Spring 2023

Static Taint Analysis via Type-checking in TypeScript

Abhijn Chadalawada
San Jose State University

Static Taint Analysis via Type-checking in TypeScript

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Abhijn Chadalawada

May 2023

The Designated Project Committee Approves the Project Titled

Static Taint Analysis via Type-checking in TypeScript

by

Abhijn Chadalawada

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Prof. Thomas Austin	Department of Computer Science
Prof. Chris Pollett	Department of Computer Science
Prof. Ben Reed	Department of Computer Science

ABSTRACT

Static Taint Analysis via Type-checking in TypeScript

by Abhijn Chadalawada

With the widespread use of web applications across the globe, and the advancements in web technologies in recent years, these applications have grown more ubiquitous and sophisticated than ever before. Modern web applications face the constant threat of numerous web security risks given their presence on the internet and the massive influx of data from external sources. This paper presents a novel method for analyzing taint through type-checking and applies it to web applications in the context of preventing online security threats. The taint analysis technique is implemented in TypeScript using its built-in type-checking features, and then integrated into a web application developed using the React web framework. This web application is then validated against different types of injection attacks.

The results of the validation show that taint analysis is an effective means to prevent pervasive online attacks, such as eval injection, cross-site scripting (XSS), and SQL injection in web applications. Considering that our proposed taint analysis technique can be implemented using existing type-checking features of TypeScript, it can be quickly adopted by developers to add taint analysis into their applications with no performance overhead. With the large number of web applications developed in TypeScript, the widespread adoption of our technique can help prevent cyberattacks and protect the online community from potential harm. By combining taint analysis with other secure web practices, such as input validation, application developers can strengthen the overall security of web applications.

ACKNOWLEDGMENTS

I am deeply grateful to Prof. Thomas Austin for his invaluable support and guidance throughout this project as my advisor. His continued encouragement and insightful feedback have been instrumental in shaping this work. I would also like to extend my sincere appreciation to Prof. Chris Pollett and Prof. Ben Reed for their time and effort in serving on my project committee, and for their valuable input and feedback. Lastly, I would also like to thank my family and friends for their unwavering support and encouragement throughout the research and writing process.

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

Introduction

1.1 Domain and Context

In recent years, the internet and web technologies have grown at a rapid pace (both qualitatively and quantitatively) and still continue to grow even further, with no signs of stopping or slowing down. At the forefront of this expansion lie web applications (web apps for short), the evolution of which has been a significant driving force behind this development.

What used to be a niche space—home to labyrinthian websites that were tedious to navigate and used by only the most savvy—has now become an accessible hub for a plethora of services and resources encompassing everything from commerce to academic texts to entertainment. With Amazon, Google and Youtube becoming household names, it is apparent that web apps are now real world tools with pragmatic applications. They are also (mostly) simple and easy to use by anyone, anytime, anywhere.

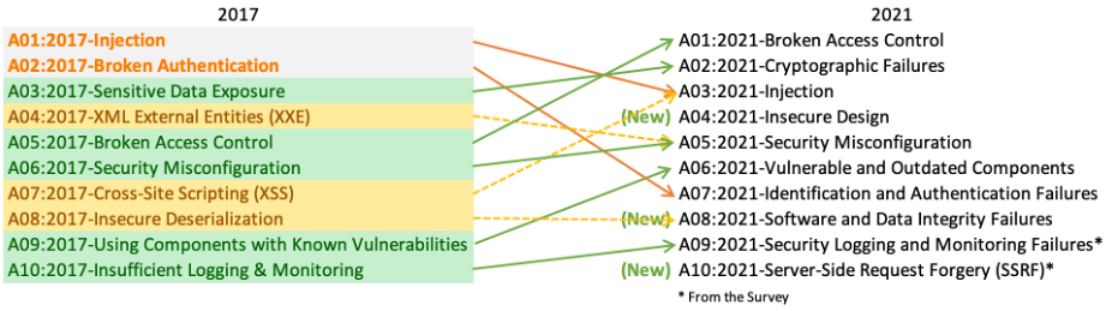
This transformation is a result of numerous advancements in web technologies, such as—improvements in Graphical User Interfaces (GUIs) leading to improved user accessibility, and the introduction of web scripting languages (PHP, JavaScript, Ruby, Perl, etc.) and automation of boilerplate web development processes by web frameworks (Bootstrap, Angular, React, VueJS, etc.) leading to improved developer accessibility.

The improved developer accessibility led to a rapid production of new web apps, which in turn led to more web users, further encouraging the production of more web apps to tap into this booming web market. This feedback loop has resulted in today's internet landscape of an abundance of web apps and a majority of the global population coming online to access these apps [1].

1.2 The Problem

However, this growth comes with its own challenges. With more users now on the internet than ever before, web security is all the more crucial to protect against the ever-rising threat of cyberattacks. Malicious individuals and organizations are always seeking to steal application and user data, or even disrupt critical web services. One particular class of attacks, known as injection attacks, is a very prevalent security threat today. The Open Worldwide Application Security Project (OWASP)—a leading authority on web application security—lists injection attacks as the third most critical security risk among the top 10 web application security risks alongside other risks including broken access control and cryptographic failures [2].

Figure 1: Top 10 web security risks from 2017 to 2021 according to OWASP [2]



Injection attacks are carried out by exploiting vulnerabilities in web application code to 'inject' malicious code into the application to perform harmful operations on behalf of the attacker. For instance, web applications gather input data from the end user in many ways, including but not limited to—web requests (e.g., GET and POST requests), and web forms explicitly seeking user input. If this input data is insufficiently validated (i.e., verified to not contain malicious code) before being processed, the application may be susceptible to injection attacks as attackers can try to inject malicious code—masquerading as input data—into the application.

A simple, yet effective preventative measure to guard against injection attacks

involves proper input validation. By thoroughly validating all the input data entering the application and ensuring that they are free from potentially harmful code, we effectively eliminate some of the most commonly used points of entry for attackers. However, ensuring thorough input validation for web applications is easier said than done. The ease of implementing such validation depends on various factors like—size, type and number of inputs, complexity of the application, the programming language and web framework used. It also involves additional effort in the design and development processes, requiring developers to track numerous inputs and their flow within the application as they are being processed.

This problem of tracking the flow of incoming data to an application is not unique to web applications. It has been extensively explored and studied in the domain of information flow analysis. As the name implies, information flow analysis is a technique used to evaluate the flow of information within the system—from sources (where data is produced) to sinks (where data is consumed). This reveals potential vulnerabilities in the system, allowing developers to address them directly and patch them up.

1.3 Proposed Solution

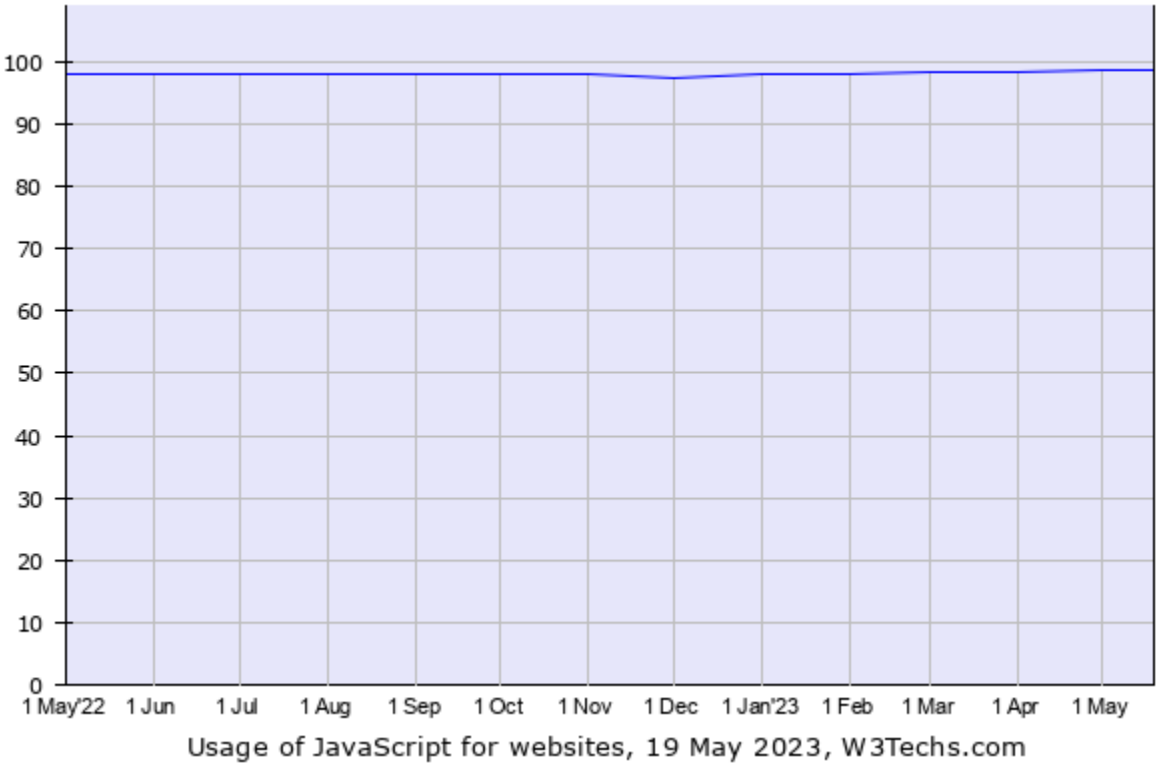
Considering that the aforementioned scenario is analogous to how injection attacks can be prevented, a similar solution may also be applicable to them. Specifically, taint analysis can be used to analyze the flow of input data (referred to as 'tainted data' indicating that it was produced by an untrusted source) in the form of web request payloads and user form inputs. By marking input data as tainted when it enters the application, its movement can be easily monitored and controlled. Before processing this tainted data, it can first be 'sanitized' to remove traces of any malicious code contained within. Thus, the likelihood of injection attacks occurring is lowered

significantly by integrating taint analysis into web applications.

In this paper, we propose a taint analysis technique that employs TypeScript's built-in type-checking features to help prevent common web security threats such as—eval injection, cross-site scripting, and SQL injection. Our reasons for picking TypeScript over any of the other widely used web programming languages for this project are as follows:

- it compiles to plain JavaScript, which is used by over 98.6% of all websites [3]
- using built-in type-checking features means there is no additional overhead caused by tracking taint
- the taint analysis can be done statically (i.e., without requiring the application to be up and running) since TypeScript is statically typed

Figure 2



JavaScript uses wrapper objects to wrap primitive values (e.g., `Boolean`, `Number`, `String`) and provides supporting methods for processing them. By defining new classes to extend these wrapper objects, we are able to create wrapper objects for storing 'tainted' versions of primitive values, which contain an additional property indicating that the values they wrap are tainted and therefore should not be processed directly. Moreover, they can also include a `sanitize()` method which may be invoked to appropriately sanitize the wrapped value according to its data type, thereby removing any potentially harmful data contained within and making it safe (untainted) and ready for processing.

Hence, our taint analysis technique can be used to help prevent injection attacks in web applications by incorporating the following two-step approach:

1. Anytime an input value is about to be processed anywhere within the application, we first perform a check using TypeScript's *type guard* feature to determine whether the value is tainted

2. If found to be tainted, the value is sanitized by calling the appropriate `sanitize()` method

1.4 Results

In addition to the choice of web programming language, modern web applications also have to make a choice for a suitable web framework. For this project, we use the React web framework since it is widely used. This helps us demonstrate the validity of our technique in a typical modern web application setting.

We have put our proposed taint analysis technique to the test by applying the aforementioned two-step approach to our candidate web application (developed using TypeScript and React) and subjecting it to various injection attacks like eval injection, cross-site scripting and SQL injection. Our results show that our taint analysis

technique is indeed effective in preventing injection attacks. Overall, the web app's application data and user data are secured from falling into the wrong hands.

Our goal for this project was to integrate taint analysis techniques into web applications to help mitigate prevalent cyberattacks. Given the distributed nature of the internet, and the amount of data being consumed and processed by web applications, information flow analysis techniques are perfectly suited for web apps to monitor and control the flow of data entering and exiting them.

Although this project's focus is to demonstrate the effectiveness of the taint analysis technique in isolation, it can certainly be combined with thorough input validation and other secure web development practices for even better protection against different kinds of cyberattacks. This could be explored by future work on this subject.

1.5 Report Overview

The rest of this paper is organized as follows:

(a) Chapter 2 discusses related work that was surveyed as part of this project and provides some background on technologies and concepts described in the paper, namely—TypeScript, React and information flow analysis

(b) Chapter 3 describes in detail the design for our proposed taint analysis technique and how we've implemented it using TypeScript, and later integrated into a web application using React web framework

(c) Chapter 4 examines how the technique can be used to prevent some prevalent cyberattacks. It also provides example applications of the technique in different web application contexts

(d) Chapter 5 provides results from performance tests conducted to measure the

overhead caused by integrating our taint tracking technique into a web application

(e) Chapter 6 summarizes the key findings from this project and also suggests avenues for future research and further exploration on the subject

CHAPTER 2

Background and Related Work

2.1 Background

This section aims to provide some background on web technologies discussed in this paper that were essential for this project. It also provides background on information flow analysis.

2.1.1 TypeScript

TypeScript is an open-source client-side programming language that is built on top of the popular web programming language JavaScript, and adds static type-checking as an optional feature. It is developed and maintained by Microsoft.

Although JavaScript is widely regarded as one of the most accessible programming languages, this accessibility comes at the cost of robustness, as the flexibility of JavaScript encourages poor, unsafe coding practices that often give rise to bugs and other vulnerabilities in web applications written in JavaScript.

To address this gap, TypeScript was developed with the goal of making JavaScript more robust and immune to poor coding practices through the inclusion of static typing. The fact that it still compiled to plain JavaScript meant that the powerful portability of JavaScript was also preserved. For example, Figure 3a shows a function written in TypeScript, as indicated by the type declarations in the function signature. This code is compiled to JavaScript, resulting in the code shown in Figure 3b.

Therefore, it is not surprising that TypeScript is now among the most popular web development languages and is considered to be well-suited for developing even large-scale web applications, owing to its robustness.

Figure 3: TypeScript compiled to plain JavaScript

```typescript
function add_concat(num1: number, num2: number, str: string): string {
  let num: number = num1+num2;
  return str + num;
}
```

(a) TypeScript code for a function

```javascript
function add_concat(num1, num2, str) {
  let num = num1+num2;
  return str + num;
}
```

(b) Resulting compiled JavaScript code

2.1.2 React

React is a popular front-end web framework that supports web applications written in JavaScript. It was developed by Meta with a focus on code reusability and efficient DOM (Document Object Model) updates to ensure smooth user experiences.

As web applications continue to grow more complex, their reusing of code also grows in parallel. For instance, a social media web app may choose to display User Avatars to represent their users wherever applicable—on the homepage, individual user profiles, comment sections under posts, etc. This requires that the code used to render User Avatars be reused in all of those places, further adding to the complexity of the codebase and making it harder to debug and maintain. React aims to address this inefficiency in code reuse by providing a natural solution for reusing code and functionality throughout the web app in the form of Components.

Components are discrete units of web app code that are fully self-contained both

in terms of business logic and user interface, i.e., they are responsible for defining their own functionality, and decribing and rendering their own user interface. When used appropriately, they can help break down a web app into smaller, individually manageable modules. This compartmentalization also leads to one of React's most useful design patterns, known as Composition. Components can be composed of other components, resulting in a tree-like hierarchical structure of components. They can also communicate with each other (mostly in the form of top-down communication, since bottom-up communication is quite limited) to pass data in the form of 'props' and even other components. It is therefore apparent why components are referred to as the 'building blocks' of React.

In the current landscape of web application development, React stands out as a prime candidate among other web frameworks due to multiple factors:

- The powerful composition pattern supported by React enables the development of large-scale web apps with ease
- Reusable components provide improved readability and maintainability for developers, boosting their efficiency
- An active community that offers plenty of support and third-party component libraries allows for rapid development of web apps

2.1.3 Information Flow Analysis

Information flow analysis is a technique used in software engineering to analyze how information flows within a system. This helps us identify weak points within the system where sensitive information may be leaked to unauthorized entities, thereby allowing us to fix them and ensure secure information flow.

In the context of web applications and the internet, information flow analysis can prove to be a particularly useful tool considering the vast amounts of sensitive

data flowing in and out, between numerous entities, simultaneously. There is also the constant threat of cyberattacks seeking to exploit any vulnerabilities in applications and steal data. By analyzing the flow of data within the application, we can quickly diagnose such vulnerabilities and work towards patching them up. Furthermore, ensuring secure information flow in web apps also helps us to mitigate a host of prevalent web security threats seeking to infiltrate the application.

Information flow analysis techniques can be broadly classified into two main types based on how they operate:

1. **Static analysis:** Static analysis techniques can be used to analyze the application via its source code statically. Consequently, there is no performance overhead associated with using such techniques as they do not require the application to be running. They help identify insecure code and other vulnerabilities in the source code beginning from the early stages of development of the application (e.g., design and implementation stages).

2. **Dynamic analysis:** Dynamic analysis techniques on the other hand, can be used to analyze the application while it is up and running. It focuses on identifying vulnerabilities based on user interactions and other external factors that come into play during runtime.

2.2 Related Work

The study of taint tracking and taint analysis has received a lot of attention since the publishing of *A Lattice Model of Secure Information Flow* by Dorothy E. Denning [4], which was one of the first papers to describe a mechanism that ensures secure information flow in computer systems.

To examine how well information flow analysis techniques apply to web applications, Staicu *et al.*, [5] conduct an empirical study of information flows wherein

they analyze information flows of 56 real-world JavaScript programs to compare and contrast lightweight taint analysis with implicit flow analysis in terms of their cost and usefulness in identifying security issues within programs. Their findings indicate that lightweight taint analysis is sufficient to help identify most security vulnerabilities in web apps.

Having realized the importance of information flow analysis on the web, researchers have explored new tools and techniques to perform taint analysis in JavaScript, including both dynamic and static approaches. Now, let us review these tools and techniques one at a time, starting with those that make use of the dynamic approach.

Sen *et al.*, [6] developed a powerful analysis tool for JavaScript called Jalangi, which supports numerous dynamic analyses such as taint tracking, symbolic execution, type-checking and more. Jalangi makes use of a combination of record-replay, shadow values and shadow execution techniques to support its dynamic analyses.

Building on the Jalangi framework, Karim *et al.*, [7] use it to implement their dynamic taint analysis techniques in Ichnaea. Ichnaea is written in ECMAScript 5 and uses code instrumentation to analyze taint, thereby making it platform-independent, as demonstrated by their application of the tool to detect privacy leaks in Tizen apps for the *Samsung Gear S2* smartwatch. This tool still fails to track implicit flows, but the authors provide possible solutions to address the gap.

AUGUR [8] is a high-performing dynamic taint analysis framework for JavaScript that makes use of instrumentation API and even supports the ECMAScript 7 specification that introduced asynchronous programming. Due to their use of a Virtual Machine for instrumentation, however, it is unable to be integrated directly with web browsers, instead relying on mock browser environments.

TT-XSS [9] aims to use dynamic taint analysis techniques to specifically tackle DOM-XSS vulnerabilities. By analyzing web pages to obtain taint traces, it is able to

automatically generate attack vectors, thereby enabling the developer to track and fix the vulnerabilities.

That covers most of the substantial work done on dynamic taint analysis in JavaScript, which leaves us to explore the static and hybrid approaches to taint analysis.

ACTARUS [10] performs static taint analysis by modeling JavaScript's characteristic features such as prototype-chain lookups and reflective property accesses. However, their results show a high false-positive rate which they aim to alleviate in future work.

Wei *et al.*, [11] present a Blended Taint Analysis approach which combines both static and dynamic taint analysis techniques to build a more robust taint analysis model. Their findings show that the Blended approach is better performing and more accurate in detecting taint when compared to purely static approaches.

Considering how frequently web apps use third party libraries, Staicu *et al.*, [12] proposed a technique for automatically extracting taint specifications from JavaScript libraries to help identify security vulnerabilities in third-party libraries. Their proposed technique is based on using dynamic analysis to infer information flow specifications from the libraries, and proves to be effective at scale.

In addition to research tackling the application of taint analysis techniques to typical web scenarios, there has also been some work done to explore more unique applications of taint analysis.

BRIDGETAINT [13] introduces a method to track taint across hybrid applications that use JavaScript bridge communication. It uses a cross-language taint mapping technique to restore taint information from sensitive data that's been transmitted through the bridge. Results from experiments conducted on 1172 apps from the Android market demonstrate that BRIDGETAINT is effective in detecting cross-language privacy leaks and code injection attacks.

Saoji *et al.*, [14] present a precise taint-tracking strategy for strings, which tracks taint at the level of characters. This enables them to perform automated sanitization of tainted strings, thereby preventing the system from crashing when tainted information is used insecurely.

CHAPTER 3

Design and Implementation

We've implemented our Static Taint Analysis technique in TypeScript (v5.0.3) and built it into a React application, which has been developed using Meta's *Create React App* as boilerplate. The crux of the technique lies in utilizing TypeScript's type guard feature to distinguish between an untainted String and a tainted String.

3.1 Type Guards

Among the numerous features TypeScript offers to ensure type safety, there exists a special kind of type checking through the use of type guards. Type guards allow TypeScript to narrow down a variable's exact type at a given point in code from a group of potential types, by examining all the different possible branches of execution.

One of the simpler forms of type guards is based on use of the `typeof` operator. The `typeof` operator returns the type of a given variable as a string, such as, "string", "number", "boolean", etc. When TypeScript encounters a usage of `typeof` on some variable, it uses the type information to narrow down the possible types of that variable. Consider the code snippet in Figure 4.

Figure 4: The `typeof` Type Guard

```
function printAll(strs: string | string[] | null) {
  if (typeof strs === "object") {
    for (const s of strs) {
'strs' is possibly 'null'.
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  } else {
    // do nothing
  }
}
```

Here, TypeScript sees the usage of `typeof` operator on variable `strs` within an if condition and uses this information to narrow down the possible types of `strs` to `string[]` and `null`, which are both JavaScript objects. Thus, it alerts the developer that the variable could possibly be `null` in that branch of execution.

To implement our taint-tracking technique, we build our own type guard—`isTainted` by making use of a type predicate, as shown in the code snippet in Figure 5.

Figure 5: The `isTainted` Type Guard

```
type TaintableString = string | TaintedString

function isTainted(str: TaintableString): str is TaintedString {
  return (str as TaintedString).tainted !== undefined;
}
```

Once we define a function whose return type is a type predicate taking the form `parameterName is Type`, we can invoke it with any variable with the corresponding possible types and the type guard will narrow down the specific type of that variable at the point of invocation.

This type predicate forms the basis of our taint tracking technique. It allows us to differentiate between tainted and untainted strings, and can therefore be used near secure sinks to guard against any tainted strings flowing in.

3.2 Tracking Taint in Web Applications

Modern web applications deal with huge amounts of form data coming in from different users, some of whom may have malicious intent. For this reason, we have chosen to build our taint-tracking technique into a React web application. By tainting all incoming user form data, we can control their flow into sinks.

In order to do this, we first need suitable representations for both tainted and untainted Strings. For untainted Strings, we use plain Javascript `String` objects to simplify our problem. For tainted Strings, we have defined a new `TaintedString` class (Fig. 6) which acts as a wrapper around the String class, with added support for tracking taint. This ensures that TaintedString objects can still function the same as regular string objects for all intents and purposes except when tracking taint.

Figure 6: The `TaintedString` class

```
class TaintedString extends String {
  tainted: boolean;
  constructor(value: string) {
    super(value);
    this.tainted = true;
  }
  sanitize(f: (str: string) => string) {
    return f(this.toString());
  }
}
```

Figure 7: The user interface

Our React web application provides a minimal user interface (Fig. 7) with basic form controls to simulate a typical web form. A text field is used to accept user input in the form of a string, which is immediately marked as tainted by wrapping it within a `TaintedString` object. Thus, we now have a source for tainted data to enter the application. For the sink, we have chosen the `eval` function due to its innate security risk of executing JavaScript from a string. To prevent the tainted string from being

directly executed by `eval`, we employ our user-defined type guard `isTainted` in the `evaluator` function as shown in Figure 8.

Figure 8: The `evaluator` function

```
function evaluator() {
  if (isTainted(name)) {
    alert("String is tainted, and must be sanitized before evaluating!");
  } else {
    alert(eval(String(name)));
  }
}
```

In this manner, we utilize TypeScript's built-in type checking features to statically track taint in a web application.

3.3 Limitations

One limitation of our taint analysis implementation is its inability to preserve and track taint across string concatenation operations. For instance, when a tainted string is concatenated with an untainted string, the desired behaviour is for the resulting string to also be tainted. However, in our implementation, we have been unable to replicate this behaviour. This could be addressed in future work, potentially by overriding the `concat()` method provided by JavaScript.

CHAPTER 4

Validation

To validate our taint analysis technique, we can include an insecure usage of tainted data in our web app. Consider the snippet of code in Figure 9. We have a function `secureOperation` which only accepts a plain, untainted string as an argument. We can confirm whether the static taint analysis is functional by attempting to call `secureOperation` with a tainted string, in the function `insecureOperation`. Upon compiling this code, we are greeted by a compilation error in the terminal window, as shown in Figure 10.

Figure 9: Insecure usage of tainted data

```
function secureOperation(str: string) {
  console.log(str)
}

function insecureOperation() {
  secureOperation(new TaintedString('test'))
}
```

Figure 10: Error displayed on compilation

```
ERROR in src/App.tsx:48:21
TS2345: Argument of type 'TaintedString' is not assignable to parameter of type
'string'.
    46 |
    47 |   function insecureOperation() {
  > 48 |     secureOperation(new TaintedString('test'))
       |                     ^^^^^^^^^^^^^^^^^^^^^^^^^^
    49 |   }
    50 |
    51 |   function evaluator() {
```

Thus, we have successfully validated that our implementation of static taint-analysis via type-checking can be used for performing taint analysis on source code at

compilation time.

By integrating this technique into a web application, we also help protect the application against certain types of web security threats, such as:

4.1 Eval injection

When a web application uses the `JavaScript eval()` method to process user input without sufficient validation, it can open itself up to eval injection attacks. Attackers may exploit this vulnerability by including malicious `JavaScript` code within the user input, which when processed by the `eval` method may result in the server crashing (denial of service) or the attacker gaining access to the file system.

For example, consider the following usage of `eval` by a web application to dynamically select the display language for its user interface based on a user-provided location string:

Figure 11: Insecure `eval` usage

```
countryLanguageMapping = {
    'US': 'en',
    'Iceland': 'is',
    'Nepal': 'ne',
    'Spain': 'es',
    'Vietnam': 'vi'
}

function selectLanguage(country) {
    setDisplayLanguage(eval('countryLanguageMapping.' + country));
}
```

An attacker may be able to exploit such usages of `eval` by providing a string input like so:

```
"US;process.exit()"
```

When this string is processed by the `selectLanguage` method shown above, it results in the application process exiting, thereby causing a denial of service for its users.

Our proposed taint-tracking technique can be used to prevent this class of attacks by adding an `isTainted` check within the `selectLanguage` method to examine the input string for taint. If the string is found to be tainted, we can call a `sanitize` method on it to remove any executable code as follows:

Figure 12: Secure `eval` usage with taint tracking

```
function selectLanguage(country: TaintableString) {
    if (isTainted(country)) {
      let sanitizedCountry = country.sanitize(sanitizer); // Removes any executable code from the string
      selectLanguage(sanitizedCountry);
    } else {
        setDisplayLanguage(eval('countryLanguageMapping.' + country));
    }
}
```

Since our technique marks all user input as tainted, the user-provided location string is tainted by default and therefore will be sanitized and cleared of any harmful code before it is processed any further.

4.2 Cross-site scripting (XSS)

In this type of attack, an attacker includes some malicious code as payload to a web request, or as form input on the application. If the application handles this incoming data in an unsafe manner, the malicious code is executed on the client-side (browser), thereby stealing the user's personal data and performing other harmful operations while impersonating the user. Consider the following snippet of code which saves a user's input data to their browser's local storage and later fetches them to display a welcome message:

Figure 13: XSS vulnerability

```
function saveProfile(name, age) {
    localStorage.setItem('name', name);
    localStorage.setItem('age', age);
}

function renderWelcomeMessage() {
    return `<p>Welcome, ${localStorage.getItem('name')}</p>`;
}
```

Here, instead of providing their name in the input field, if the user enters a string containing some `JavaScript` code, such as:

`"<script>alert('This is an XSS vulnerability!')</script>"`

The application then saves this string to the local storage, and the contained script is executed every time the `renderWelcomeMessage` method is called to display the welcome message.

To prevent these types of attacks, we can use taint tracking to track all forms of incoming data to the application and mark them as tainted by default. Before performing any secure operations with the tainted data like saving to storage, we first pass it through a sanitizing function which examines the tainted data for the presence of `<script>` tags or other unsafe data, removing them if present. We can thus transform the `saveProfile` method shown above as follows:

Figure 14: Using taint tracking to guard against XSS vulnerabilities

```
function saveProfile(name: TaintableString, age:TaintableString) {
    if (isTainted(name)) {
        saveProfile(name.sanitize(sanitizer), age)
    } else if (isTainted(age)) {
        saveProfile(name, age.sanitize(sanitizer))
    } else {
        localStorage.setItem('name', name);
        localStorage.setItem('age', age);
    }
}
```

4.3 SQL injection (SQLi)

SQLi attacks are similar to XSS attacks in that they both involve injecting some malicious code into a target web application via web requests or form inputs, however, unlike XSS attacks, SQLi attacks specifically target the application's database. Modern web applications typically connect with a database to store and manage their user data and application data, making it a high priority target for web attacks. A malicious user may attempt to access the application database by including some SQL query expressions within a web request or form inputs. For example, consider the following snippet of code where a `signup` function connects to the database and inserts user data (username) into a table:

Figure 15: SQLi vulnerability

```
function signup(username, db) {
  const SQLQuery = "INSERT INTO Users (Username) VALUES ('"
  + username + "')";

  db.query(SQLQuery, (err, result) => {
    if (err) {
      return false;
    }
    console.log('User signed up successfully.')
    return true
  })
}
```

An attacker can then build a string containing a harmful SQL query to pass as user input to the application (username in this case) like so:

```
"user123'); DROP TABLE Users; --"
```

When this string is processed by the `signup` function, the `DROP` command is executed on the application database, resulting in the `Users` table being removed from the

database.

SQLi attacks rely on web applications insecurely handling data when including them in database queries. As such, these attacks can be prevented by tainting all incoming data and passing them through a sanitizing function (which validates the tainted data by ensuring that they do not contain any unwanted characters, SQL expressions, etc.) before building them into any database queries:

Figure 16: Preventing SQLi attacks using taint tracking

```
function signup(username: TaintableString, db) {
  if (isTainted(username)) {
    return signup(username.sanitize(sanitizer), db)
  } else {
    const SQLQuery = "INSERT INTO Users (Username) VALUES ('"
    + username + "')";

    db.query(SQLQuery, (err, result) => {
      if (err) {
        return false;
      }
      console.log('User account created successfully.')
      return true
    })
  }
}
```

As demonstrated in the examples above, we have successfully validated our taint analysis technique against eval injection, cross-site scripting and SQL injection attacks. Overall, taint tracking proves to be an effective preventative measure against injection attacks.

CHAPTER 5

Performance Results

One drawback that is often associated with taint analysis techniques is the performance overhead caused by tracking taint. We have conducted some performance tests to measure the impact of our own taint analysis technique on the responsiveness of a web application. Page responsiveness is a crucial metric that is frequently used to benchmark websites as it directly translates to user experience—lower response times mean lower wait times for the user, thus ensuring a smooth user experience.

The tests were performed on a 2020 Macbook Air (M1) running macOS Monterey (Version 12.3) and equipped with 8 GB of Memory. For measuring page responsiveness on our web app, we use the Lighthouse tool included in the Google Chrome browser (Version 112.0.5615.137) as part of its Developer Tools (DevTools for short) utilities. Specifically, we use the Timespan mode in Lighthouse to record a user interaction on the web app and measure the Interaction to Next Paint (INP) times during that interaction. INP is a performance metric that measures the time it takes for a web page to become interactive after the user initiates an action, such as clicking a button or scrolling the page. As such, it is a useful metric for measuring page responsiveness.

We designed the test case to span one user interaction of clicking a button, which triggers the app to create some tainted input data and then check it for taint before displaying the data in an alert box. A snippet of our test code is shown in Figure 17. We perform the tests with a varying number of inputs, at 10 runs each. By repeating the tests for untainted input data (i.e., plain strings), we can effectively measure the performance overhead caused by tracking taint in our web app. Our results are summarized in Table 1.

Figure 17: Test code for measuring performance overhead

```
const [name, setName] = useState<TaintableString[]>([""]);

function taintSetName(value: string) {
  let s = new TaintedString(value);
  // let s = value;
  setName([s]);
}

function evaluator() {
  console.log(name);
  let result = name.map((str) => {
    if (isTainted(str)) {
      let x = str.sanitize()
      return x
    }
    return str
  })
  alert(result)
  // if (isTainted(name)) {
  //   alert("String is tainted, and must be sanitized before evaluating!");
  // } else {
  //   alert(eval(String(name)));
  // }
}
```

Test Case	Avg. INP (ms)
20 Tainted Inputs	46
20 Untainted Inputs	39
10 Tainted Inputs	39
10 Untainted Inputs	43
1 Tainted Input	41
1 Untainted Input	48

Table 1: Summary of Lighthouse Test Results

As indicated by the average INP times, it is evident that our web app performed relatively similarly in all of the test cases with the page responsiveness varying from 39-48 ms. A difference of 9 ms can be considered negligible in the case of page response times and can therefore be safely ignored for the purposes of this performance test.

Our performance tests clearly demonstrate that there is virtually no overhead

associated with adding our proposed taint analysis technique to a web app. This is expected, as we make use of standard type-checking features that come baked into TypeScript for our taint tracking implementation.

CHAPTER 6

Conclusion and Future Work

In this paper, we have proposed a novel approach to performing static taint analysis through the use of type-checking. We have implemented this approach in TypeScript, and validated it against prevalent cyberattacks by integrating our TypeScript implementation into a web application built using React. By including support for taint analysis in the web application, we are able to monitor the various flows of input data within the application. This allows us to track the 'tainted' input data and prevent them from being processed or used in any secure operations without performing appropriate sanitization procedures on them first. Our findings show that the proposed taint analysis technique is an effective preventative strategy against injection attacks, which are among the top web security risks today. Furthermore, as it is implemented in TypeScript, a widely used web programming language, it can be adopted by application developers with minimal development effort and at no performance overhead cost, as measured by our performance tests.

For this project, we have primarily focused on the string data type as it is the most common type of input data seen on the web. Future work can explore expanding our technique to work with other data types, such as number, boolean, etc. We can also explore if and how taint analysis can be used to mitigate web security risks other than injection attacks. This will allow us to develop a more holistic approach for mitigating different kinds of web security risks and improving the overall robustness of web applications. Other avenues for future work include:

- automating the taint analysis integration process such that it can be incorporated into web applications with no additional development effort, perhaps as a third-party library
- adding support for tracking taint across string concatenation operations, i.e.,

tainting the resulting string of a concatenation operation between a tainted string and an untainted string

- adding support for dynamically detecting form inputs present on the web application and tainting them without requiring any developer intervention
- improving the sanitization workflow for tainted data and including it within the taint checking process—if some data is being checked for taint, it may as well be sanitized in the same step if found to be tainted

With the increasing adoption of progressive web applications, more users are using web apps now than ever before. Therefore, it is all the more important to protect against vulnerabilities that plague the web and its users. Taint analysis techniques are typically used to track information flowing into a system from untrusted sources. For a web application, untrusted sources may include all of its users, and any other external entity on the internet. Accordingly, given the fact that web apps are constantly exposed to data from untrusted sources, tainting all forms of incoming data is a natural way to track and control their flow within the application. Furthermore, sanitizing tainted data before performing any secure operations with them is a simple but effective approach to prevent pervasive web security threats.

LIST OF REFERENCES

[1] S. Kemp, ‘‘Digital 2023 April Global Statshot Report,’’ April 2023, Accessed on May 6, 2023. [Online]. Available: https://datareportal.com/reports/digital-2023-april-global-statshot

[2] OWASP, ‘‘OWASP Top 10 - 2021,’’ September 2021, Accessed on May 6, 2023. [Online]. Available: https://owasp.org/Top10/

[3] W3Techs, ‘‘Usage statistics of JavaScript as client-side programming language on websites,’’ 2023, Accessed on May 7, 2023. [Online]. Available: https://w3techs.com/technologies/details/cp-javascript

[4] D. E. Denning, ‘‘A lattice model of secure information flow,’’ *Commun. ACM*, vol. 19, no. 5, p. 236–243, may 1976. [Online]. Available: https://doi.org/10.1145/360051.360056

[5] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, ‘‘An empirical study of information flows in real-world javascript,’’ in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, ser. PLAS’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 45–59. [Online]. Available: https://doi.org/10.1145/3338504.3357339

[6] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, ‘‘Jalangi: A selective record-replay and dynamic analysis framework for javascript,’’ in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498. [Online]. Available: https://doi-org.libaccess.sjlibrary.org/10.1145/2491411.2491447

[7] R. Karim, F. Tip, A. Sochůrková, and K. Sen, ‘‘Platform-independent dynamic taint analysis for javascript,’’ *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1364--1379, 2020.

[8] M. W. Aldrich, A. Turcotte, M. Blanco, and F. Tip, ‘‘Augur: Dynamic taint analysis for asynchronous javascript,’’ in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1--4.

[9] R. Wang, G. Xu, X. Zeng, X. Li, and Z. Feng, ‘‘Tt-xss: A novel taint tracking based dynamic detection framework for dom cross-site scripting,’’ *Journal of Parallel and Distributed Computing*, vol. 118, pp. 100--106, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517302186

[10] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, ‘‘Saving the world wide web from vulnerable javascript,’’ in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 177--187.

[11] S. Wei and B. G. Ryder, ‘‘Practical blended taint analysis for javascript,’’ in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 336--346.

[12] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, ‘‘Extracting taint specifications for javascript libraries,’’ in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 198–209. [Online]. Available: https://doi.org/10.1145/3377811.3380390

[13] J. Bai, W. Wang, Y. Qin, S. Zhang, J. Wang, and Y. Pan, ‘‘Bridgetaint: A bi-directional dynamic taint tracking method for javascript bridges in android hybrid applications,’’ *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 677--692, 2019.

[14] T. Saoji, T. H. Austin, and C. Flanagan, ‘‘Using precise taint tracking for auto-sanitization,’’ in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 15–24. [Online]. Available: https://doi-org.libaccess.sjlibrary.org/10.1145/3139337.3139341

APPENDIX

Additional Figures and Source Code Listing

Figure A.18: Global internet use [1]