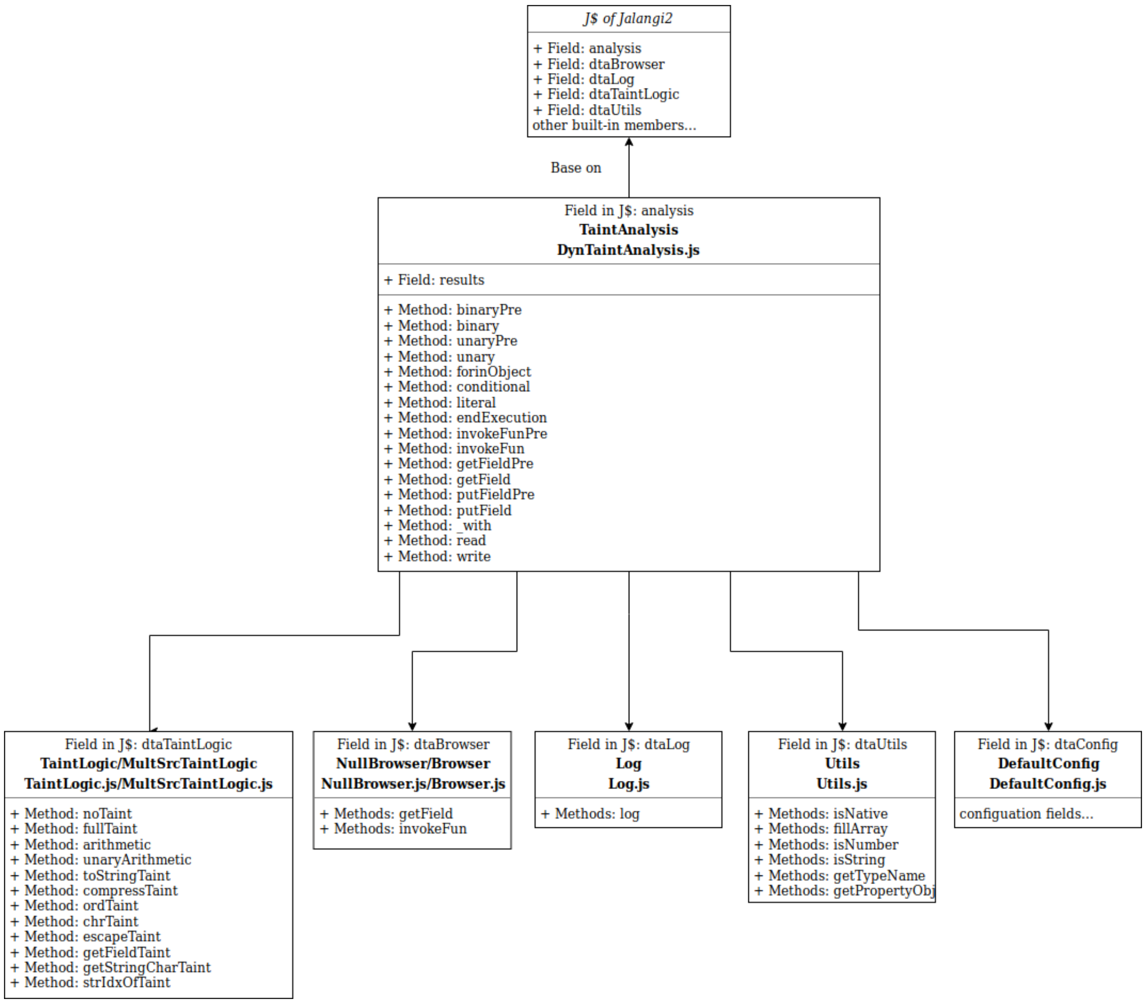


Figure 3.1

As I have suggested in background chapter, everything is based on Jalangi2 framework, thus every class is a field of `J$`.
Field `analysis` is an instance of class that implements the main dynamic taint analysis logic, which is implemented in file `DynTaintAnalysis.js`. In this class, callback functions that will be called in runtime dynamic analysis are defined and implemented. The `results` field is used to store the result of taint analysis, which will be covered later.

Field `dtaTaintLogic` is the actual implementation of taint propagation rule for the particular type of `taint information variable`, which will be covered later. In other word, the `template method design pattern` is used here: if I want to change the type of `taint information variable`, ideally I only I need to change the instance stored in `dtaTaintLogic` field, without modifying codes in other files, which is a good software engineering practice.
Field `dtaBrowser` is some browser-side handling codes. This is separated from `DynTaintAnalysis.js` because I want the code to be both runnable in `node.js` and in browser. This is also a `template method design pattern`, and I will cover the detail about this when browser integration is discussed.
Field `dtaUtils` is a utility class in which some utility functions are implemented. These functions could not only be used by `DynTaintAnalysis.js`, but can also be used by other files, because this is simply a low-level utility class.
Field `dtaConfig` is a configuration instance used to specify the some behaviors of taint analysis algorithm. Relevant properties will be covered in detail later.

3.2 Design of Shadow Value

3.2.1 Taint Information Variable

`Taint Information Variable` is used to record the taint state of basic variable types such as number and single character. I will discuss how taint information variable, which are used to describe basic type only, can be used to describe taint state of complex types such as `Object` in next subsection. In this subsection, I will discuss several design choices about taint information variable.

Boolean Variable

This is the simplest design: the shadow value is simply `true` or `false`. `true` if the variable is tainted, and `false` if the variable is not tainted. This is the easiest design choice to implement. In the following report, if I am going to make an example of `AnnotatedValue`, I will use this design choice in the example because this is simple and makes the example easy to understand.

Boolean Array for Sources

User inputs of web page in browser can come from different sources. For example, user input can come from argument in URL and `<input>` HTML label. Because of that, it is necessary to be able to identify the source of the taint given a tainted variable.
To implement this, a boolean array can be used, in which different indexes denotes taint state from different source. For example, element at index 0 can be taint state from source URL argument, while element at index 1 can be the taint state from `<input>` HTML label. If one array element at particular index is `true`, it means current basic variable can be affected by the source corresponding to that index.

There is also an possible optimization: instead of using boolean array, we can use an integer instead, where each bit correspond to an original boolean variable in array. Such optimization saves space and time, but exerts limitation on number of sources: maximum number of source is 32 if the integer is only 32-bit wide, for example.

Boolean Array for Bits

Instead of having only one boolean variable for an integer, such array traces the taint information for every bit of the integer. This can be used to handle edge case of bit operation such as `&` and `|`. For example, the following code may cause false positive if we simply use a boolean variable to record the taint information.

```
var t1,t2,r;
//t1 and t2 are tainted integer
t1 = t1 & 0xff;
t2 = t2 & 0xff00;
r = t1 & t2;
```

In this example, `r` must always be `0` whatever what `t1` and `t2` are, but if only a boolean variable is associated with the integer, false positive will be caused. If we taint the result as long as one of the operands are tainted (which is a common design), obviously `r` will be falsely tainted. However, if we trace the taint information at bit level, only `0-7 bits` of `t1` and `8-15 bits` of `t2` are tainted before the last statement. Noting untainted bits to be all zeros, dynamic taint analysis can give the result that finally non of the bit in `r` is tainted.
Such boolean array can also be optimized to integer like `Boolean Array for Sources`. In addition, since such design is independent from boolean array for sources, they can be combined so that the `taint infomation variable` is a 2D boolean array, although sounds very expensive.
However, tracing at bit level is expensive and unnecessary. Cases like the example above rarely occur so it is not worthy to have such expensive design. To compensate such drawbacks, message can be recorded to inform analyzer the possibility of making false positives.

Taint Level

This is an “soft” version of boolean variable. Unlike boolean variable which can only be `true` or `false`, tainted level is a floating point number that range from 0 to 1. Value 1 means the variable along with this shadow value is fully tainted. In other word, the variable can be fully controlled by user. Value 0 means the variable is not tainted and cannot be controlled by user. The value in between means the variable can be controlled by user, but cannot be fully controlled by user.
For example, there is a number with taint level 1, say `AnnotatedValue(1337, 1.0)`, and is converted to string, so the result is `"1337"`. However, what taint information variable should I assign to each character? If a boolean strategy is used, they are all `true`, which makes sense but is not completely accurate, because user can only

control the string within the range `‘0’-‘9’`, and this is different from a string that can be fully controlled. This is where `taint level` comes, the taint level for each character of this string should be somewhere between 0 and 1.
Nonetheless, specific rule for this strategy need more investigation: for example, what specific formula should be applied to calculate taint level as the tainted variable propagates? The details can give rise to many problems, so this strategy should be regard as extension.

Symbolic Expression

This is actually not about taint analysis but about symbolic execution. Instead of just recording a state of taint, whole expression is traced. I have covered some of this when [5] is discussed in background section.
However, different from normal symbolic execution which will result in multiple states when a symbolic expression is used in conditional jumps, my current code does not support such multiple states, because different implementation of taint variable is simply different implementation of `dtaTaintLogic` field in `J$`. Thus, the main taint analysis logic in `DynTaintAnalysis.js` (will be discussed later) does not vary as the implementation of taint information variable changes, and this file is not designed to have multiple states.
Therefore, current strategy could only be to log the expression of both sides when a symbolic expression is used in conditional expression, such as ===.

3.2.2 Shadow Value for Different Types

The `taint information variable` discussed in last subsection can only be used to describe basic types. However, we need to consider cases like `Object`, `Array` and `String` that are not appropriate to just have one `taint information variable`.

String

The reason why it is not good to mark the whole string as tainted or not is that part of the string can be affected by user while part of the string cannot. For example, user can control the argument field of an URL, which should be marked as tainted, but cannot control the domain field, which should not be marked as tainted. Therefore, shadow value of `String` is an array of `taint information variable` whose length is same as the length of the string. For example, `AnnotatedValue("AABB", [true,true,false,false])` denotes that `"AA"` part is tainted but `"BB"` part is not tainted.

Object and Array

Obviously it is not accurate to mark the whole object or array as tainted or not, because values stored in the object or array are independent and they can always be modified. For example, if `obj` is an `Object`, after executing `obj["a"] = "b"` and `obj["t"] = some_tainted_string`, using one `taint information variable` only, we cannot track taint states of these 2 fields independently.

Therefore, the better design choice is to taint the values inside `Object` or `Array`, rather than taint the object or array itself as a whole. For example, if we have an array of integers and all of them are tainted, we do not taint the array to a `AnnotatedValue` with actual value as an array and shadow value as a boolean, but taint each individual elements such that the array becomes an array of `AnnotatedValue` with actual value as an integer.
(e.i. `[AnnotatedValue(1,true), AnnotatedValue(2,true)]` instead of `AnnotatedValue([1,2], true))`

Originally I employed such design. However, later on, I found that this design is still naive, so I then reconstructed my code. This problem is when the array is used in some operation, sometimes the taint information must be stripped before using the array. I will cover the detail about it later in *taint stripping and merging* subsection. Therefore, an alternative design is to still wrap the object or array with `AnnotatedValue`, but use an `Object` to describe taint state instead of a single `taint information variable`. Using the example above, in current design, the structure will be `AnnotatedValue([1,2], {'0':true, '1':true})`. I will also discuss the details later.

Nonetheless, we cannot make property tainted, since in JavaScript only `String` or `Number` type is allowed to be used as property. Even if we assign value using `AnnotatedValue` as the property, it will still be casted to `String` before being used. However, the drawback is that sometimes property should have been tainted. For example, if argument passed to `JSON.parse` are a string that is totally tainted (e.i. every character is tainted including keys), the key should be tainted intuitively, but that's not the case in current design. Fortunately, usually websites will not use key to propagate information, so this drawback does not hurt so much.

Note that, since `null` is also an `Object`, so it would not be tainted either.

There is also another point to note: as long as an object is created, it must be wrapped by `AnnotatedValue`. For example, when an object literal is assigned to a variable (e.g. `var o = {}`), in the `analysis.literal` callback, must be changed to `AnnotatedValue({}, {})`. The reason is when `putField` is applied to any object, base variable cannot be modified in the `analysis.putField` callback function. Even if the assigned value is tainted, we are not able to change the base to `AnnotatedValue` anymore. Therefore to prevent such cases, objects created by JavaScript program should always be wrapped by `AnnotatedValue`.

Basic types

For any other basic types, such as `Number` and `Boolean`, have shadow value with only one `taint information variable`. This also includes the case like `undefined` and `NaN`.

Function

In current design, function variable is never tainted, since it is quite rare for function to be controllable by user.

Chapter 4

Algorithm Implementation

4.1 Overview

JavaScript is a dynamic and weakly-typed language. Due to such feature, the taint analysis implementation is much more complicated than dynamic taint analysis over binary executables. In this section I will discuss my implementation of JavaScript dynamic taint analysis, which I have employed in this project.

4.1.1 Common Functions

Before covering the implementation of dynamic taint analysis algorithm, I will discuss implementation of some basic functions that are used frequently first.

`isTainted`

This is the function that returns `true` if the shadow value is tainted, and returns `false` if not. The shadow value can be `taint information variable` for basic types, and array of `taint information variable` for string types, and object for object types. These 3 cases are also illustrated in codes.

```
function isTaintedH(taint, outArr)
{
    if (Array.isArray(taint))
    {// original value is string type
        for (var i = 0; i < taint.length; i++)
        {
            if (taint[i] !== rule.noTaint)
                return true;
        }
        // return false only if no character is tainted
        return false;
    }
    else if (typeof taint === 'object')
    {// original value is object
```

```
        outArr.push(taint);
        for (var k in taint)
        {
            if (outArr.indexOf(taint[k]) === -1)
            {// prevent infinite recursion
                //recursion
                if (isTaintedH(taint[k], outArr))
                    return true;
            }
        }
        outArr.pop();
        // return false iff nothing inside is tainted
        return false;
    }
    else
    {// original value is basic type
        return taint !== rule.noTaint && typeof taint != 'undefined';
    }
}
function isTainted(taint)
{
    return isTaintedH(taint, []);
}
```

`rule.noTaint` is the value of shadow value that stands for untainted state. For example, this value is `false` if boolean shadow value is used, and is `0` if number shadow value is used to represent an array of boolean variables. In `isTaintedH` function, as long as one of the shadow values of the variable does not equal to `rule.noTaint`, the function will return `true`.
The reason why there is a variable `outArr` is to prevent infinite recursion caused by circular reference. For object with circular reference, its shadow value is also an object with circular reference, so it is required to record references of object that does not need to be recursed again, and perform recursion call only if an object reference is not in the `outArr`.

`getTaintResult`

When a `AnnotatedValue` object is going to be returned, some optimization is required: if the shadow value is not tainted at all, `AnnotatedValue` wrapper is not needed so the value can be returned directly. Therefore, `getTaintResult` is implemented: given a value and its shadow value, return `AnnotatedValue` object if it is really needed (e.i. the `taint` is really tainted). However, there is one exception: as suggested in background section, a object type variable should always be wrapped by `AnnotatedValue`, so it should still be wrapped by `AnnotatedValue` even if it is untainted.

```
function getTaintResult(result, taint)
{
    if (isUntainted(taint))
    {
        if (typeof taint == 'object' && !Array.isArray(taint))
            return new AnnotatedValue(result, {});
            // object should be wrapped always, even is untainted
        else
            return result;
            // return the value only
    }
    else
        return new AnnotatedValue(result, taint);
        // wrap them with AnnotatedValue and return it
}
```

`getPosition`

Function that converts `iid` to position, which is a JSON storing the position of instruction represented by given `iid`, as covered in background chapter.

```
function getPosition(iid)
{// sandbox is J$ here
    return sandbox.iidToLocation(
        sandbox.getGlobalIID(iid));
}
```

4.2 Taint Stripping and Merging

As I mentioned when discussing shadow value of object and array, there are 2 possible design choices: for example, we can have either `[AnnotatedValue(1,true), AnnotatedValue(2,true)]` or `AnnotatedValue([1,2], {‘0’:true, ‘1’:true})`, where the second one is more favorable. However, even if second form is a more favorable design, sometimes the first form is easier to process. Therefore, we need some functions that enable us to switch between these 2 forms. To make it simple, I will call the first form as `merged form` and the second form as `stripped form` in the following section.
`Taint stripping` is the operation that *recursively* transforms the `merged from` to `stripped form`. I will explain what “recursively” means here later. If the merged form design choice is used for object and array variables (which is my original design), and when variable is used in JavaScript operations, the taint information must be stripped first to ensure the correctness of the result of JavaScript operations. These operations include JavaScript native function call and basic operator operation. The correctness of the operation can be affected by `merged form` as shown below.

```
var arr;
//arr is [AnnotatedValue("AB", [true,false])]
arr += "C";
```

If `arr` variable is in merged form, `"[object Object]C"` will be the final result of `arr`, which is not correct, because dynamic taint analysis should not change the behavior of the program being analyzed, and the final result of `arr` should be `"ABC"` instead, as shown below.

```
// AnnotatedValue("AB", [true,false]) is essentially ["AB"]
> var arr = ["AB"]
undefined
> arr += "C"
'ABC'
> arr
'ABC'
```

Therefore, to prevent such case from occurring, we need to *strip the taint information* (e.i. transform the variable to `stripped form`) to recover the original variable before putting them into JavaScript built-in operation. `JsTainter` has implemented a function called `stripTaints` to separate taint information and real value from a `merged form` variable. The return value is an object with `taints` field being `taint values` and `values` field being `real values`.

```
//The way stripTaints function should be used
var v;
var sv = stripTaints(v);
sv.taints; //access taint part
sv.values; //access value part
var stripped_form = new AnnotatedValue(sv.values, sv.taints)
//convert it into `stripped form`
```

Here are some examples illustrating the effect of calling `stripTaints`.

```
//Example 1:
//before stripping
AnnotatedValue(1, true)
//after stripping
1
true

//before stripping
[AnnotatedValue("AB", [true, false])]
//after stripping
["AB"] // values
{'0':[true,false]} // taints
```

```
//before stripping
{i:new AnnotatedValue(1, true), u:1337,
    u2:AnnotatedValue(1337, false),
    o:{x:new AnnotatedValue(2, true),
        a:[new AnnotatedValue("A", [true])]}}
//after stripping
{i:1, u:1337, u2:1337, o:{x:2,a:["A"]}} // values
{i:true, o:{x:true, a:{'0':[true]}}} // taints
```

When taint is stripped from a variable, it separates the real values and taint information into 2 different variables, and this is done recursively: if the variable is an object and there are more objects inside that object, objects inside the object are also stripped. If the original object is an `Object` or `Array` type, the basic structures of these separated variable are same as the original variable, which is illustrated well in the examples.
Here is a few points to note:

1. Since original variable is an object, it is not cloned, and it will share the same reference as the `values` part. So, if I use the original object reference after `stripTaints` is called to it, the stripped object instead of the unstripped object will be obtained.

2. Even if original structure is an `Array`, when it is separated, the taint part becomes an `Object` with numeric string as properties. The reason of this design is that if we keep it as `Array`, it will be confused with taint information of `String`, which is also an `Array`. The JavaScript feature that make this design proper is that `a[123]` and `a["123"]` will always be mapped to the same value, no matter `a` is a `Object` or an `Array`, so a numeric string as property will not cause any problem even if number is used to access the object field, and vice versa.

3. Recursion should not be applied to circular references, since that will cause infinite recursion. Instead, a stack that stores all current outer object references should be used to check the existence of circular reference, and if any, skip that variable without applying any recursion call to it.

4. As optimization, if a specific field of object or array is not tainted, corresponding field in taint information object will be simply `undefined` instead of `false`. In the third example, `u2` and `u` fields are both untainted, and in the taint object these 2 keys are simply undefined instead of having a `u2:false,u:false`.

Paired with `stripTaints` function, I also need to implement a `mergeTaints` function that transforms the `stripped form` to `merged form`. Here is the way to use this function.

```
var stripped_form; // a stripped-form AnnotatedValue instance
v = mergeTaints(actual(stripped_form), shadow(stripped_form))
```

Similar to `stripTaints` function, there are also some points:

1. The result object being returned is not cloned from the input, but share the same reference as the first argument passed to `mergeTaints`.
2. Since for the untainted fields, corresponding field in shadow value is undefined, it is more efficient to traverse taint information object instead of real value object, and assign a newly created `AnnotatedValue` with shadow value fetched from taint information object to the corresponding field of real value object.
3. Like `stripTaints`, circular reference is also checked and being prevented from infinite recursion.

Implementation

This is the codes for function `stripTaints`

```
function stripTaintsH(val, outArrs)
{
    var aval = actual(val);
    if (typeof aval == 'object' && aval === val)
    {// only perform stripping when \texttt{val} is a merged-form object
        outArrs.push(aval);
        // push the object being processed to prevent infinite recursion
        var taints = {};
        for (var k in aval)
        {
            if (outArrs.indexOf(val[k]) === -1)
            { // iterate unprocessed elements
                var stripped = stripTaintsH(val[k], outArrs); // recursion
                if (isTainted(stripped.taints))
                    taints[k] = stripped.taints;
                    // create taint only if the variable is tainted
                val[k] = stripped.values;
                // update value
            }
        }
        outArrs.pop();
        return {taints:taints, values:val};
    }
    else
    {// variable is basic type
        return {taints:shadow(val), values:actual(val)};
    }
}
function stripTaints(val)
{
    return stripTaintsH(val, []);
}
```

The implementation is very similar to depth-first-search algorithm. The `stripTaintsH` is a helper function that given a variable in `merged form` and `outArr` that is used to record references of already recursed object, and returns `taints` and `values` which are `stripped form`.
The codes for `mergeTaints` are very similar, which is also a depth-first-search algorithm.

```
function mergeTaintsH(val, taints, outTaints)
{
    outTaints.push(taints);
    for (var k in taints)
    {
        if (typeof taints[k] == 'object' && !Array.isArray(taints[k]))
        {//recurse for unseen non-basic type taint
            if (outTaints.indexOf(taints[k]) === -1)
                val[k] = mergeTaintsH(val[k], taints[k], outTaints);
        }
        else if (isTainted(taints[k]))
        {//merge directly for basic types
            val[k] = new AnnotatedValue(val[k], taints[k]);
        }
    }
    outTaints.pop();
    return val;
}

function mergeTaints(val, taints)
{
    if (typeof taints == 'object' && !Array.isArray(taints))
    {// if taints are object other than Array, val should be object
        return mergeTaintsH(val, taints, []);
    }
    else
    {// for basic types including string, return directly
        return getTaintResult(val, taints);
    }
}
```

4.3 Result of Taint Analysis

Since dynamic taint analysis is a kind of analysis that gives the result of information flow of a particular program when it is run with some given input, we need some ways to represent such results. The results of the dynamic taint analysis is stored in an array called `results`, each element is a piece of record, and the records are pushed into the array in execution order (e.i. if record A appears before record B when the program is executed, then index of record A in that array will be smaller than index of record B). One piece of record is stored in JSON form, and there are 5 types of record: `read`, `write`, `source`, `sink` and `log`.

Tainted Variable Read and Write

Although `read` and `write` are 2 different types of records, their implementation are very similar, so I will discuss them together. As the name suggests, a `read` record stands for the information that a tainted variable is read (e.i. used), while a `write` records stands for the information that a variable is written by a tainted value (e.i. assigned). The idea is clear: I want to represent the taint flow as the program executes, so when a tainted variable is used or when a tainted value is assigned to a variable, there might be some taint flow, which I need to record and to be shown to analyzer.
What I mean by tainted here is not only a tainted basic-type value like a tainted integer, but also a tainted object or string. As long as one of the character in string is tainted, or as long as one of the value inside the object is tainted, the object or string will be regarded as tainted.
The `read` record will be pushed into `results` when `analysis.read` and `analysis.getField` callbacks are called, and similarly the `write` record will be pushed into `results` when `analysis.read` and `analysis.getField` callbacks are called, when relevant value used in the operation is tainted.
Fields
There are several fields in a piece of `read` and `write` record:
`type` is used to specify the type of the record, and should be `"read"` for `read` record and `"write"` for `write` record.
`typeOf` is used to specify type of the tainted value. For `read` record, this is the type of the tainted variable being read; and for `write` record, this is the type of tainted value used to assign the variable. The reason why such information might useful is that current approach to record taint information flow sometimes causes inaccurate result. For example:

```
var arr;
//arr is an Array with one of the element being tainted
arr.push(1);
```

When `arr.push(1)` is executed, `arr` object will actually be read and thus `analysis.read` callback will be called. Since `arr` is tainted, a `read` record will be pushed into `results` array. However, this does not really make sense because here we are

pushing a value into `arr` instead of propagating any taint value. The key problem is that here it is the reference of `arr` that is used instead of tainted variable inside the object.
I could simply make no recording when the value passed into `analysis.read` is an object type and record only in `analysis.getField` when tainted variable inside an object is fetched, but I think it is better to record everything but specify the type of the value in field `typeOf`, so that more information can be provided; and analyzer can also choose to filter out the records with object type if not needed.
`file` and `pos` fields are used to specify the position where this record occurs: `file` is the filename and `pos` specifies line number and column number in array form. They come from the return value of `iidToLocation` (`fname` and `pos` fields respectively), which I have covered in background chapter.
`name` is the variable name or field name. For example, statement `a.c = b` (`b` is tainted) will produce `"b"` as `name` field of the `read` record, and `"c"` as `name` field of the `write` record.
Implementation
Since `write` record and `read` record are very similar, and the only difference is the `type` field, they are implemented in same function, `rwRec`, except type is accepted as argument, which are passed differently in `analysis.read` and `analysis.write`.

```
function rwRec(analysis, iid, name, val, rw)
{
    const pos = getPosition(iid);
    if (val instanceof AnnotatedValue || isTainted(shadow(val)))
    {// if val is tainted
        const typeOf = typeof actual(val);
        analysis.results = analysis.results.push(
            {// push relavant information into `results` array
                type: rw, typeOf: typeOf,
                file: pos.fname, pos: pos.pos,
                name: name
            });
    }
}
this.read = function (iid, name, val)
{
    rwRec(this, iid, name, val, 'read');
};
this.write = function (iid, name, val)
{
    rwRec(this, iid, name, val, 'write');
};
//everything is same except type field
```

In addition, `this.read` and `this.write` functions are also called in `this.getField` and `this.putField`, respectively, in order to record when tainted value is fetched from or set to a field of any object.

Source and Sink

These 2 types of records will only appear if JsTainter is run on web application. They are similar to `read` and `write` records except they are used to specify reading from source and writing to sinks. I will cover the details of these concept in browser chapter later, so here I will only cover the structure of JSON for these 2 types.
Fields of Type Source
The JSON structure of `source` is almost identical to `read` type record, except there is one more field, `id`, which is the id of this source. This can be used to distinguish different sources when there are multiple sources in browser.
Fields of Type Sink
The JSON structure for `sink` is also similar to `write` type record, but here more information about tainted value being passed to sink is stored: instead of simply storing the type of the variable in `typeOf` field, real value and shadow value are stored in field `value` and `shadow` respectively, if they can be converted to JSON (e.i. not a circular structure).

Special Information

The special information is also one of the results of taint analysis, and its `type` field is `"log"` to be specific. It is used to record special information that user might want to note about. For example, information could be recorded when tainted variable is used in a `if` statement. User can also choose whether to record a particular type of special information by setting the configuration. The different types of special information will be covered later when relevant concept is covered.
Fields
`file` and `pos` fields are exactly same as ones in `read` or `write` record.
`msg` is the message string used to inform analyzer, which is different for different special information.

4.4 Taint Propagation

In this section the taint propagation rule will be covered. Although the final product that works in browser has employed multiple source boolean array as `information taint variable`, I will still use boolean as `information taint variable` to demonstrate taint propagation rules. The primary reason is that I want to focus on taint propagation in this section, so other factors should be kept simple in order to illustrate the primary idea. Also, the implementation of multiple source boolean array as `taint information variable` is very similar to the boolean one, so there is no such big difference.

4.4.1 Binary Operators

In this subsection I will discuss taint propagation rule design for binary operator in this project. Because there is no operator overloads in JavaScript, the behaviors of binary operators are always certain.

Add

In JavaScript, there are only 2 behaviors for + operator: **numeric plus** and **string concatenation**. However, according to different operand types, the behavior of + varies. Of course, when 2 operands are both `Number`, the behavior is numeric plus, and when 2 operands are both `String`, the behavior is string concatenation. However, since JavaScript is a weakly-typed language, I also need to also consider the case other than these two, such as when 2 operands are both `Object`. To make things clear, I have written a script that shows the behavior of + for different types of operands:

```
function Test()
{
    this.t = 1;
}
var arr = [];
arr[0] = 1;
arr[2] = 4;
arr["a"] = 'b';
arr["c"] = new Test();
var typeVals = {str : "abc", numstr : "123", num : 123,
    undefined : undefined, null:null, bool : true, object : {a:1, b:1},
    objectTest: new Test(), arr : arr};

const print = (s)=>process.stdout.write(''+s);
const println = (s)=>print(s+'\n');

print('\"\",')
for (var t2 in typeVals)
{
    print('\"' + t2 + '\",');
}
println('');
for (var t1 in typeVals)
{
    print(t1 + ',');
    for (var t2 in typeVals)
    {
        var res = typeVals[t2] - typeVals[t1]
        var type = typeof res;
        print('\"' + type + ': ' + res + '\",');
    }
    println('');
}
```

Opening the output as `.csv` file, it can be clearly seen that if both operands are `number`, `undefined`, `null` or `boolean`, the result would be `number` type, so they can be regarded as numeric add; for other cases, since the result is `string` type, so we can regard them as string concatenation. Even though these edge cases can be handled, message will still be recorded into `results` as `log` record if any operand is weird.
For **Numeric Plus**, the rule is simple: the result is tainted if one of the operands are tainted, which is implemented in `rule.arithmetic` function. However, in some cases false positives may arise when both operands are tainted.

```
var tainted_int; //number-type tainted variable
var zero = tainted_int + (-tainted_int);
```

Cases like this is unavoidable with pure taint analysis (e.i. without symbolic execution), but fortunately such case rarely occurs. My approach to handle such situation is to record into `results` as `log` record if both operands are tainted in + operation and user chooses if this is recorded in configuration.
String Concatenation will happen not only when 2 operands are `string` type, but will also happen when they are array or object, which makes things complex. The approach to solve this is to implement a function called `getTaintArray`, which takes a value with any type as input and returns the corresponding taint array if that value is casted to `String`. There are several cases to consider:
String
If the value is string, just return its shadow value directly, since nothing will change if it is used as string.
Number, Boolean and Undefined
In current implementation, the method is easy: if the shadow value is `true`, which means the variable is tainted, then every character is tainted when it is casted to `String`; if the shadow value is `false`, which means the variable is untainted, then every character is untainted when it is casted to `String`. However, drawback exists in such design: the string is tainted even if the string cannot be completely controlled by the user who provides the input. In other word, we lose the information that the string can only be partly controlled by user. If we are using this taint analysis to perform automatic vulnerability detection, this will cause false positive if the goal is to detect vulnerability such as `XSS`. For example, consider the following code:

```
//variable `input` is tainted and can be controlled by attacker
var num = Number(input);
document.write("Age: " + num);
```

The `num` must be a `number` type, and attacker can control it by controlling variable `input`. When `num` is converted to `string`, every character will be tainted according to the rule described above, then it is passed into `document.write`. If the rule that is used to detect vulnerability is to report the vulnerability as long as data passed into `document.write` is tainted, the false positive will be reported in this case. The reason is that even if the string converted from variable `num` is tainted, attacker cannot have full control of the string: attacker can only control the character ranging from ‘0’ to ‘9’, so DOM-based XSS is not here. If we are using taint analysis to detect vulnerabilities, we may want to remove the taint as long as it is sanitized, or to use different taint value strategy such as `taint level` that has been discussed before.

Array

Before looking at how `getTaintArray` for `Array` type can be implemented, the behavior when `Array` is converted to `String` should be investigated first: when array is casted to string, every element is converted to string, and joined using `','`, for example

```
> String([{a:1},1,"test", true, [1,4]])
'[object Object],1,test,true,1,4'
```

`{a:1}`, `1`, `"test"`, `true` and `[1,4]` are converted to string, then joined with `","` as separator. Note that the conversion of `[1,4]` to string is done recursively using the same way as the conversion of outer array.

However, there are several special cases to note. *Firstly*, array can also has `string` variable as the property just like object.

```
> var a = []
undefined
> a[0] = 1
1 // assign to a integer index
> a[1] = 2
2 // assign to another integer index
> a[-1] = -2
-2 // assign to a negative integer
> a["key"] = "test"
'test' // assign to a string key
> a[a] = 5
5 // assign to a array key
> a[{}] = 'obj'
'obj' // assign to a object key
> a[10000000000000000000000000000000000000000000000000000000]='big'
'big' // assign to a large integer
> a[0.1] = 0.1
0.1 // assign to a floating point number
> a
[ 1,
  2,
  '-1': -2,
  key: 'test',
  '1,2': 5,
  '[object Object]': 'obj',
  '1e+55': 'big',
  '0.1': 0.1 ]
```

```
//except 0 and 1, everything else is converted to string
//before being used as property
> ''+a
'1,2'
//however, when casted to String,
//value inside string key will not be used
```

However, as shown clearly above, these value bounded with `string` key will not contribute when the array is converted to `string`, and any type other than positive small integer will be converted to `string` as key.
Secondly, `null`, `undefined` and circular reference will not be converted to string, but will be an empty string.

```
> var a = []
undefined
> a
[]
> a[0] = "first"
'first'
> a[2] = "2"
'2'
> a[3] = undefined
undefined
// assign undefined to index 3,
//also index 1 is not assigned so it's also undefined
> a[4] = "4th"
'4th'
> a[5] = null
null // assign null to index 5
> a[6] = 'six'
'six'
> a[7] = a //assign circular reference at index 7
[ 'first', , '2', undefined, '4th', null, 'six', [Circular] ]
> ''+a
'first,,2,,4th,,six,'
/*cast to string, obviously null, undefined and circular reference
is empty string*/
```

If the index is never assigned or has value `undefined`, `null` or circular reference, it will simply be converted to empty string. The way to define circular structure is when an element is the reference to any outter arrays, for example:

```
> a = []
[]
> a[0] = []
[]
```

```
> a[0][0] = a
// assign circular reference,
// although not direct circular reference
[ [ [Circular] ] ]
> a[0][1] = a[0] // assign direct circular reference
[ [ [Circular] ], [Circular] ]
> a
[ [ [Circular], [Circular] ] ]
// first circular is `a`, second circular is `a[0]`
```

Thirdly, `Array.prototype` should be noted. Assigning value to prototype is also a way to set the index of array, but this would work for all `Array` instances.

```
> var a = []
undefined
> Array.prototype
[]
> Array.prototype[0] = 1
1
> a.length
0 // array prototype will not make the array longer
> ''+a
''
// array prototype will not contribute
// when casted to string if the length <= index of prototype
> a[2] = 2
2
> a.length
3
> ''+a
'1,,2'
// however, prototype will contribute if length > index of prototype
```

Luckily, value in `prototype` will still contribute when an array is converted to string, but only if the index is smaller than `a.length`. Therefore, this does not affect our implementation so much.
Considering these factors, we can implement the function that obtain the taint array when an JavaScript `Array` is converted to string. We iterate over array using for loop bounded by `length`, convert elements to taint array by using recursion call if necessary, and `concat` them together; the '`,`' in between is always untainted.

Binary Arithmetic Operator

Binary arithmetic operators are operators like $-$, $*$, $/$ and $\%$. If both operands are numeric, dynamic taint analysis rule can work easily: the result is tainted as long as one of the operands is tainted. But for types other than `Number`, things become

hard to analyze. Therefore, to inform user, `JsTainter` would record the message into `results` as `log` type record when the type of operand is something other than number.
Without considering the case that `valueOf` has been overwritten, it is obvious that as long as one of the operands is `Object`, the result is always `NaN`, no matter how values inside the object changes. Therefore, as long as one of operands is object, the result should be untainted.
Another possible situation is when the operand is string. When `string` is used as operand of arithmetic operation, it will be converted to number first before doing arithmetic operation except +.

```
> "3"/10
0.3
> "3" - "10"
-7
> "3" + "10"
'310' // only + will perfrom string concat
```

The result should not be tainted if the string is always converted to `NaN` no matter how tainted characters changed. For example, `AnnotatedValue("a123", [false, true,true,true]) - 0` should be untainted, because no matter how last 3 characters are changed, the result will always be `NaN` due to the existence of ‘`a`’ at index 0; and `AnnotatedValue("1ea", [false,false,true]) - 0` should be tainted, because if last ‘`a`’ is changed to a number such as ‘`1`’, the left hand side operand can be interpreted as a floating point number in scientific form and the result will not be `NaN` but something user can control.
My approach to detect if the numeric result is actually controllable is to replace all tainted characters by number characters such as ‘`0`’, then try to convert the string to number. If it is still `NaN`, the result should not be tainted; if it becomes a number, the result should be tainted. The taint value is obtained through calling function `compressTaint` over the shadow values of the characters. However, even with such careful design, false positive could still come up. For example:

```
var b;
// `b` is a tainted variable, but is always boolean
var i;
// `i` is a tainted variable, but is always integer
var r = String(b) - i;
// characters in String(b) are all tainted
// `r` is always NaN no matter what `b` and `i` are
// but it will be marked as tainted
```

Case like this is unavoidable, unfortunately. Therefore, the design choice is record the information into `results` as `log` type record when the operand type is not number, as I suggested above.

Shift operator

Shift operators are <<, >> and >>>. The taint propagation rule for shift operator can also be similar to arithmetic ones: result is tainted if one of the operands are tainted (`rule.arithmetic` is used). However, special cases are a bit different:
1. The result is not `NaN` if there is any `NaN` among operands. Instead, the value that would be evaluated to `NaN` will be regarded as `0`. 2. If the left hand side is always evaluated to 0 (e.g. including `NaN` case), the result should be untainted since result is always `0` no matter how operand at right hand side changes.

```
> 123 << "asc"
123
> "asc" << 42
0
> 123 << {}
123
> 123 << [1,2]
123
// operand that will be casted to NaN will be regarded as 0
> 123 << [1]
246
```

Boolean Operator

There are two types of boolean operators: comparison such as ==, < and >; logic such as `&&` and `||`.
For comparison, the taint propagation rule is still same: taint the result if one of the operands is tainted. However, for expression like `tainted == tainted`, where `tainted` is a tainted integer, the result is always true no matter how variable `tainted` changes, but it will still be marked as tainted according the current rule design. Fortunately, situation like this rarely occurs in real program. Nonetheless, analyzer can still choose to record message into `result` as `log` record when both of operands of comparison are tainted.
For logic, Jalangi2 will simply treat it as conditional expression: therefore, these operators are not treated as binary operators but as conditional. They would not cause `analysis.binaryPre` and `analysis.binary` handlers to be called, but would cause `analysis.conditional` handler to be called. For example, `a && b` will be translated to **a ? b : a**, therefore it is not a binary operator at all.

Bit-wise Operator

As I suggested in last section, if bit-wise taint tracing is used, taint tracing will much more accurate here. However, since this is not necessary, such design is not employed. Instead, we treat it in the same way as comparison operators like ==. However, user can still choose to record message into `results` as `log` type record when tainted variable is used as operand of bit-wise operator.

4.4.2 Put and Get Field

Overview

Field getting is a JavaScript operation that obtains array element, object field, and character in string. Let's look at the behavior of **object** first.

```
> a = {}
{}
> a["qwer"] = 1
1 // put field
> a.qwer
1 //a["qwer"] is equavalent to a.qwer
> a
{ qwer: 1 }
```

As shown above, `a["xxx"]` and `a.xxx` both give same behavior. In addition, there are also some edge cases as always, for example:

```
> a = {}
{}
> a[1] = '1'
'1'
> a[{}] = 'obj'
'obj' // {} will be casted to string first before used as key
> a['1']
'1' //a['1'] is equavalent to a[1], for the same reason
> a[[[1]]]
'1' //[[1]] will be casted to string first, which is '1'
> a['0x1']
undefined //'0x1' cannot be casted to 1
> a
{ '1': '1', '[object Object]': 'obj' }
```

When the key is something other than string, it will be converted to string first, then used as the key to obtain the value.
The behavior of **array** is almost identical to object, except number keys are treated specially.

```
> a = []
[]
> a[0] = 0
0
> a[3] = 3
3
> a.length
4 // the length is dependent on largest index being assigned
```

```
> a["qwer"] = 's'
's' // assign using string as the key
> a[{}] = 'obj'
'obj' //{} will also be casted to string first
> a[[[["qwer",]],]]
's' //same, casted to string before used as key
> a['0']
0 //a['0'] is equivalent to a[0],
//which means numeric string will be casted to int
> a[new String('0')]
0 //new String('0') behave same as '0'
> a['0x0']
undefined // numeric hex string will not be converted to number
> a[[[[0]]]]
0
> a[[[['0']]]]
0
> a[NaN]
undefined
> a
[ 0, , , 3, qwer: 's', '[object Object]': 'obj' ]
```

Unlike some other languages, in JavaScript, there is no array-out-of-bound error when assigning the array with out-of-bound index; instead, the array is extended automatically. Also, we can also have string as the key in array, and things that is neither `String` nor `Number` will be converted to string automatically. However, `Number` type has privilege: as long as a string is a numeric integer string, it will be regarded as integer instead of string. In other word, there is no numeric string key in `Array`, because they will all be regarded as numeric index.
String is similar to array in how it handles the numeric string, but it does not have string as the key, and it remain unchanged when `putField` is applied.

```
> a = "qwer"
'qwer'
> a[1]
'w'
> a['1']
'w' // numeric string will be treated as integer
> a[[[['1']]]]
'w' // cast to string first before used as key
> a[4] = 'k'
'k'
> a[0] = 'k'
'k'
> a
'qwer'
```

```
> a['qwer']
undefined
```

For **other types**, they all return `undefined` and remain unchanged when `getField` and `putField` respectively. However, there are some built-in fields like `__proto__`, which also exist in 3 types covered above.

```
> x = 0x100
256
> x["test"] = 'test'
'test'
> x.test
undefined
> x.__proto__
[Number: 0]
```

Taint Analysis Rule

For `getField`, the current design is to return the same thing as the value corresponding to the key. Therefore, if the element being fetched is tainted, the result is tainted; otherwise, the result is not tainted. The rule is same for the case of `String` type, but the implementation is a bit different due to different structures of shadow value that is used to record taint state. For example, an tainted array `[1,2]` would have structure `AnnotatedValue([1,2], {"0":true, "1":true})`, while an tainted string `"12"` would have structure `AnnotatedValue("12", [true,true])`. If we fetch index 0 (e.i. `a[0]`), the array one should give `AnnotatedValue(1,true)` which can be obtained through directly accessing both shadow value and real value, while the string one should give `AnnotatedValue("1", [true])`, which is almost same except that `true` is putted into an array because the result of accessing a string index is still a string, although with length 1.
For `putField`, the current design is to directly assign the real value and shadow value to the corresponding key at both field of `AnnotatedValue`. For example, there is an empty object `obj = AnnotatedValue({}, {})`, and after operation `obj.a = x`, where `x` is a tainted variable `AnnotatedValue(1, true)`, the result of `obj` should be `AnnotatedValue({a:1}, {a:true})`.
However, such rules may give false negatives. For example, when `base64` is implemented in JavaScript, a index that might be tainted is used to access a constant `base64` array. The result should be tainted since information propagation exists, but if the taint state of index is ignored, the result will be untainted since the constant array itself is not tainted, which means false negatives. To mitigate such error, message will be recorded into `results` as `log` record if index is tainted, and user can also modify the configuration so that result of field fetching is tainted if either corresponding element or index is tainted.

4.4.3 Native Function Call

Native Function Recognition

The native functions are JavaScript built-in functions. These do not have to be functions defined in JavaScript standard, but can also be some environment dependent functions, such as DOM APIs. For example, `alert` is a built-in function when JavaScript is running in the browser, and `Number` is a built-in function defined in JavaScript standard that should works fine in all JavaScript implementations.
Therefore, I need to check if a particular function is a native function or user-defined function. After some investigation[13], I found that I can convert the function into string by built-in `Function.prototype.toString` function, and then check the result string. If the string is in the form like `"function funcName() { [native code] }"`, it is obvious that the function is a native function. We can check this by regular expression.

```
/function [a-zA-Z_][a-zA-Z0-9_]*\(\)[ \t\n]*\
\{[ \t\n]*\[native code\][ \t\n]*\}/
```

Native Function Call Handler

In `Jalangi2` framework, I can set `invokeFunPre` and `invokeFun` fields of `analysis class` as function handlers, and they will be called before and after any function call is made in the program being analyzed, respectively. In `invokeFunPre`, nothing is implemented, but set the `skip` field of return value as `true`. By setting this field, `Jalangi2` will not perform the function call anymore. This is the desired behavior because assigning the work to `Jalangi2` will give the wrong result since the arguments are wrapped by `AnnotatedValue`. Thus, instead, the function is called in `invokeFun` handler by `JsTainter`.
In `invokeFun` handler, we firstly check if `f` (the variable being called) equals to specific strings, if so, corresponding assertion function is called. This piece of codes is used for testing purpose, which will be discussed when evaluation strategy is covered, but this code will be removed in the final product. Then I try to check if the function is native function. If it falls into the category of native function, a big `switch` statement will be used to check which native function `f` is , and jump to the corresponding case handler, in which the taint propagation logic and actual function call are implemented for that particular native function. The pseudo code is shown below.

```
if (isNative(f))
{
    switch (f)
    {
        case String.prototype.substr:
        //....
        case String.prototype.charAt:
        //....
        //Other native funtions
    }
}
else
{
    //...
}
```

Function Call Implementation

Sometimes, native function would be called with `new`, such as `new XMLHttpRequest`. Therefore, when `isConstructor` argument is `true`, the function need to be called as constructor. Therefore using `.apply` is not appropriate. Since in `analysis.js`, there is already a function `callFun` being defined, in which constructor is also implemented, so we can modify codes of Jalangi2 and export this function by assigning it to field of `analysis` and call it in `analysis.invokeFun`.

```
//in analysis.js
sandbox.callFunExport = callFun;
```

```
//in invokeFun callback at DynTaintAnalysis.js
const callFun = sandbox.callFunExport;
```

Then, this `callFun` function can be used to execute the function call.

Native Function Rules

In this subsection I will cover the detail of handler for each different native function.
String.prototype.substr
The `substr` function, as its name suggests, takes the sub-string of given string. The first argument is the index and the second argument is length. However, there are some special cases.
Firstly, the input string does not have to be string type: if a variable other than string type is passed as the `this` argument, it will be converted into string first before applying `substr` operation, as shown below. I can use `getTaintArray` that we implemented before to get the taint array if the variable is converted into `String`, and this problem can be solved.

```
> String.prototype.substr.apply(123456789, [2,3])
'345' // 123456789 will be converted to '123456789' first
> String.prototype.substr.apply({}, [2,10])
'bject Obje' // {} will be converted to '[object Object]' first
> String.prototype.substr.apply([1,2,3,4,5,6,7,8], [2,3])
'2,3' // the array will be converted to '1,2,3,4,5,6,7,8' first
```

Secondly, index can be negative, and when it is negative, it will start from the end (e.i. `-x` has same effect as `length-x`). Also, index and length does not have to be `Number`; if they are not `Number`, they will be converted to `Number` first. When they are evaluated to `NaN`, they will be regarded as `0`.

```
> "abcdefghij".substr(-3,2)
'hi'
> "abcdefghij".substr("abcdefghij".length-3,2)
'hi'
//case of negative index
> "abcdefghij".substr(-100, 3)
'abc'
//but index will be regard as 0 if index < -length
> "abcdefghij".substr(NaN, 3)
'abc'
> "abcdefghij".substr({}, 3)
'abc'
//anything that will be converted to NaN will be regarded as 0
> "abcdefghij".substr([[[3]]], [3])
'def'
> "abcdefghij".substr('3', '3')
'def'
//anything that will be casted to numeric string
//will be regarded as number
```

When `substr` is handled, we must also slice the taint array in the same way as `substr`. For example, for a string `AnnotatedValue("AABB", [true,true,false, false])`, and `substr(1,2)` is applied, the result should be `AnnotatedValue("AB", [true,false])`. However, if we use `Array.prototype.slice` to slice the taint array of string in `substr` operation, we will have too many cases to consider. Thus, I came up with a quick and dirty way to implement it: the taint array can be converted to string first, then apply the `substr` with the same argument, and finally convert it back to taint array. I have implemented a function called `taintArrToStr`. In this function, for each character of the result string, the Unicode value is the index to the taint array, so what this function actually does is to generate a string with same length and characters with ascending values starting from `"\u0000"`. Another function that I have implemented is `strToTaintArr`. This function is called after `substr` is applied: the Unicode values are used as indexes to fetch the taint array elements, which are joined together to be the result. A piece of code may illustrate my idea better.

```
> var s = taintArrToStr([true, false, true, false])
> s
"\u0000\u0001\u0002\u0003"
> s = s.substr(1,1)
"\u0001\u0002"
> strToTaintArr(s, [true, false, true, false])
[false, true]
```

However, the disadvantage of this approach is that the maximum length of string cannot exceed 65536, the max value of numeric value of character, but fortunately string with such big size rarely occurs.

Number
This function convert variable to `Number`, we can just use `rule.compressTaint` to obtain the result taint if the variable will not always be evaluated to `NaN`. The logic is almost same in arithmetic taint propagation handler.
String.prototype.charAt
This function obtains the character at given index, which should return the same taint information as that of character at that index. For example, `charAt(1)` of `AnnotatedValue("ABCD", [true,false,true,true])` should be `AnnotatedValue("B", [false])`.
There are also cases where the index is tainted, but the characters in string are not tainted. In this case message is added into `results` as `log` record.
String.prototype.charCodeAt
This is similar to `chatAt`, except that the ascii value will be returned, so the taint information at that index will also be returned as shadow value. For the same example, `charCodeAt(1)` of `AnnotatedValue("ABCD", [true,false,true,true])` should be `AnnotatedValue(0x42, false)`.
String.fromCharCode
This function convert the integer to string of length 1 whose unicode value is same as the given index. For example, `String.fromCharCode(0x41)` will give `"A"`. The result taint information is same as the taint value of given argument. However, the given argument does not have to be `Number`. They can be some other types.

```
> String.fromCharCode([[0x41]])
'A'
> String.fromCharCode({a:0x41})
'\u0000'
> String.fromCharCode('0x41')
'A'
> String.fromCharCode(null)
'\u0000'
> String.fromCharCode('0x41\u0000')
'\u0000'
```

Therefore, we need to call `compressTaint` to the shadow value of given input. Also, if the argument always evaluates to `NaN` no matter what tainted parts are, the return value, which is `"\u0000"`, should not be tainted. For example `String.fromCharCode(AnnotatedValue("0x41\u0000", [true,true,true,true,false]))` should return `AnnotatedValue("\u0000", [false])`.
String.prototype.concat
This is almost same as the + operation as string concatenation. However, the difference is that this function can concatenate several strings together.

```
> "abc".concat("def", "ghi", "jk")
'abcdefghijk'
> "abc".concat({}, 123, [2,3])
'abc[object Object]1232,3'
```

Thus, we need to concatenate all taint array together, using `Array.prototype.concat`.
escape
This is the function that converts the string into URL encoding if necessary.

```
> escape("abc\"\'<>\ucccc")
'abc%22%27%3C%3E%uCCCC'
```

There are 3 cases: if the character does not have to be converted into URL encoding, it will be left unchanged; if the character need to be encoded but is smaller than `0x100`, it will be converted to `%XX`; if the character need to be encoded but is larger than `0x100`, it will be converted to `%uXXXX`. For the first case, the taint information is left unchanged; for the second case, the taint information is copied 3 times; and for the third case, the taint information is copied 6 times.
unescape
Function `unescape` converts the escaped URL-encoded string back into original string. There are also 3 cases, same as function `escape`. For unchanged characters, the taint information is also unchanged; for other 2 cases, the taint information is reduced with `or` operator. Nonetheless, it is vary rare for taint information within the encoded version of a character to be different (e.g. `AnnotatedValue("%41", [true,false,false])`), so a message will be recorded into `results` as `log` record in such case.
To check if a sequence of characters can be decoded, we must ensure all characters after it to be the correct format. For example, `"%qq"` should not be regarded as encoded character.
String.prototype.indexOf
Function `indexOf` is used to fetch the first appearance of a substring. Firstly, if the substring is tainted, the result is tainted; otherwise, the taint result is dependent on characters before the returned value plus length substring: taint states of these characters are reduced by `or` operation to be used as taint state of return value.

User Defined Function Call

For user defined function call, instead of calling the function using `callFun`, I have implemented another function called `callFunc`. This function perform same as `callFun`, except when function call is a constructor, the base object being passed into constructor is an `AnnotatedValue` object with shadow value {}. This is used to track taint flow when any argument passed into the constructor is tainted and it is used to assign field of the object, so that after object construction the field is correctly tainted.

4.4.4 Other Instrumentation

_with

`_with` is the instrumentation callback that will be executed when `with` statement is executed. Since currently, the design is to wrap the object inside an `AnnotatedValue`

object, if `with` is applied directly to an `AnnotatedValue` object, instead of extending the scope chain using the fields inside the original object, scope chain will be extended using fields of `AnnotatedValue` (e.i. `val` and `shadow`), which is incorrect. For example, we have an object `AnnotatedValue(a:1,b:2, a:true`, after being applied by `with`, scope being extended will be `val===a:1,b:2` and `shadow===a:true`, which not what we want because we actually want `a===AnnotatedValue(1,true)` and `b===2`. Therefore, we want to instrument the `with` statement and make it work like that.
According to the Jalangi2 documentation, we can modify the object being used by `with` statement by setting the `result` field of return value. Therefore, the desired effect can be implemented by creating an new object and setting the corresponding field to intended `AnnotatedValue` object, and enable `with` statement to use it by setting the `result` field of return value object. The instrumentation callback is shown below.

```
this._with = function (iid, val)
{
    var aval = actual(val); // get val field
    var sval = shadow(val); // get shadow field
    var ret = {};
    for (var k in aval)
    {
        //traverse all fields,
        //create correct AnnotatedValue object
        //with corresponding shadow value if needed,
        //assign it to the corresponding field of
        //the newly created object,
        ret[k] = getTaintResult(aval[k], sval[k]);
    }
    return {result:ret};
    //return that newly created object as the result
};
```

forinObject

Similar to `with` statement, if an object wrapped by `AnnotatedValue`, and a `for` loop is used to iterate the keys of the object (e.i. `for (var k in obj) ...`), the iteration will be wrong. The reason is very similar: `val` and `shadow` fields of `AnnotatedValue` will be iterated instead of fields inside keys in the original object. Using the same example, `AnnotatedValue(a:1,b:2, a:true)`, we want the iteration to be string keys `"a"` followed by `"b"`, but in reality `"val"` and `"shadow"` string keys will be iterated.
Therefore, we need to set `forinObject` instrumentation callback function to handle it. Since this operation only cares about the properties instead of values, we only need to return `val` field of `AnnotatedValue` because it is exactly the original

object but without shadow values, which must have the same properties. Thus the implementation is very simple, shown below.

```
this.forinObject = function (iid, val)
{
    return {result: actual(val)};
};
```

Chapter 5

Browser

5.1 Browser Integration Overview

In last chapter, I have discussed the dynamic taint analysis algorithm part of the project, and made it successfully run in `node.js`. However, the final product should be used to analyze the front-end client website instead of back-end server, so it is required to integrate the product into browser.

5.1.1 Setups

Originally, the goal of the project is to make a browser extension, but in the documentation of Jalangi2[11], it is suggested that `Jalangi2` is dependent on `mitmproxy`, which is a `pip` package. It is hard to find a counterpart of `mitmproxy` in browser extension and even if I found one I may need to modify the code of Jalangi2 to make it support proxy other than `mitmproxy`, which is beyond the scope of this project. Thus, to make it simple, rather than being a browser extension, the product is now proxy based.
According to the official example, I figured out the command to run the instrumentation and analysis:

```
mitmdump --quiet --anticache -s \
"jalangi2/scripts/proxy.py --inlineIID --inlineSource \
--analysis jalangi2/src/js/sample_analyses/ChainedAnalyses.js \
--analysis Utils.js --analysis Log.js --analysis TaintLogic.js \
--analysis Browser.js --analysis DynTaintAnalysis.js"
```

The `ChainedAnalyses.js` is a JavaScript file that chain the analysis together and files followed by it are my analysis files.

5.1.2 Browser.js

Different from `node.js`, there are many DOM APIs when JavaScript is run in browser, including some native functions and some global objects. I will especially take care about `Sources` and `Sinks`. `Sources` are where the user input can be obtained: for

example, user input can come from `window.location.hash`, which is part of the URL and can be controlled by user; and user input can also come from native function used to get input such as `prompt`, which pops a window and get the input from user. `Sinks` are some important APIs that analyzer might want to note about when its argument can be controlled by user: for example, web page can modify the HTML content of DOM object by setting field `.innerHTML`; web page can also use `XMLHttpRequest` to send package. If the contents being used for the operations are tainted, analyzer might be interested about such information, so `JsTainter` may need to record when a tainted variable is passed into them.
Therefore, the reason why this file exists is to write some browser specific code that processes the `Source` and `Sink` and to separate it from the dynamic taint analysis algorithm codes. The advantage for this is that I want my code to be still runnable on `node.js` for more convenient testing, and if we just simply put everything together (e.i. put the browser logic into `DynTaintAnalysis.js`), it will not only be bad software engineering design, but might also cause `ReferenceError` exception since global DOM object variable is not defined in `node.js`.
Nonetheless, in `DynTaintAnalysis.js`, the `dtaBrowser` field of `J$`, which is used to export `Browser` object, will still be accessed, so my way to solve this problem is to have a `NullBrowser` object for `node.js` that basically perform nothing except the methods are defined so no `Error` will be thrown. To be specific, in the command line of running this in `node.js`, `--analysis Browser.js` is changed to `--analysis NullBrowser.js`. This piece of code may illustrate the idea better.

```
//Browser.js
(function (sandbox) {
function Browser(){}
Browser.prototype.getField = function (base, offset)
{
    //process DOM global object get field operation...
    //e.g. window.location.hash
};
Browser.prototype.invokeFun = function (f, abase, args)
{
    //process DOM native function call...
    //e.g. prompt
};
//process other DOM-related operations...
sandbox.dtaBrowser = new Browser();
})(J$);

//NullBrowser.js
(function (sandbox) {
function Browser(){}
Browser.prototype.getField = function () {};
Browser.prototype.invokeFun = function () {};
//other DOM-related operations are also dummy functions...
sandbox.dtaBrowser = new Browser();
})(J$);
```

When the relative taint analysis is performed, the method in `Browser` will be called first, and if it returns `undefined`, normal handling will be performed; if not, which means the operation is DOM-related and has already been handled, so analysis function will return directly. For example, here is how `dtaBrowser.getField` is called in `analysis.getField`.

```
ret = sandbox.dtaBrowser.getField(abase, aoff);
if (typeof ret != 'undefined')
{
    //ret is the result, if is not undefined
    return new AnnotatedValue(ret.ret, ret.sv);
}
else
{
    // normal getField handling
}
```

The reason why `dtaBrowser.getField` does not return a `AnnotatedValue` object is that I do not want `Browser.js` to be dependent on `AnnotatedValue` class, which is implemented in `DynTaintAnalysis.js`.
For sinks, everything is same except that only record will be added to `results`, but the function will not be returned.

```
if (isTainted(sval))
{
    const ret = sandbox.dtaBrowser.putField(
        abase, aoff, aval, sval, config);
    if (typeof ret !== 'undefined')
    {
        addLogRec(this, getPosition(iid), ret.msg);
    }
}
```

5.1.3 Multiple Sources Implementation

As I have mentioned in last chapter, we can use an array of boolean variable to represent multiple source, which can be further optimized to a number. In browser, this becomes more important, since user inputs can come from different sources. `MultSrcTaintLogic.js` is the file that implements multiple source taint propagation. The logic is almost same except `||` is changed to `|`, because now number is used to represent a boolean array, and bitwise `or` is same as applying `or` to every corresponding element of the boolean array. Another difference is `taintSource` function used to obtain the initial `taint information variable` for source, instead of just returning true, `id` argument is used as the shift amount. The code is shown below.

```
TaintUnit.prototype.taintSource = function (id)
{
    return 1 << id;
};
```

When this function is called, `id` must be different for different sources. For example, in my implementation, `id` is 1 for URL string.

ID Allocator

However, even if the type of input is same, the taint `id` should still be different sometimes. For example, when `prompt` function is called multiple times, the `id` should be different for each time in order to specify which `prompt` a particular taint comes from. Therefore, an `id` allocator is required.
My approach to implement allocator is very easy, the code is shown below.

```
//Implemented in class Browser
var nextId = 1;
this.getNextId = function ()
{
    if (nextId >= 32)
        throw Error("Too many input sources");
    return nextId++; // return nextId, and increment it
};
```

Since JavaScript shift operator only support 32 bit integer (e.i. because `1 << 32 === 1`), the maximum size of this boolean array represented in integer form is 32, and an `Error` is thrown if there are too many `id`s being allocated.

5.2 Source and Sink

In this section, I will discuss different types of possible source and sink in browser, and my approach to handle them.

5.2.1 Source

window.location

As I suggests in last section, this stands for URL of web page. However, there are several different fields in this object.
Field `search` is the query string begin with `?`, and field `hash` is the fragment string begin with `#`. These two are all parts of the URL and can by controlled by user, thus all of characters in string should be tainted, which means all elements in shadow value array should set to return value from `taintSource`. However, there are some special cases.

Field pathname is the path of the URL, which can be both tainted and untainted: when static path is used, the pathname should be untainted because content of page will change if path is modified; when path is used in the same way as query string, pathname should be tainted because user can control this string without going to another page. Therefore, since this is dependent on different website implementation, I should enable user to manually set this: there is a field in config variable that is used to specify if pathname should be tainted.
Field href is the whole URL, which obviously cannot be fully controlled by user: user can control query string and fragment only, and can control path if corresponding config field is set as I mentioned before. Therefore, only part of the string is tainted (set to return value from taintSource) and remaining string is untainted (set to value 0). My approach to identify the boundary is to use indexOf function as shown.

```
const start = config.taintPathName ?
    base.href.indexOf(base.origin) + base.origin.length :
    base.href.indexOf('?');
```

If taintPathName is set to true, the tainted part starts after base.origin, which is start of the path; if not, the tainted part starts after the first ?, which is start of the query string.

prompt

Since the string returned from prompt function can be fully controlled by user, all characters in string should be tainted. However, when user click "cancel", null will be returned and it will not be marked as tainted, since in current rule null cannot be tainted.

HTMLInputElement

HTMLInputElement is the HTML input tag, for example <input type="text" id="myText">. JavaScript program can access the content in this input tag by using document.getElementById("myText").value, which should all be tainted as it can be fully controlled by user.
Store Previously Allocated ID
For different input tags, different id values are used. However, for same input tag, the id should be same. Therefore, we need a recording that maps the input tag DOM object to previously allocated id value, if any. Since JavaScript object only supports string as property, we cannot use object to implement the map. Therefore, an array is used to represent such information. I have implemented a getInputId function: given a input tag DOM object, return the corresponding id, which can be previously recorded or newly allocated, depending on if this object is found in that recording array.

```
var inputsArr = [];
this.getInputId = function (input)
{// `input` is DOM object of input tag,
    //e.g. document.getElementById("myText")
    for (var i = 0; i < inputsArr; i++)
    {
        if (inputsArr[i].input === input)
            return inputsArr[i].id; // return the id, if found
    }
    const nextId = this.getNextId();
    inputsArr.push({input:input, id:nextId});
    return nextId; // allocate and record new id, if not found
};
```

However, there are drawbacks for this approach: the complexity is `O(n)`. Another approach could be replacing `inputsArr` with an object, which uses `id` field of DOM object as the property that maps to previously allocated `id` value (e.g. `{myText: 1}`). However, even `id` for each DOM object should be unique according to the specification, a HTML page does not have to conform the standard, which might cause problem. By contrast, the `O(n)` complexity does not hurt so much because there would not be as many input tags in a HTML page.

5.2.2 Sink

There are two types of sinks. The first one is when field of global object is set, and the second one is native function that deserve notice. Although there are still many types of sinks, the way to handle them is exactly same: record the 'log' type record. Thus I will only briefly mention this.

Global Object Field Set

This include changing `window.location.href` and `document.cookie`. When they are modified to something tainted, message is recorded as `sink` type record.

Native Function Sink

This includes functions like `document.write`, the only thing to do is simply record as `sink` type record.

Chapter 6

Evaluation

6.1 Testing

To have proper software engineering design, I have written test cases to ensure the analysis is running correctly. However, the approach to test is somewhat different from normal program. Since the `JsTainter` heavily relies on `Jalangi2` framework, it's quite hard to simply import the taint analysis unit and perform unit testing only on that unit. The reason is that `Jalangi2` has done many things for `JsTainter`, and `JsTainter` does not work without this framework.
Therefore, instead, I have formulated a way to perform testing. Since we can instrument the JavaScript program that the analysis is running on, we can instrument on function call to perform our assertion.

6.1.1 Checking Taint State

```
var a; //something whose taint state is to be examined
const assertTaint = "assertTaint";
assertTaint(a, true);
```

The codes above will simply throw an exception in normal execution. However, if it is instrumented by our analysis program, we can examine the value of `function` parameter in the function call instrumentation callback. This value should be a `function` type variable in normal case, but if it is a `string` type variable and has value `"assertTaint"`, then we know we are going to perform assertion against the taint state of given variable, instead of executing the function call that will throw the error.
In `assertTaint` function, the main goal is to check if `shadow(val)` (actual shadow value) is same as `taint` (expected shadow value). If they are not exactly same, assertion will fail. Variable `position` is the position of instruction that will be printed if assertion fails, which makes debug more convenient.

```
function myAssert(b, position)
{
    if (b !== true)
    {
        Log.log("Assertion failure at" + JSON.stringify(position));
        assert(false);
    }
}
function assertTaint(val, taint, position)
{
    taint = actual(taint);
    // taint might be wrapped by AnnotatedValue, just in case
    const s = shadow(val);
    myAssert(typeof s === typeof taint, position);
    // type must be identical
    if (Array.isArray(s))
    {// if shadow value is array, all elements must be same
        myAssert(s.length === taint.length, position);
        for (var i = 0; i < s.length; i++)
        {
            myAssert(s[i] === taint[i], position);
        }
    }
    else
    {// for any other cases such as basic-type case,
    //shadow must be equal
        myAssert(s === taint, position);
    }
}

//in the instrumentation callback handler of function call
if (f === 'assertTaint')
{
    assertTaint(args[0], args[1], getPosition(iid));
}
```

Note that these 2 pieces of codes above are in different files. The first code piece is in the JavaScript file that is going to be analyzed (e.i. `test.js`); while the second code piece is in the file that performs the dynamic taint analysis (e.i. `DynTaintAnalysis.js`) . Therefore, even if we have same `assertTaint` name as identifier in both files, there will not be any conflict.

6.1.2 Checking Real Value

Using the similar technique, real value of variable can also be checked. `"assert"` can be used to examine the correctness of real value of variable, and it is handled in the same way as "assertTaint".

```
else if (f === "assert")
{
    myAssert(actual(args[0]), getPosition(iid));
    return {result : undefined};
}
```

The usage of assert is a little bit different from assertTaint. Because real value can be directly access by program that is being analyzed, comparison can be done in the JavaScript program. For example,

```
const assert = "assert";
assert(a == 1);
```

6.1.3 Evaluation on Basic Test Cases

Implementation

To test the correctness of JsTainter, I have written many test cases in directory tests/. This will be tested by a simple Python script.

```
from os import system,walk
from re import search
cmd = "node jalangi2/src/js/commands/jalangi.js" + \
" --inlineIID --inlineSource" + \
" --analysis jalangi2/src/js/sample_analyses/ChainedAnalyses.js" + \
" --analysis Utils.js --analysis Log.js" + \
" --analysis TaintLogic.js --analysis NullBrowser.js" + \
" --analysis DefaultConfig.js --analysis DynTaintAnalysis.js tests/%s"

i = 0
for root,subdirs,files in walk("./tests/"):
    i += 1
assert i == 1

for f in files:
    ret = search("^test[a-zA-Z0-9]+\\.js$", f)
    if ret: # iterate file with format testxxx.js
        print "Testing file: " + f
        ret = system(cmd % f) # execute analysis
        if ret != 0:
            print "Error in file %s" % f
            exit(-1)
```

The reason why regular expression is used to filter file name is that Jalangi2 will generate some temporary files in that directory when analysis is performed, such as testxxx_jalangi_.js and testxxx_jalangi_.json, and only files with correct file name format should be analyzed and tested.

Tests

There are many test cases, and I will discuss them one by one.

`testarith.js` is used to test the taint propagation of arithmetic operation, especially when one of the operand is tainted string. For example, `(taintedStr + '123' + taintedStr) * 7` should be tainted because the result of this operation can be affected by `taintedStr` if it is a numeric string; while `(taintedStr + '0x' + taintedStr) * 7` should not be tainted because it always gives `NaN` no matter how `taintedStr` changes.

`testarithAdd.js` is used to test correctness of taint propagation of `add` operation.

`testbitoper.js` and `testshift.js` are used to test correctness of taint propagation of bit-wise operation, cases that operands are types other than number are also considered here.

`testcharAt.js`, `testindexOf.js` and `testsubstr.js` are used to test correctness of taint propagation of function `String.prototype.charAt`, `String.prototype.indexOf` and `String.prototype.substr` respectively, cases that arguments are types other than number are also considered here.

`testconstructor.js` is used to test taint propagation in JavaScript class. For example, when argument passed to constructor is tainted and is used to initialize the member fields, the fields should also be tainted. Also, `with` statement is also tested here.

`testeval.js` is used to test `eval` statement. In other word, taint propagation must also works well even if the statement that causes the taint propagation is executed using `eval`.

`testException.js` is used to test the case that when a tainted variable is thrown, the `catch` statement that receive the variable being thrown must also get the tainted value.

`testfield.js` is used to test correctness of taint propagation of putting field and getting field.

`testforinObject.js` is used to test correctness of `for in object` loop, which are properly handled by `analysis.forinObject` instrumentation callback function.

`testfunc.js` is used to test non-constructor function call, including anonymous function.

`testNumber.js` is used to test `Number` function. For example, when tainted string is casted to number by `Number` function, the return value should be tainted if it is controllable by the tainted argument.

`testConcat.js` is used to test string concatenation by operator `+`.

6.2 Evaluation on Website

In this section I am going to evaluate my analysis using web JavaScript program instead of `node.js` program.

6.2.1 Simple Example

I have written a simple website that can be used to evaluate the effectiveness of taint analysis over JavaScript program with multiple sources and sinks, shown below.

```
<div id="sth"></div>
<script type="text/javascript">
function myclick()
{
    const url = window.location.href;
    const idx = url.indexOf('#')
    var hello, n;
    if (idx === -1)
    {
        hello = "";
    }
    else
    {
        n = url.substr(idx + 1);
        const desc = prompt("Please input the description: ");
        hello = "name:" + unescape(n) + " desc:" + desc;
    }
    const num1 = document.getElementById("text1").value;
    const num2 = document.getElementById("text2").value;
    const sum = Number(num1) + Number(num2);
    hello += " sum:"
    hello += sum
    document.getElementById("sth").innerHTML = hello;
    const req = new XMLHttpRequest();
    req.open("POST", window.location.origin + '/' + n);
    req.send(sum.toString())
}
</script>
<form>
    <input type="text" id="text1">
    <input type="text" id="text2">
    <input type="button" value="click me" onclick="myclick()">
</form>
```

The result will be printed in `console` dialog, which is a list of JSON strings. In the following section, I will try to explain the result of the taint analysis and compare the result with real behavior of the JavaScript program, so that effectiveness of taint analysis can be evaluated.
Firstly, URL is fetched, which is partially tainted, and written to variable `url`. The `id` of this source is 0, so `taint information variable` for tainted character is `1 << 0 == 1` (`number` type is used to implement boolean array). This behavior is recorded properly.

```
{"type":"source","typeOf":"string","file":
"91ea3bc2825abb590247bbfa10e8631f.js","pos":[4,17,4,37],"name":"href",
"id":0}
{"type":"write","typeOf":"string","file":
"91ea3bc2825abb590247bbfa10e8631f.js","pos":[4,17,4,37],"name":"url"}
```

The file name is a MD5 hash, which is generated by Jalangi2, and is indeed a bit weird. However, we only need to know that the file represents the JavaScript code in the `<script>` tag shown above. The value in this property is same for all JSON results, so to make the report more clear, I will delete this field in following section, but actually they still exist.
Then, `indexOf` is called on `url` and the return value is tainted, which is then written to variable `idx`. This is intended result because `idx` can vary as the user input changes. For example, since there could be a query string before the first `#`, and its length is dependent on user, which means the value of `idx` can be controlled by user and thus should be tainted.

```
{"type":"read","typeOf":"string","pos":[5,17,5,20],"name":"url"}
{"type":"write","typeOf":"number","pos":[5,17,5,33],"name":"idx"}
```

Then, variable `idx` is read and used in a `if` condition, and corresponding `log` message is produced, which again is the intended result.

```
{"type":"read","typeOf":"number","pos":[7,9,7,12],"name":"idx"}
{"type":"log","pos":[7,9,7,19],"msg":
"Tainted variable false being used in conditional"}
```

Then the program goes into the `else` branch. Function `substr` function is called using `url` and `idx`, and its return value, which is tainted, is assigned to variable `n`. The corresponding results are still correct.

```
{"type":"read","typeOf":"string","pos":[13,13,13,16],"name":"url"}
{"type":"read","typeOf":"number","pos":[13,24,13,27],"name":"idx"}
{"type":"write","typeOf":"string","pos":[13,13,13,32],"name":"n"}
```

At line 14, input is obtained from `prompt` function again and assigned to variable `desc`, and at line 15, tainted variable `n` and `desc` are used to generate a partially tainted string, which is assigned to variable `hello`. The `id` of this source is 1, so `taint information variable` for every character is `1 << 1 == 2`. These behaviors are all recorded properly.

```
{"type":"source","typeOf":"string","pos":[14,22,14,62],"name":"prompt",
"id":1}
{"type":"write","typeOf":"string","pos":[14,22,14,62],"name":"desc"}
{"type":"read","typeOf":"string","pos":[15,36,15,37],"name":"n"}
{"type":"read","typeOf":"string","pos":[15,52,15,56],"name":"desc"}
{"type":"write","typeOf":"string","pos":[15,17,15,56],"name":"hello"}
```

After the else block, input string is obtained from 2 input tags, and assigned to variable num1 and variable num2. There are also log type JSONs that has been recorded, since getElementById native function is not handled in invokeFun handler; but actually this function does not need to be handled, so this JSON can be ignored. In addition, even if the inputs are both from <input> field, the id numbers being allocated are different, thanks to the id allocator.

```
{"type":"log","pos":[17,18,17,50],"msg":
"Unhandled native function getElementById"}
{"type":"source","typeOf":"string","pos":[17,18,17,56],"name":"value",
"id":2}
{"type":"write","typeOf":"string","pos":[17,18,17,56],"name":"num1"}
{"type":"log","pos":[18,18,18,50],"msg":
"Unhandled native function getElementById"}
{"type":"source","typeOf":"string","pos":[18,18,18,56],"name":"value",
"id":3}
{"type":"write","typeOf":"string","pos":[18,18,18,56],"name":"num2"}
```

The num1 and num2 are converted to number, and used to calculate sum, which is tainted and is concatenated to string hello in next step. Note that hello += sth is identical to hello = hello + sth, so a read record on hello variable will also be recorded, which is not a mistake.

```
{"type":"read","typeOf":"string","pos":[19,24,19,28],"name":"num1"}
{"type":"read","typeOf":"string","pos":[19,39,19,43],"name":"num2"}
{"type":"write","typeOf":"number","pos":[19,17,19,44],"name":"sum"}
{"type":"read","typeOf":"string","pos":[20,5,20,10],"name":"hello"}
{"type":"write","typeOf":"string","pos":[20,5,20,21],"name":"hello"}
{"type":"read","typeOf":"string","pos":[21,5,21,10],"name":"hello"}
{"type":"read","typeOf":"number","pos":[21,14,21,17],"name":"sum"}
{"type":"write","typeOf":"string","pos":[21,5,21,17],"name":"hello"}
```

Here is the first sink being detected: variable hello is written to innerHTML field of a <div> DOM object. The value being written to the sink and corresponding shadow value are presented in JSON.
The 4 1s correspond to "2019", which comes from source with id==0, URL; the 3 2s correspond to "AAA", which comes from source with id==1, return value of prompt. The 2 12s are interesting: since 24 results from adding values from 2 input tags, it can be affected by both source with id==2 and source with id==3, and 12 is the result from (1<<2) | (1<<3) == 4 | 8 == 12.

```
{"type":"log","pos":[22,5,22,35],"msg":
"Unhandled native function getElementById"}
{"type":"read","typeOf":"string","pos":[22,48,22,53],"name":"hello"}
{"type":"sink","pos":[22,5,22,53],"value":"name:2019 desc:AAA sum:24",
"shadow":[0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,2,2,2,0,0,0,0,0,12,12],
"name":"[object HTMLDivElement].innerHTML"}
```

Then, to test native function sink, `XMLHttpRequest` is used. The result is also correct.

```
{"type":"log","pos":[23,17,23,37],
"msg":"Unhandled native function XMLHttpRequest"}
{"type":"read","typeOf":"string","pos":[24,53,24,54],"name":"n"}
{"type":"sink","pos":[24,5,24,55],
"value":["POST","https://www.doc.ic.ac.uk/2019"],
"shadow":[[0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
1,1,1,1]],
"name":"open"}
{"type":"read","typeOf":"number","pos":[25,14,25,17],"name":"sum"}
{"type":"sink","pos":[25,5,25,29],"value":["24"],"shadow":[[12,12]],
"name":"send"}
```

6.3 Weakness

6.3.1 Implicit Flow

To detect implicit information flow, JavaScript must be analyzed from high-level perspective. However, since I have applied pure dynamic analysis, analysis can only be performed for each individual JavaScript operation. Therefore, automatic detection of implicit flow is not possible.

6.3.2 Unable to Track Taint of Native Object Fields

In JavaScript, there are some native objects. For example, `Error` is the object that is used to throw an exception, and can be used in this way:

```
try
{
    throw new Error("some message");
}
catch (e)
{
    console.log(e.message);
}
```

There might be cases that the string passed into `Error` is tainted. However, unlike non-native classes, Jalangi2 cannot instrument into constructor of `Error`, thus unable to tackle the taint state of `message` field. Thus, false negative would be produced.

6.3.3 Unable to Execute Codes with Lambda Function

This is actually a problem from Jalangi2 instead of JsTainter. In Jalangi2, lambda expression is wrongly instrumented: the parameters of lambda expression are instrumented like variable, which causes JavaScript grammar to be wrong. For example, when expression `const lambda = (a,b) => a+b;` is instrumented, `(a,b)` part would be instrumented to `(J$.R(81, 'a', 'a', 1), J$.R(89, 'b', 'b', 1))`, which is certainly wrong because this is not variable read and should not be modified, just like `(a,b)` part in `function (a,b) return a+b`.

6.3.4 Detectable by Program being Analyzed

For some JavaScript programs, anti-debug techniques are applied to prevent people from reverse engineering the product. For example, JavaScript program can convert function to string and check if the function is modified.

```
function some_func()
{ /*code that does not want to be modified by reverse engineer*/ }
const correct_crc = 0x708D2F22; //crc32 value of String(some_func)
function crc32(str) { /*code that implements crc32 hash algorithm*/ }
if (crc32(String(some_func)) != correct_crc)
    throw Error("Hack3r detected!")
```

If `some_func` function is instrumented by Jalangi2, the CRC-32 value will a become different one, so an exception will be thrown, which means the behavior of the program becomes different after instrumentation. This is not desirable.

6.3.5 Unhandled Prototype Poisoning

In current implementation, prototype poisoning is not properly handled. For example, behavior of field setting can be modified to particular function by using

```
Object.defineProperty(SomeClass.prototype, "key",
{set:function(){console.log("1337")}})
```

After this statement is executed, if `obj` is an instance of `SomeClass`, and `obj["key"]=1` is executed, instead of executing normal field setting, `function()console.log("1337")` will be executed, so `obj["key"]` will still be `undefined`. In this case, tracking the shadow value of the object in the normal way might cause inaccuracy.

Chapter 7

Conclusion

In this project, dynamic taint analysis over JavaScript program is implemented. I have thoroughly investigated the basic propagation rules of dynamic taint analysis not only for different JavaScript operations but also for general program. Also, I have also investigated behavior of JavaScript as a programming language.

7.1 Future Work

Firstly, the output of the analysis is not user-friendly. Therefore, instead of JSON, it is better to visualize or even animate the taint propagation, which makes the taint flow very clear.
Secondly, the drawbacks mentioned in Evaluation chapter could be tried to handled. For example, the prototype poisoning can be handled by handling operation that will modifies the prototype.
Thirdly, I can tried to implement different types of `taint information variable` design. For example, the taint level and symbolic execution can be implemented. Also, some additional feature can also be tried, such as automatic detection of client-side vulnerability like XSS.

Chapter 8

User Guide

8.1 Installation

Firstly, clone the repository.

```
git clone https://github.com/Mem2019/JsTainter
cd JsTainter
git clone https://github.com/Mem2019/jalangi2
```

Then, install Jalangi2 according to the instruction from Jalangi2 repository.

8.2 Usage

In JsTainter directory, type the following command to run the instrumentation proxy.

```
./run.sh
```

Finally, change the proxy setting of browser to `127.0.0.1:8080`, then enter the URL of website to start analysis. After analysis, click the `Jalangi2` button at the left hand side, and inspect console of developer tools for the result of analysis.
You may also edit the configuration file `DefaultConfig.js`, to customize the behavior of taint analysis.

Bibliography

[1] A javascript ctf challenge that uses *eval* expression. https://github.com/EmpireCTF/ctfdojo/blob/master/src/chal/js/A04.js. pages 5

[2] javascript eval() and security. https://stackoverflow.com/questions/39058482/javascript-eval-and-security. pages 6

[3] my previous reflection about taint analysis when preparing this project. https://mem2019.github.io/jekyll/update/2019/01/26/Taint-Analysis-Reflection.html. pages 7

[4] Shadow value. https://mem2019.github.io/jekyll/update/2019/04/26/Jalangi2-Shadow-Value.html. pages v, 19

[5] David Brumley Edward J. Schwartz, Thanassis Avgerinos. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. *2010 IEEE Symposium on Security and Privacy*. pages 7, 8, 12, 30

[6] Indradeep Ghosh Guodong Li, Esben Andreasen . Symjs: Automatic symbolic testing of javascript web applications. *FSE 2014 Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459. pages 12

[7] Franziska Hinkelmann. Understanding v8's bytecode. https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775. pages 9

[8] Intel. Pin tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. pages 16

[9] Professor Paul Kelly. Slides from imperial college london department of computing 221. https://www.imperial.ac.uk/computing/current-students/courses/221/. pages 12, 13

[10] Tasneem Brutch Simon Gibbs Koushik Sen, Swaroop Kalasapur. Jalangi: a selective record-replay and dynamic analysis framework for javascript. *ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498. pages 19, 20

[11] Samsung. Jalangi2. https://github.com/Samsung/jalangi2. pages 16, 60

[12] Manu Sridharan. Jalangi2 tutorial. `https://manu.sridharan.net/files/JalangiTutorial.pdf`. pages 17, 19

[13] David Walsh. Detect if a function is native code with javascript. `https://davidwalsh.name/detect-native-function`. pages 53

[14] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. *ISSTA 2013 Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346. pages 5, 6, 13, 14

[15] Google Project Zero. Afl fuzzer. `http://lcamtuf.coredump.cx/afl/`. pages 16