UCLouvain

epl

École polytechnique de Louvain

Data leaks and suspicious behavior detection in smart home using Taint Analysis

Authors: **Valentin Baillieux, Antoine Dumoulin**
Supervisor: **Etienne Riviere**
Readers: **Charles-Henry Bertrand Van Ouytsel, Donatien Schmitz**
Academic year 2022–2023
Master [120] in Computer Science

Acknowledgment

We are very grateful to Etienne Rivière for his constant feedback, guidance and availability throughout this whole master's thesis. We also want to express our gratitude to Igor Zavalyshyn for his knowledge and help during the first steps of this work. Thanks to our readers Charles-Henry Bertrand Van Ouytsel and Donatien Schmitz. Finally, we acknowledge and thank every person who helped in any way.

Valentin Baillieux and Antoine Dumoulin

Abstract

The proliferation of modern technology, including smartphones, computers, and Internet of Things (IoT) devices, has revolutionized convenience and connectivity. However, this advancement comes with a trade-off between user comfort and the risk of data leakage and privacy breaches. This master's thesis focuses on privacy issues within smart homes. The objective is to provide a proof of concept of whether dynamic taint analysis on its own is a viable solution to identify and prevent sensitive data leaks within a smart home environment. To achieve this, we created TaintWasm a dynamic taint analysis for WebAssembly binaries. TaintWasm is based on Wasabi a dynamic analysis framework for WebAssembly. The research unveils that efficient dynamic taint analysis is challenging to implement in many aspects. The Most notable ones are real-time analysis, portability and implicit flow handling. Though some results are encouraging and dynamic taint analysis arises as a strong asset for such systems. It appears that to be a viable solution it should be combined with other techniques to overcome real-world limitations.

Contents

1 Introduction **5**
1.1 Contribution 8
1.2 Organisation 9

2 Context **10**
2.1 Internet of things, devices and hubs 10
2.2 Privacy concerns in smart homes systems 11
2.2.1 Privacy concerns and perception of vulnerable users 12
2.2.2 Existing privacy solution in smart homes 12
2.3 TaintWasm context 12

3 Dynamic and static analysis **14**
3.1 Static analysis 14
3.1.1 Usage of static analysis 15
3.1.2 Benefits and limitations 16
3.2 Dynamic analysis 16
3.2.1 Usage of dynamic Analysis 17
3.2.2 Benefits and limitations 18
3.3 Dynamic taint analysis 18
3.4 Technical details of dynamic taint analysis 19
3.4.1 Taint introduction 19
3.4.2 Taint sink, point of detection 20
3.4.3 Taint flow propagation and taint checking 20
3.4.4 Existing dynamic taint analysis systems 22
3.4.5 Limitations 24
3.5 Duality and synergies of static and dynamic analysis 25

4 WebAssembly **27**
4.1 WebAssembly Overview 27
4.2 Why using WebAssembly 27
4.2.1 Advantages of using WebAssembly in IoT systems 28

4.3 Impact of WebAssembly in our theoretical architecture 28
4.4 Wasabi 28
4.4.1 Our contribution to Wasabi 29

5 TaintWasm design 30
5.1 TaintWasm Objectives 30
5.1.1 Implicit flow handling 31
5.2 Environment 32
5.2.1 Trust model 34
5.2.2 WebAssembly execution environment 34
5.2.3 The APIs 35
5.3 TaintWasm dynamic taint analysis structure 36
5.3.1 Taint propagation policies 37
5.4 Challenges 38

6 TaintWasm implementation 39
6.1 Technical prerequisite 39
6.1.1 WebAssembly Execution 39
6.1.2 WebAssembly Text Format (WAT) 40
6.2 Wasabi instrumentation and hooks 43
6.2.1 Instrumented Code and Analysis 43
6.3 TaintWasm implementation 47
6.3.1 Mirrored Stack implementation 48
6.4 Flows analysis examples 49
6.4.1 Explicit flow complete analysis example 50
6.4.2 Step-by-step illustrated analysis for explicit flow 52
6.4.3 Implicit Flow and Potential Implicit Flow Handling 61
6.5 Implementation challenges 64
6.5.1 Managing Function Stack Popping 64
6.5.2 Challenges in Compiling C Code with Emscripten 65

7 Results, limitations and possible improvements 67
7.1 Our Benchmark 67
7.2 Discussion 71
7.3 Possible improvements 73
7.4 Conclusion 74

Chapter 1

Introduction

In current times, the widespread use of smartphones, computers, and IoT devices has transformed our relationship with technology, providing exceptional ease and connectivity. Yet, this progress also involves a fundamental compromise between, comfort ease of utilization, and a high vulnerability to data leakage and privacy violations. The last decade saw several smart home systems emerge for purposes ranging from the pure comfort of users to health care systems. Even though this growth in the demand for such systems does not seem to diminish, we see a parallel growth in the demand for privacy as well as a desire to have proof of the honest use of sensitive data in such systems. Our work takes place in the scope of privacy by design, smart home system, and with the will to give visibility on the usage of their sensitive data to the end user. To be more precise, we work with the objective to give control and visibility over sensitive data to vulnerable users. Indeed, a lot of innovations in the field of home healthcare systems have been done thanks to the emergence of IoT [1] [2]. However, most of those systems heavily rely on a huge amount of sensitive data making them prone to data leak and thus slowing their acceptance rate for vulnerable users. The objective of this master thesis is to find a possible solution to detect and prevent data leaks in a smart home context and evaluate its viability. We would like to find a solution that would empower vulnerable users toward their sensitive data. To find where we could work we analysed the context of IoT smart homes in further detail.

There is already a plethora of privacy-enhancing technologies and research occurring for existing smart-home systems. Those are deployed at several different levels, applications, sensors in the cloud etc. The deployment of those technologies depends on the architecture of the smart home system. The most noticeable architectures are cloud-based smart homes with a hub that serves mainly as a proxy with the cloud and most computations are done in the cloud. In opposition to the cloud-based smart home system, we have architectures that are hub-only,

there is no connection to the internet and all computations are done locally. In our case, both of those types of architectures are problematic, indeed, since the cloud is managed by exterior stakeholders it is difficult to have control over how data is handled, thus we consider the cloud to be untrusted. On the other hand, resources in hub-only architectures are limited restraining the possible applications that could be deployed on them. Hybrid architecture can be used to overcome those limitations. We consider the cloud to be untrusted though necessary. Healthcare systems need the cloud to be effective. As a simple example, when we consider a fall detection system. If a fall is detected, the system must be able to quickly send a notification to the person in charge of the monitored user. This is not possible or at least very difficult without the use of the Internet. But most computations take place locally. Doing so makes the smart home system less reliant on the cloud while not too much limited in resources. Nevertheless, if the smart home system accept third-party application and rely on a proprietary cloud the end user still has not much control nor visibility over its sensitive data. We decide to work on this architecture model since it was the most promising for healthcare applications while keeping room to enhance privacy.

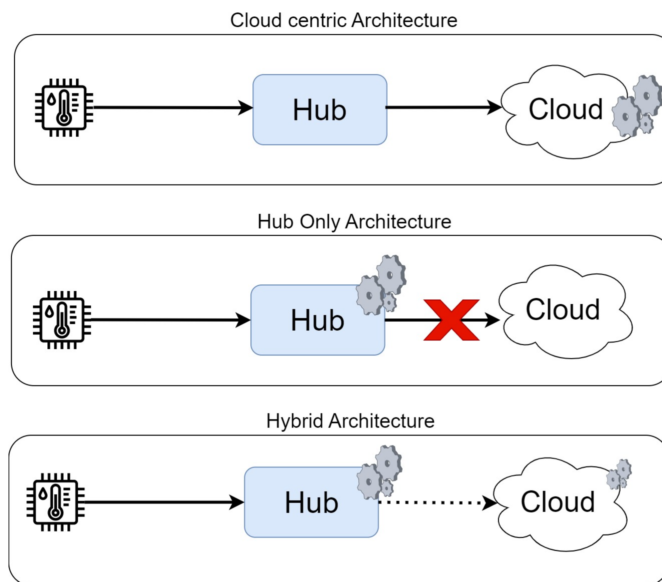

Figure 1.1: Comparison of smart home architecture

We encounter a problem at this point. Even though hybrid architecture seems to be promising, there are not a lot of such systems on which to experiment. Or at least there were no such systems suitable for our case. At this point, we found in the literature that more and more papers propose to use WebAssembly at the

edge [3] [4] [5]. WebAssembly is a compilation target hardware and platform independent. As some papers suggest it can be used beyond the sole context of the web even though it is its primary use. Indeed, the inherent portability and efficiency of WebAssembly make it a great technology for IoT systems. Since WebAssembly seems to be a promising technology in this context and more research propose

Our goal is to find a solution to ensure sensitive data is used correctly by the smart home system and sent to the cloud strictly when necessary for the proper work of the application. Moreover, we want to give a visualization of their sensitive data usage to the end user. In other words, we want to find a solution to identify and report sensitive data leaks in hybrid IoT smart homes. After some research, we found that dynamic taint analysis (DTA) is a promising avenue for further exploration. Comparable systems have already been implemented in similar contexts such as smartphone environments [6]. Though we found various applications of DTA, it seems to our knowledge that it has not been done in the context of hybrid cloud architecture of IoT systems. Existing systems of DTA are implemented with a lot of specificities at the glance of a wide range of different contexts. Indeed, it largely depends on various factors imposed by the application that is monitored and at which level the analysis is implemented. For example, if it is an interpreted language or a compiled language would greatly impact the base of the DTA implementation. If they have access to the source code or not too. We will explore those flavours of DTA in a dedicated chapter. As we said in the previous paragraph, there are no suitable IoT systems on which to implement a new DTA system within a reasonable amount of work. Dynamic taint analysis systems require a lot of fitting depending on the contexts in which it is implemented, and it can be a massive work. A usual way to implement a new dynamic taint analysis is to extend an already existing framework. Indeed, many dynamic analysis systems are relying on previously implemented systems. Thus we searched for an existing framework to extend. Since our goal is to test whether or not this is a good solution for the IoT context this approach seemed to be the most suitable.

Since WebAssembly is promising for IoT systems we decided to perform taint analysis on WebAssembly binaries. To implement our DTA we decided to extend an already existing framework because it was the most viable option in our case. We use Wasabi, which is a prototype dynamic analysis framework for WebAssembly. Wasabi works by instrumenting the compiled binaries producing a version with low-level hooks that can then be used to perform an analysis at a higher level using JavaScript. It is on top of that instrumented binaries that we implemented our dynamic taint analysis systems. Although Wasabi gave us a base on which to work it is still a prototype, thus the installation and the understanding of the framework

necessary to extend it represent a certain amount of work. In addition, it has some limitations in comparison with other existing dynamic analysis frameworks and we had to perform some bug corrections. Due to this our taint analysis system might seem not as complete as some other existing systems in other contexts, however, it has sufficient capacity to make our proof of concept.

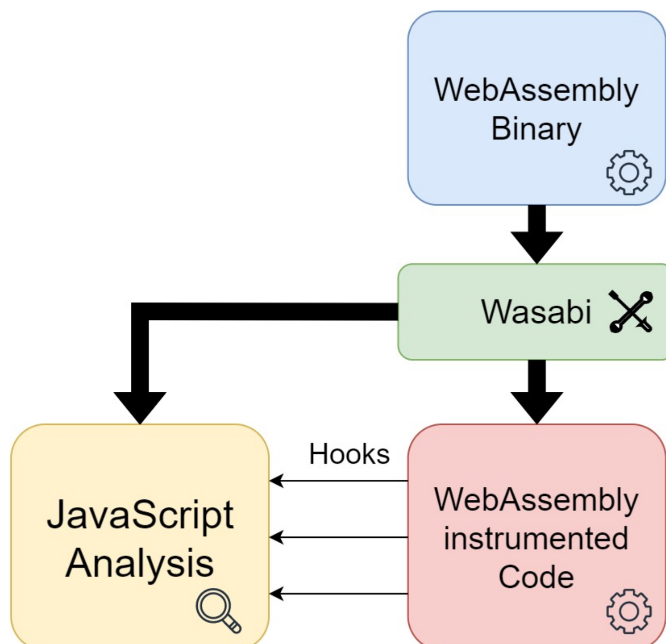

Figure 1.2: Wasabi dynamic analysis overview

Our implementation of dynamic taint analysis over WebAssembly binaries showed that although it is possible to perform tracking over sensitive data to prevent them from exiting the authorized area of a system, it is necessary to put in place other systems or extensions to the tool to prevent every possible data leaks. Moreover, a pure DTA system implemented on top of instrumented application binaries induce a large overhead in comparison with the raw application making it difficult to implement in real-time. Even if some limitations have arisen from our tests, we found some valuable tracks to explore that would mitigate those limitations in already existing DTA systems.

1.1 Contribution

Our contribution is a new dynamic taint analysis tool for WebAssembly. We developed and tested this tool for a theoretical hybrid smart home architecture to

perform a proof of concept. The architecture is theoretical since we did not find a suitable system on which to experiment, thus we create a theoretical context to compensate. Our objective is to test if dynamic taint analysis is a promising approach to detecting and preventing sensitive data leaks in an IoT context. We tested as well if it could be used as a tool for logging data flows and gives developers a tool to visualize their use of sensitive data or directly to the end user. TaintWasm has been developed on top of Wasabi [7], a dynamic analysis framework for WebAssembly binaries. We also contributed to Wasabi by correcting some bugs in the source code and the simple taint tracking example.

1.2 Organisation

Our objective during this master thesis was to find and explore a solution to enhance privacy within a smart home context. We found that testing dynamic taint analysis on WebAssembly binaries seemed to be a good contribution in this context. We had to perform some research to make our way up there. In the document, we will present the research we made to come up with our goal, then we cover in detail our implementation of TaintWasm and the results of our tests around it.

First, we will cover the context in which our research begins. Then, we will cover the broad topic of dynamic analysis with a small overview of static analysis as well, to understand why dynamic taint analysis is promising in a smart home context. After that, we will explain why WebAssembly is a promising technology for IoT. At this point, we will dive into the implementation of TaintWasm itself by covering our design choice before diving into lower-level details. We will then explain our testing around the tools and present our results. Finally, we will discuss those results and give improvement ideas for this concept.

Chapter 2

Context

In this chapter we will cover the context in which our research takes place. We will cover the wide environment that represents smart homes and explain our motivation to work in this context. First, we will quickly cover the present state of smart homes and existing architectures. Then we will cover the privacy concerns of smart home users. Then we will speak more specifically about the concerns of vulnerable users. And finally, we will make an overview of existing solutions to those concerns

2.1 Internet of things, devices and hubs

The number of IoT devices is continually growing and has reached billions [8]. A large number of these devices are used in smart homes or home automation scenarios [9]. These smart home applications rely heavily on the internet, as some tasks require considerable resources to be performed and more simply because it is convenient and efficient. The connection between devices and the internet is usually handled by hubs [9]. These smart home systems are often offered by commercial stakeholders. Examples of such smart hubs include Amazon Alexa [10], Samsung SmartThings [11], or Somfy [12]. On the other hand, some hub systems are open-source and can be set up by the users themselves using devices like Raspberry Pi. Two examples of hubs that can be installed on a Raspberry Pi are OpenHab [13] for home automation and Mycroft [14] for voice recognition. It is interesting to note that all of these hubs have a wide range of architectural and design differences making them difficult to compare directly. Some are based on local data storage and processing, such as OpenHab, while others serve as a proxy with a cloud back-end. In the SOK paper [9] they refer to those systems as cloud-centric, hub-only and hybrid ones as hub-centric, we will from now use those terms too. There are also a plethora of other differences in other aspects whether

it is the handling of third-party applications, the securities that are put in place, or the way data is collected and treated. Indeed, IoT solutions have a wide range of different architectures and platform designs [15]. Regardless, these hubs access and collect a large amount of data that could be sensitive, raising safety concerns.

IoT systems that support third-party applications are very similar to smartphone environment. Indeed, smartphone operating systems supports a wide range of sensors and devices such as camera, it also heavily relies on the use of the cloud and supports third-party applications. In fact, most of the applications that are installed on smartphones are third-party ones. Thus both environments have access to and collect a large number of data ranging from simple temperatures to video feeds, making them sensitive to data leaks etc.

2.2 Privacy concerns in smart homes systems

Even though we know that IoT systems represent a threat to the privacy of users, those solutions are adopted anyway. We can see this in the continually increasing number of IoT devices. This is normal knowing that users adapt their behavior toward their privacy depending on the comfort and the convenience systems brought to them. Furthermore, since a substantial amount of IoT systems heavily rely on clouds that are managed by the provider of the systems, users must trust those providers to handle their private data properly [16]. This as we would expect make users suspicious of those providers [17]. To reassure them, those providers assure the users that a lot of security systems have been put in place to protect their privacy. Now, since the users get enough convenience from the systems most of them will not bother to verify that the statements concerning their privacy are indeed respected by the providers. Moreover, a vast majority of those users are unaware or at least partially unaware of the security threat that inference on non-audio nor visual data represents [18].

We know that IoT environments raise legitimate privacy concerns. Even if some users are willing to adopt those solutions accepting the privacy issues that might result for the sake of convenience, this does not make it less important to address those issues. Those concerns are still a barrier to the acceptance of smart home solutions which is still a problem beyond pure comfort.

Indeed, a lot of studies have been conducted on data leaks caused by IoT systems as well as the perception IoT users have on their devices. They show that most of the users have real concerns about the privacy and security of IoT even if they are

already using such systems. A substantial part of those users perceive their devices as mysterious and are willing to see security improvement [19].

2.2.1 Privacy concerns and perception of vulnerable users

On the side of vulnerable users such as the elderly privacy concerns are also very present. This might be a problem since the elderly population is growing and more of them would like to avoid going to care homes [20]. To avoid it or at least differ it there is a huge amount of possible smart homes application [1] [2]. In those solutions, we can think about Human Activity Recognition (HAR) to detect falls, or sensor monitoring to spot if the oven has been working for an abnormal amount of time etc. Still, those solutions require a huge amount of data collection that might prevent users to accept them if they do not trust the environment deploying those solutions. A lot of older adults are willing to accept those technologies to live in their house for the longest time possible but they still have a lot of concerns [21].

2.2.2 Existing privacy solution in smart homes

In this context we see that a lot of new technologies and techniques aiming to enhance privacy are being developed in the industry, as well as in the scientific communities. Since IoT has so many different architectures, designs and objectives, technologies aiming to enhance privacy are deployed on a very large spectrum. Those range from Network to device securities, passing by applications and many others.

The similarities between IoT environment and smartphone environment can be useful here since a lot of innovation has already been made on the side of smartphones, which can be inspiring for the IoT context. For instance, TaintDroïd proposes a dynamic taint analysis system for the Androïd operating system. It aims to give visibility to the user [6].

Another example of solution in this domain aims to tackle privacy and security concerns associated with third-party-driven IoT solutions and extensive data collection. This paper's primary objective is to establish a robust method for verifying data transmission to third parties while preventing potential misuse [22].

2.3 TaintWasm context

Our objective is to provide a new solution to enhance privacy in hybrid smart home architecture. We decide to choose hybrid smart home architecture since it

provides the best compromise between the possibility to deploy privacy-enhancing solutions and the use of the cloud for applications. We found that dynamic taint analysis seems to be a promising technology to identify and prevent sensitive data leaks in smart homes. This proves to work well in similar environments such as smartphones with Taintdroid [6]. We thus decided to test dynamic taint analysis in the context of hybrid smart home architecture. However, we did not find a suitable system on which to develop a dynamic taint analysis tool. As we said in the previous chapter we decided to work with WebAssembly because of its inherent security and portability making it a good choice for the IoT environment.

To compensate for the fact we have no system on which to experiment, we decided to design a theoretical architecture emulating a hybrid smart home system. Thanks to WebAssembly portability even though our dynamic taint analysis system will not be deployed on a concrete system the results would still be useful. Our theoretical architecture will be a hybrid smart home with a hub that runs third-party applications provided through binaries similar to a smartphone environment. Those applications will be forced to handle sensitive data inside WebAssembly binaries embedded in their source code. We force this by providing this data only through an API. This API will push data only inside a WebAssembly binary. This is on the WebAssembly binaries that TaintWasm our dynamic taint analysis tool will run. This is in this emulated environment that we will test if dynamic taint analysis meets the expected objectives.

Chapter 3

Dynamic and static analysis

This chapter will explore the broad topic of dynamic and static analysis in various contexts, with a particular focus on dynamic taint analysis. To fully comprehend the value of TaintWasm, it is crucial to have a clear understanding of these concepts. As previously mentioned, our work is a dynamic taint analysis tool for WebAssembly binaries, so we will delve deeper into this type of analysis than the other flavours of analysis techniques. We will begin by providing an outline of static and dynamic analysis techniques, including their specific challenges and limitations along with some existing usage of both techniques. Then, we will examine dynamic taint analysis (DTA) in greater detail, looking at existing use case scenarios and solutions. Following this, we will discuss the technical aspects of dynamic taint analysis. Finally, we will see that both static and dynamic analysis have limitations that can be mitigated by combining both techniques.

3.1 Static analysis

Static analysis is a technique used to inspect code without running it. It is used in software development as well as in security. This analysis entails examining the program's source code, bytecode, or binary code to detect potential problems, security vulnerabilities, and issues with code quality. There exist specialized tools to scan the codebase for possible bugs, patterns, violations of coding standards, and other potential issues. PolySpace [23], Coverity Prevent [24] and Klocwork [25] are examples of such static analysis tools. Static analysis is frequently employed early in the development process and prior to code deployment to identify errors and improve overall code quality [26].

We will illustrate static analysis with a concrete example. We consider a C function 3.1 that scans a user input and then compares it before printing a result.

Static analysis could automatically detect that x is uninitialised which could lead to exceptions during runtime. For instance, if we use Splint [27] which is a static analysis tool for C code. It will consider control-flow paths, analyze loops and control statements using heuristics to recognize common idioms, and other strategies [28]. Using those techniques Splint will raise a warning so that the developer could correct his code before running it.

Listing 3.1: simple C code

```
1  #include <stdio.h>
2
3  int main() {
4      int x;
5      printf("Enter a name: ");
6      scanf("%d", &x);
7
8      if (x > 10) {
9          printf("x is greater than 10\n");
10     }
11
12     return 0;
13 }
```

3.1.1 Usage of static analysis

Static analysis is used in a wide range of applications, therefore it is difficult to give a complete view of usage scenarios. However, we will cover some noteworthy examples in this section. First, Emanuelsson et al [26]. give in their study a non-exhaustive list of runtime errors that are not usually detected by testing nor compilers but that can be found thanks to static analysis. They cited improper resource management such as leaks or non-freed memory in dynamic allocation, dead code and data (code that will never be executed or unused data) and incomplete code such as missing return statements or ill-written switch cases. We will illustrate with a concrete example.

Secondly, here is more use cases for which static analysis has proven to be valuable. Detecting defects and vulnerabilities in code is a field where static analysis is widely used. Conducting thorough testing before deploying a program can result in significant resource savings. Static analysis methods are typically used for debugging purposes. It can efficiently identify and locate bugs in software, thereby reducing the time and resources required to improve code as well. Industrial static analysis tools generally propose a debugging functionality. Another example would be when working with untyped or weakly typed languages, static analysis is often employed to make type inferences and type checking, Jens Palsberg and

Michael I. Schwartzbach's work "*Object-Oriented Type Inference*" is an example of such a use [29]. Optimization of source code can also be performed with static analysis [30]. To conclude, we can also note that static analysis is also used for impact analysis, identifying the impact of code change in the software, and software metrics such as complexity, reliability, or maintainability [26].

3.1.2 Benefits and limitations

After covering the topic of static analysis, we will now explore the various advantages and limitations of this technique.

On the side of the benefits, as we noted in the prior section the early detection of potential issues such as certain bugs, vulnerabilities, security flaws, errors, etc. permits their correction before runtime avoiding misuse of resources and time. Static analysis also allows uncovering problems that are not found by the compiler or simple testing. Furthermore, some analyses are easily set up during the development phase of softwares. Overall, all those aspects give a better code quality, better cost-effectiveness, and a reduction in risks [26].

However, there are still some limitations too. Indeed, since static analysis occurs before the execution of the source code, it cannot detect some runtime-specific issues or interactions that happen during the execution. For instance, the scope of static analysis is certainly limited, directly linked to the inaccessibility of runtime-specific resources, it can not be performed on user input which is a big source of potential security threats. The algorithms and techniques used for static analysis also incur false negatives as well as false positives inducing manual verification and tuning tools which is time-consuming. Some complex issues like data flow analysis or race conditions might be very challenging to detect solely relying on static analysis. Although some static analyses may be simple to implement, other tools can be quite complex to set up [26]. Finally, static analysis is most of the time unusable when source code uses some obfuscation techniques [31]. Obfuscation is a technique used to change source code without altering the code semantics, it is generally used to protect intellectual property by making reverse engineering very complex [32].

3.2 Dynamic analysis

Dynamic analysis is a technique that involves the evaluation of a program from a current execution. It helps to find runtime-specific issues such as memory leaks, performance issues, or other unexpected behaviours linked to the actual execution of the analyzed program. So Dynamic analysis (DA) operates by observing the

behaviour of the program during its execution. To do so DA capture runtime data, monitor interactions, and perform analysis on the interaction of the program with the environment and data flow. For instance, the Valgrind memcheck [33] tool is a dynamic analysis that will capture memory allocations and deallocations, memory reads and writes, memory leaks, invalid memory accesses etc.

The main difference between static and dynamic analysis is straightforward. Indeed, static analysis techniques are used directly on the source code, the byte code, or even the binaries of a program, while dynamic analysis is performed on the execution itself. However, those techniques are complementary we will see that in our case some limitations of our dynamic taint analysis tool can be mitigated by static analysis techniques.

3.2.1 Usage of dynamic Analysis

In this subsection, we will cover notable domains where dynamic analysis in general has been used and proven to be efficient.

Dynamic analysis offers a wide range of use cases in various domains such as Web applications, multithreaded systems, distributed systems, Objected Oriented systems, or security [34] [35]. Furthermore, dynamic analysis is widely used in the field of program comprehension [34]. However, the concept of *program comprehension* is quite vague, Cornelissen et al [34]. used the definition from Biggerstaff et al [36]. to clarify this concept "*A person understands a program when he or she can explain the program, its structure, its behaviour, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program*". In this scope, they pointed out that a lot of studies have been conducted on *feature location* (the identification of source code linked to a feature of the program) with the help of dynamic analysis techniques which prove to be particularly efficient for such tasks. This can be illustrated by D. Liu et al [37]. semi-automated technique for feature location in source code. Though as pointed out by Cornelissen et al [34]. dynamic analysis has been less used for program comprehension on the side of the end user. TaintWasm has not specifically been created for program comprehension or feature location, however, it aims to give a visualization of how sensitive data is handled by a program to the end user.

Dynamic taint analysis is also used plenty in the field of security. Some notable domains of security where dynamic analysis is used are malware analysis, detection of cross-site scripting and SQL injections in web applications, understanding of

network protocols, and detection of sensitive data leakage [35]. We will give more details about those subjects in the following section covering dynamic taint analysis.

3.2.2 Benefits and limitations

As we begin to see or understand dynamic analysis and static analysis have a strong duality and possible synergies, we will talk about these aspects in a future section, however, we will thus see similarities with the previous section covering benefits and limitations of static analysis.

Dynamic analysis offers a lot of benefits. To begin with, since dynamic analysis is performed during a proper execution of the analyzed software, issues that are runtime-specific are available to the analysis. This feature helps identify problems that may be related to a particular setting, like how the software interacts with other components or responds to user input. Dynamic analysis can also be used in complex software environments such as web applications, distributed systems, or multithreaded systems.

However as with everything dynamic analysis is not spared from downsides. The first flaw is directly linked to the idea that dynamic analysis is performed on a specific execution. Due to this, it lacks a general view of the software to produce its results. Specifically, the results of dynamic analysis are only applicable to the particular execution in which it was performed. This makes it so that the analysis cannot easily find every possible path of execution or every specific problem directly. Then, since the execution of the code is required, running the examination can be very costly for intricate or considerable software. Undoubtedly, dynamic analysis often induces a large overhead on the analyzed program.

Expanding upon this, it becomes evident that dynamic analysis engenders a notable performance overhead, as exemplified by the outcomes of the Wasabi tool [7]. Empirical investigations reveal that the observed overhead associated with Wasabi varies, falling within the range of 49 times to 163 times the base execution time [7].

3.3 Dynamic taint analysis

Dynamic taint analysis (DTA) closely monitors a program while it runs and effectively traces the path of *tainted* data. This kind of analysis is usually done at a fairly fine-grained level and tainted data is most of the time derived from untrusted sources, such as user input [38].

Typically, DTA involves tainting data at a specific source known as a *taint source*. Taints are then propagated according to a certain *taint policy*, after this, the analysis will raise a warning or take some actions such as interrupting the execution if a tainted value arrives at a *taint sink*. The fundamentals of this technique remain consistent despite the variations in implementation that arise from different analysis contexts. Most challenges and limitations are also consistent across these contexts [39].

3.4 Technical details of dynamic taint analysis

It is crucial to understand the technical aspects of DTA to fully comprehend the challenges and implementation of TaintWasm. As previously stated, TaintWasm is a tool for analyzing WebAssembly binaries using DTA techniques. Therefore, this section will delve into the specifics of DTA implementations.

3.4.1 Taint introduction

How to store taints in a DTA system is not a question we can easily answer since it heavily depends on every specific implementation. However, when analysing DTA in a general context it is convenient to think that every data in the scope of the analysis is represented as a tuple composed of data and a taint. The taint value is usually binary, meaning it is either tainted or not. However, we do not address the concept of multi-coloured taint in our work.

A common method is to define every component, such as variables, memory slots, and blocks, at the start of the analysis. The level of detail depends on the scope of the analysis. Then those are tainted when data is coming from a taint source or a taint has been propagated. What are the taint sources in a DTA system also varies a lot in function of its implementation, however, every taint source has the same role which is to give a taint when data is coming from them.

To be more concrete we consider a DTA system that taints sensitive data coming from a camera. A possible taint source would be for example the API call to get the camera feed. Indeed, the variable in which the result is going to be stored is untainted when it has been instantiated. Then the API is called to get the camera feed and store it in the variable, at the same time since the API call is a taint source the variable will be tainted.

In a nutshell, everything is untainted at the beginning of the execution then every data derived from a tainted source is tainted.

3.4.2 Taint sink, point of detection

A taint sink is a statement in the code where the DTA system will check if the value going through the sink is tainted and if so it will take action.

3.4.3 Taint flow propagation and taint checking

Now that values are tainted by sources and detected at taint sinks we need to have a policy to propagate them through the execution. Taint propagation rules or policy can be seen as a kind of propositional logic operation over another operation. For example, take a part of the program that adds value to another, how to check if the result must be tainted? A very simple and usual approach is to say that if one of the two value are tainted then the result must be tainted as well. This is a simple taint propagation rule. So for an operation with the tuples of values and taints, (v_1, t_1) and (v_2, t_2), if we have the operation $v_1 + v_2 = r$ then if t_1 or t_2 are tainted r is tainted as well in accordance with the previous taint propagation policy. We can formalize the generalization of $\oplus$ any binary operator, thus having this formula.

$$(v_1, t_1) \oplus (v_2, t_2) = (v_1 \oplus v_2, t_1 | t_2)$$

In practice, DTA systems implement a set of propagation rules depending on their environment and their objective. Tuning taint propagation policy correctly is very important to avoid under-tainting and over-tainting, respectively cases where a value should have been tainted and a case where a value should not have been tainted. If under-tainting and over-tainting occurs then the DTA will produce false positive or false negative.

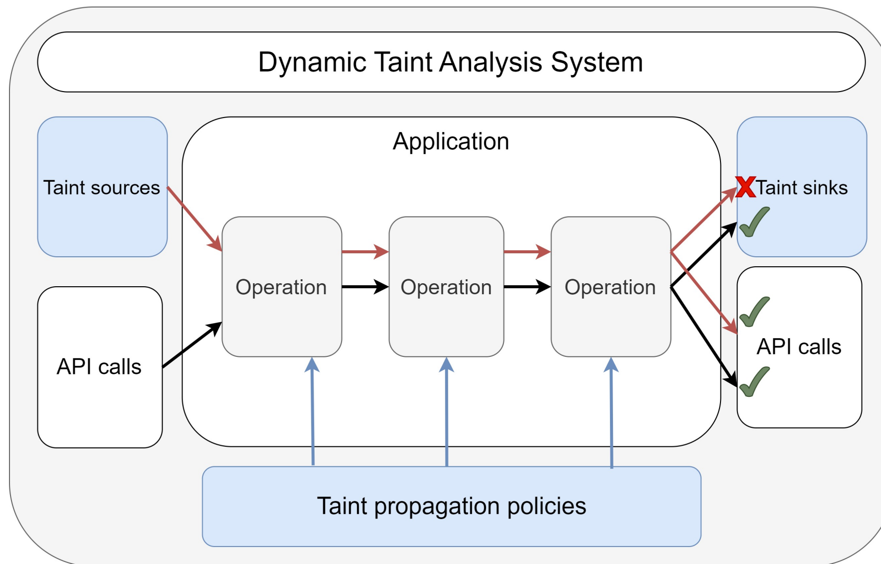

Figure 3.1: Illustration of a fictive DTA system

The figure 3.1 does not aim to represent every dynamic taint analysis system but gives an illustration of the previous explanation. Here some API calls are taint sources and other are taint sinks, but every API calls are not always a sink or a source, since some flows might be untracked. We see that flows coming from the sources are tainted and propagated according to the propagation policies, and then some actions are taken when a tainted flow is detected at the taint sinks. However, untainted flows do not trigger the sinks, and tainted flows are not detected outside of sinks.

Explicit flow

When data is explicitly used in the program such as in the previous example we have an explicit flow. Dynamic taint analysis handles those flows by default with not many challenges with proper taint propagation policies.

Implicit flow

Implicit flows are also called control-dependent flows. Such flow occurs when statements of a program are control dependent on another statement of the program. The listing 3.2 is a very simple example of such a flow, indeed the statement "*push_to_cloud()*" at line *4* is control dependent on the statement "*b = get_input()* at ligne *2*, the second statement will occur depending on the previous statement.

With simple taint propagation policies it is difficult the propagate taints correctly within such flows, this often results in under-tainting or over-tainting. To illustrate that take a look at codes 3.2 and 3.3 below. If we consider the method "*get_input()*" as a taint source and the method "*push_to_cloud()*" as a taint sink. We would like to raise a warning if a taint propagated from the source arrives at the sink. In the first code, we see that the variable "*a*" should not be raised by the taint sink since it is not influenced by a tainted value. While in the second code, we see that we reproduce the value of b and then push it to the cloud, thus we would like to raise a warning. But how to tune taint propagation policies to achieve this? It is quite challenging to get the exact behaviour we would like using exclusively DTA and taint propagation policies. Indeed, if we decide that every control block that is in contact with a tainted value should propagate the taint then in the case of the first code we would raise a warning for "*a*" which is over-tainting. Whilst if we do not put this taint policy, in the case of the second code the sink would not raise a warning which would cause under-tainting. Those examples are not meant to be exhaustive by representing well the kind.

Listing 3.2: Simple control dependent statement example

```
1  a = 0
2  b = get_input()
3  if b == 0 then
4      push_to_cloud(a)
```

Listing 3.3: Undertainting of implicit flows example

```
1  a = 0
2  b = get_input()
3  while a != b then
4      a++
5  push_to_cloud(a)
```

3.4.4 Existing dynamic taint analysis systems

In the previous section about dynamic analysis, we covered the general application of dynamic analysis, in this section we will go through some implementation and existing solutions of DTA. To do so we selected five different research papers each presenting a dynamic taint analysis method. We choose those papers because they each try to mitigate different limitations inherent to DTA.

We begin with *DTA++* [40] an enhanced dynamic taint analysis aiming to cover implicit flows also known as control-flow dependencies. A more traditional approach to DTA does not consider control-dependent flows since doing so implies a huge over-tainting (false positives). DYTAN [41] is an example of a system that deals

Table 3.1: Comparison table of DTA implementations

Paper	Implementation[1]	Realtime	Portability	Implicit flow
DTA++	I	-	no	yes
TaintCheck	O	yes	no	yes
JS DTA	I	-	yes	n/a
DECAFF++	O	yes	yes	n/a
TaintDroid	M	yes	no	no
TaintWasm	I	-	yes	yes

1. "I" means implementation done by code instrumentation, "M" means implementation done by interpreter modification, and "O" stands for other techniques.
2. Since DTA has a vast amount of different features and those features are not binary either, this table aims to tell if the paper focus on those aspects or not, thus "yes" if the aspect was in the focus, "no" if not and "-" if it was not in the primary focus but still implement some functions going in the way of the feature.

with implicit flows but induces a lot of over-tainting. However, ignoring those implicit flows induces under-tainting (false negatives). DTA++ diagnoses tainted implicit flows and performs additional taint propagation only within information-preserving transformations, resolving under-tainting without causing over-tainting. In the case of DTA++, information-preserving transformation is a transformation that implements an injective function, this means that every legal input for the transformation produces distinctive outputs. Without getting into too much details it means that tainted data have a direct impact on the result of the transformation and thus the result must be tainted to avoid under-tainting. DTA++ uses symbolic execution to identify such transformations [40].

To continue we will now cover *TaintCheck* [38] a system for automatic detection and analysis of *overwriting attacks* as well as signature generation of exploits for commodity software. This system is interesting because it can be deployed on commodity software with no source available, indeed, it performs runtime binary rewriting to implement its DTA. The runtime binary rewriting is used to insert additional instructions that are used to perform the tracking of tainted data. TantCheck's default approach is designed to identify and prevent format string attacks as well as overwrite attacks that aim to modify pointers used as return addresses, function pointers, or function pointer offsets. TaintCheck also

implements a signature generation system to register new incoming attacks [38].

Our third choice is a platform-independent dynamic taint analysis for javascript [42] based on code instrumentation. Most existing DTA for JavaScript is implemented by modifying the JavaScript interpreter, however, this comes with the drawback that the analysis becomes platform-dependent. Without getting into too low-level details, the authors use code instrumentation to keep track of taints during runtime by maintaining a mirrored execution stack. This technique involves a larger overhead than the more classical modification of the interpreter, however, this resolves the problem of portability since it can be installed on any system using JS [42].

DECAF++ is an elastic whole-system dynamic taint analysis that strives to perform taint analysis as least frequently as possible to reduce overhead while maintaining accuracy [43]. DECAF++ is a prototype implemented over DECAF which is the whole-system dynamic taint analysis, the contribution of DECAF++ is to find software optimization that reduces the overhead induced by taint propagation and taint status checking [43].

Finally we have *Taintdroid* a system-wide dynamic taint tracking and analysis system for Android smartphones [6]. Taintdroid is designed to track sensitive data flow inside third-party applications originating from the Android Play Store. Those third-party applications are considered to be untrusted, even if the user gave the authorization to use some sensitive data. Taintdroid is implemented at four different levels of granularity by modifying the Android Virtual Machine, we will not go into the details of those levels, the one that interested us since it is similar to the previously discussed topics is the variable level taint tracking. This taint analysis is performed by modifying the Dalvik VM. This VM is responsible for executing Android application compiled source code. With his different levels of taint analysis, TaintDroid reports misbehaving third-party application that leaks data to the web with no justification [6].

3.4.5 Limitations

We have already talked about the benefits and the limitations of dynamic analysis in a previous section 3.2.2. It goes without saying that they apply to dynamic taint analysis since DTA is a specific kind of dynamic analysis.

In the previous section we saw several implementations of DTA that try to mitigate some of its inherent limitations. One of those limitations is the overhead induced by the implementation of DTA. Since DTA consist of the tracking of

data during the runtime of a code it is obvious it will cause significant time and memory overhead. Solutions to attenuate those have been explored by existing implementation from the previous section and have been summarised in table 3.1. Indeed, Taintdroid aimed to be deployed in real-time and thus optimised their analysis to have a minimum overhead, regardless it still has some notable overhead of 32% on CPU-bound microbenchmark with JIT enabled [6]. DECAF++ managed to get a small overhead but needed to implement a substantial amount of optimization and a special implementation [43] as well. Without those optimizations, the overhead can make the analysis unusable in real-time.

We also talked about implicit flows and the under-tainting and over-tainting implied by those. We saw that it is quite difficult to find taint propagation policies that solve this problem. Most of the reviewed papers that dealt with those use other analysis techniques to overcome this limitation. DTA++ uses symbolic execution to identify implicit flows that must be tainted for instance. Other papers suggest using static analysis techniques to identify those implicit flows before performing the DTA, however, this requires access to the source code.

The last noteworthy limitation is the portability of the DTA implementation. The portability of the solution heavily relies on the techniques used to implement the analysis. We saw that some solutions decided to modify the interpreter of the environment for which they build the analysis, however, those are not very portable since the analysed software must use the modified interpreter thus a simple update could break the analysis implementation. Code instrumentation can counter this problem but implies a bigger overhead.

In conclusion, we see that those inherent limitations can be addressed by adapting the implementation of the DTA but rarely together. We can see that to address most of the limitations this is not sufficient and other techniques must be used with dynamic analysis such as static analysis or others.

3.5 Duality and synergies of static and dynamic analysis

To overcome some limitations of both techniques some systems implementing hybrid approaches have been proposed. Michael D. Ernst [44] suggests hybrid techniques that merge static and dynamic analyses. This approach requires compromising the accuracy of dynamic analysis and the soundness of static analysis.

However, doing so can lead to the mitigation of some limitations of both techniques. Wei S. et al. proposed such a hybrid technique [45].

Chapter 4

WebAssembly

As we know TaintWasm is implemented to analyze WebAssembly binaries, in this chapter we will cover what WebAssembly is, why is WebAssembly a promising technology for other domains than the web and more particularly for IoT and edge computing in general. We will cover how we decided to use WebAssembly for our theoretical hybrid Iot system and what impact was induced by such choices. Finally, we will cover Wasabi the dynamic analysis framework for WebAssembly we decided to extend to implement TaintWasm. We will not go into low-level details in this chapter, we will cover those aspects in Chapter 6 alongside the concrete implementation of TaintWasm.

4.1 WebAssembly Overview

First what is exactly WebAssembly? It is a low-level assembly-like language that has been designed to be hardware and platform-independent [46]. It is a compilation target for various languages, for instance, C/C++, that runs with near-native performance, enabling deployment on the web for client and server applications [47]. The main compiler for WebAssembly is Emscripten, it is a toolchain used to compile languages using LLVM infrastructure into WebAssembly binaries. Emscripten has a lot of convenience tools to generate default JavaScript harnesses so the compiled code is easily run into a web browser.

4.2 Why using WebAssembly

Nowadays, it is easy to see that more and more complex Web applications using vast amounts of data including sensitive data are becoming widespread thus making the security of code always a very important matter [46]. JavaScript is currently the only natively supported language on the Web, WebAssembly does not try to replace

it but adds missing features to it by providing safe and portable low-level bytecode. Some previous attempts to provide low-level Code for the Web were made but had security, performance, or portable issues. In our case, WebAssembly does not interest us for its completion to the web environment but the features it implements are very useful outside this environment. Indeed, portability and security are very important in our hybrid IoT architecture. Furthermore, WebAssembly is hardware-independent, this feature is perfect for the IoT environment.

4.2.1 Advantages of using WebAssembly in IoT systems

As we said the portability and security of WebAssembly enable its use in another context like in an IoT environment. Indeed, some research papers already propose using WebAssembly on the edge, since the performance of WebAssembly is nearly the same as the native languages used to compile the WebAssembly binary, it is very useful to create efficient and portable applications [3] [5]. Those advantages of WebAssembly fit adequately for the IoT context as well. Indeed, some IoT systems already use WebAssembly [4] [48].

4.3 Impact of WebAssembly in our theoretical architecture

As we stated earlier our objective is to assess the viability of employing dynamic taint analysis for detecting and mitigating data leaks within hybrid Internet of Things systems. Given the absence of suitable real-world hybrid IoT systems for our experimentation, we have chosen to develop our system for WebAssembly. The selection of WebAssembly as our platform is motivated by its portability, robust security features, and performance. This makes it a fitting candidate for emulating a hybrid IoT environment.

To fit inside a coherent theoretical hybrid IoT system we decided that WebAssembly would be used by the third-party applications of the system. This lets us take advantage of the security features of WebAssembly. Thus our tool aims to ensure that those third-party applications do not try to steal sensitive data and push it to the web.

4.4 Wasabi

In this section, we will talk about Wasabi, a general-purpose framework for dynamically analyzing WebAssembly. It provides a high-level API, that allows the

implementation of heavyweight dynamic analyses that can monitor all low-level behavior. The approach is based on binary instrumentation, which inserts calls to analysis functions written in Javascript into a WebAssembly binary. This tool is a fairly new prototype thus some bugs can still be found [7].

We choose to base TaintWasm on Wasabi since implementing a dynamic taint analysis framework from scratch is a major challenge and takes a considerable amount of time. Extending an existing framework makes the implementation feasible within the time limits we had.

4.4.1 Our contribution to Wasabi

Wasabi is still a prototype tool, thus we had the opportunity to contribute to it. During the implementation of TaintWasm, we found some bugs that lead to wrong results in our analysis. Those bugs were challenging to spot since we had to discover they were not due to our implementation but due to Wasabi itself. However, we found them and made two pull requests that have been accepted by the owner of Wasabi. The first adjustment made involves revising a specific condition from being "===1" to "!==0." This alteration aligns with the fundamental understanding that positive values are treated as true, while only the value 0 is considered false. This adjustment ensures a more accurate and consistent interpretation of truthfulness within the context of the specified hooks [49]. The second pull request involves adding a new test for the basic taint analysis example of Wasabi and making a correction to the select instruction [50].

Chapter 5

TaintWasm design

In this chapter we will cover the design choices we made to implement TaintWasm. As we explained in the context chapter, we wanted to implement a dynamic taint analysis tool for an Iot hybrid architecture. We choose this architecture because it seems promising for privacy by design in smart homes. Since such architecture is not currently widely used and it is very challenging to write a dynamic taint analysis system from scratch, we searched for another approach. We thus found that WebAssembly is a promising technology that could be used in the context of IoT thanks to its inherent security and portability. Fortunately, though WebAssembly is a quite recent technology, we found a dynamic analysis framework that we could extend to implement TaintWasm, to know Wasabi. In this chapter, we will first cover the objectives we set to make our proof of concept. Then we will cover the environment in which TaintWasm was implemented. And finally, we will cover the general structure of TaintWasm.

5.1 TaintWasm Objectives

As we already stated TaintWasm has been created to be a proof of concept. We try to implement a dynamic taint analysis framework to prevent third-party applications to steal sensitive data in a smart home environment. However, it was not possible to create a whole smart home with third-party applications as it would take years. Instead, we decided to focus on the aspect the dynamic taint analysis system would cover. To be deployed in such an environment the system should be portable, that is why we choose to use WebAssembly. Since the usage of Wasabi induced some strong overhead we did not focus too much on making TaintWasm runnable in real-time. Managing implicit flows is our primary area of focus. However, achieving success with just this technique has proven to be quite challenging when compared to other dynamic taint analysis approaches.

5.1.1 Implicit flow handling

We have decided to address implicit flows, as they can be used to leak data without detection by the receiving sink. Indeed, it is possible to clean tainted data with such flows. The code 5.1 is a simple example of such a case. Simply monitoring explicit flow is not enough to prevent such occurrences. Our primary goal was to determine if using pure DTA techniques could prevent the leakage of sensitive data, which required addressing the issue of implicit flow. When it comes to other systems, relying solely on pure DTA can cause noticeable over-tainting. We found several ways to use implicit flow to reproduce tainted data value into untainted data, the first was using implicit flows induced by control-dependent statements. And another was to use those blocks indirectly, we found such cases with compiler optimization. Nevertheless, we found taint propagation policies to cover those cases, as we know they induced a huge amount of over-tainting. Due to the inevitable issue of over-tainting in pure DTA cases, we have chosen to divide our analysis into three levels of sensitivity. To identify potential misuse of sensitive data, there are three levels of sensitivity. The first level detects explicit flows that clearly show misuse of sensitive data. The second level detects simple implicit flows, which suggest potential misuse of sensitive data. The third level detects indirect implicit flows, which taint any data that has been in any way influenced by tainted data. Further investigation is required to confirm any misuse of sensitive data in the last case.

Listing 5.1: Copy data without explicit flows

```
1  a = 0
2  b = get_input()
3  while a != b then
4      a++
5  push_to_cloud(a)
```

To exemplify the concepts outlined in the preceding paragraph, we undertake an analysis of three distinct code instances that share a common objective but exhibit varying characteristics concerning explicit and implicit tainting.

Listing 5.2: Simplest explicit flow

```
1  a = get_data()
2  push_data(a != 0)
```

The code segment labelled 5.2 embodies the most straightforward explicit flow scenario. Here, the variable `a` is derived from a data source, and its value undergoes a non-equality check against zero ($a! = 0$). The outcome of this comparison is then tainted and subsequently pushed.

Listing 5.3: Simplest implicit flow

```
1  a = get_data()
2  if a == 0 then
3      push_data(0)
4  else
5      push_data(1)
```

The code excerpt designated as 5.3 serves as an illustration of the simplest implicit flow. The value of the variable `a` is obtained from a data source, and an `if` condition evaluates whether `a` is equal to zero. Depending on this condition, either a zero or a one is pushed to the destination. In this case, the tainting is implicit, as the pushed value is contingent on the condition governed by the presence of tainted data.

Listing 5.4: Simplest potential implicit flow

```
1  a = get_data()
2  if a = 0 then
3      push_data(0)
4      return
5  push_data(1)
```

The code identified as 5.4 illustrates the simplest scenario of a potential implicit flow. When the value of the variable `a` is zero, the code pushes a zero to the destination and immediately encounters a return statement. In this specific case, the tainting can be considered implicit, as the pushed value is dependent on a condition with tainted data. However, when the value of `a` is not zero, the code pushes a one to the destination. This scenario presents a potential implicit flow, as the dynamic analysis cannot ascertain in advance the presence of the return statement. Given this uncertainty, the policy cautiously treats this as a potential implicit flow, considering the possibility of hidden implicit characteristics.

5.2 Environment

In this section we will explain the overall architectures of the theoretical smart home system for which our DTA tool has been developed. As we explained earlier smart homes system have a wide variety of architectures and designs, from cloud only to hub only, with the support of third-party applications or not, with proprietary devices or non-proprietary ones etc. Since our architecture is theoretical some abstractions have voluntarily been made since the aim of TaintWasm is not to be deployed as an industrial solution but to be a proof of concept. Is dynamic taint analysis a reliable method for detecting and preventing data leaks in an IoT system?

Before delving into the specifics of our design decisions, we would like to emphasize that, as this is a theoretical framework, the sensors and features typically found in a real IoT system are being simulated. The aim of this architecture is not to build a whole new IoT system but to give a concrete context for TaintWasm. Thus our implementation environment corresponds to the following architecture but does not run real sensors and applications. Now we will explain this environment, we based our model on the hybrid architecture, with a smart hub that allows the installation of multiple applications whether they are proprietary or published by third-party developers. Those applications must go through an API to access the data from devices and sensors that are available in the smart home. Those applications must also go through another API to access the cloud. The supplier of the devices and sensors is not precise in our solution, we decided to emulate them but in a real smart home context, it would be the job of the API to gather their data and make them available to applications of the hub. We do not dive into the details of the cloud applications themselves either since TaintWasm is set up inside the hub, thus the security of the cloud applications is out of the scope of our solution. A very important point about our design is at the level of the applications themselves, we impose with the API that sensitive data is only available within embedded WebAssembly code, we explained the reason for this choice in the chapter about WebAssembly.

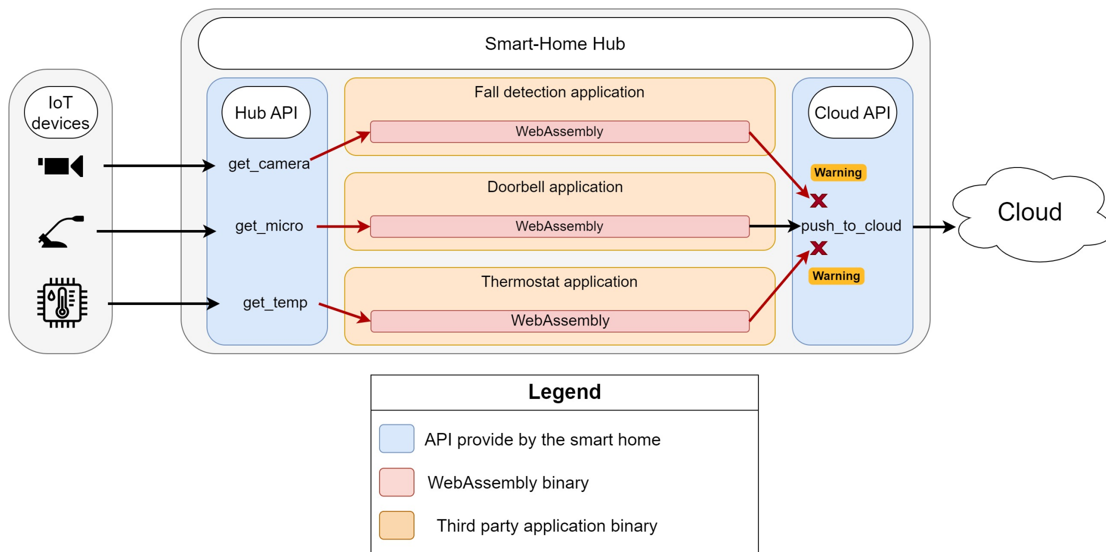

Figure 5.1: Illustration of our theoretical architecture environment

5.2.1 Trust model

We will now describe the trust model of this system. Since our main objective in this document is to present a new taint analysis tool for WebAssembly binaries in the context of a smart home system, we decided to focus our attention specifically on the application part of the system. So in our architecture, we consider that the devices and the sensor are correctly handled by the provider and do not represent a potential source of data leaks. This is not the case for the cloud application and the cloud provider, we consider that the cloud is an untrusted zone where raw sensor data should not get, thus our system aims to detect potential data leaks and data that has been influenced by raw sensor data. The hub APIs are trusted, we consider that the hub providers are honest and do not try to steal sensitive data.

5.2.2 WebAssembly execution environment

We decided to force applications to process data inside WebAssembly code, this has been chosen to take advantage of the inherent securities and the portability of WebAssembly. Doing so we have an environment in which data is not accessible without the use of the given API and it is not possible to send data to the internet without getting out of the WebAssembly. In fact, data could be accessed and exported out of the WebAssembly binary without going through the API, however, thanks to Wasabi we can monitor every imported and exported function of the binary, thus we report usage of those functions that are not accessed through the API. We chose to consider the usage of such import and export malicious by default. Even though this is not the primary usage for which WebAssembly has been created, since the Web context imposed a lot of strong requirements to this compilation target it has made it very useful at the edge too.

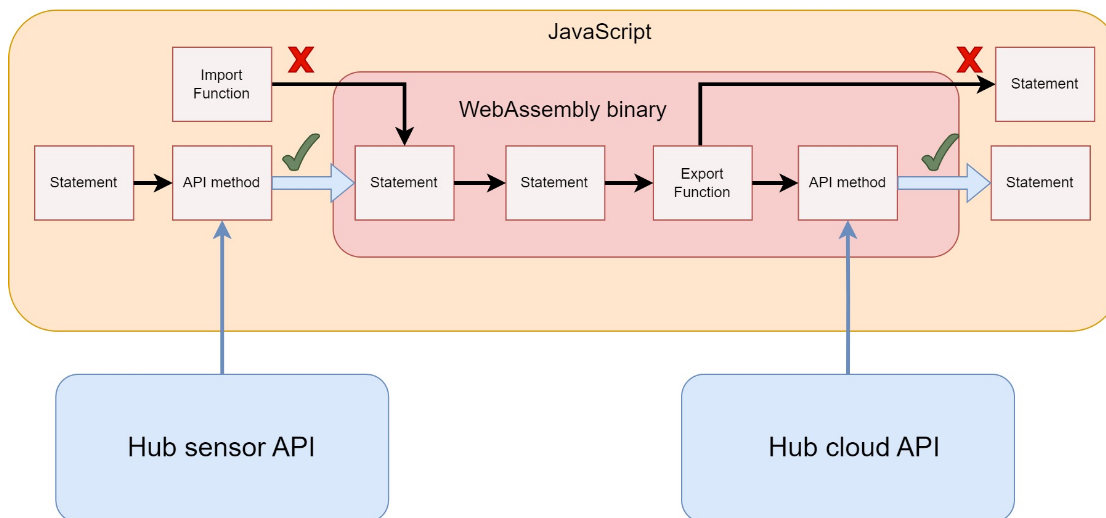

Figure 5.2: Interdiction to import and export function from Javascript to WebAssembly

5.2.3 The APIs

Our API in Javascript is where all contact to the outside of the WebAssembly code is made. Since TaintWasm has for objective to serve as proof of concept, our API does not make real contact with the exterior nor get real information from sensors. Nevertheless, the implementation of that real behaviour would be done by modifying solely the JavaScript API and not the analysis implementation nor the WebAssembly binaries. As we explained earlier, WebAssembly binaries are embedded into a JavaScript code that runs into a web browser. Even if WebAssembly seems promising to be used outside the web context for now it is the easiest environment to develop TaintWasm.

API function	DTA
get_data	source
push_data	sink
get_data_dummy	source
push_data_dummy	sink
print	No surveillance

Table 5.1: API calls and its taint analysis function

We present the API used in TaintWasm, which encompasses five methods. The first method, "`get_data`", retrieves sensor data based on a provided string representing the sensor's name and returns the data in any format. The second method, "`push_data`", requires two string parameters: one indicating the destination address and the other representing the data to be pushed. There are two auxiliary methods: "`get_data_dummy`" and "`push_data_dummy`". The "`get_data_dummy`" method simply returns an integer value, while the "`push_data_dummy`" method accepts a single integer parameter for data pushing. Additionally, the "`print`" method is introduced, designed to serve as an output mechanism for any data, without introducing potential danger. This is illustrated in the figure 5.3

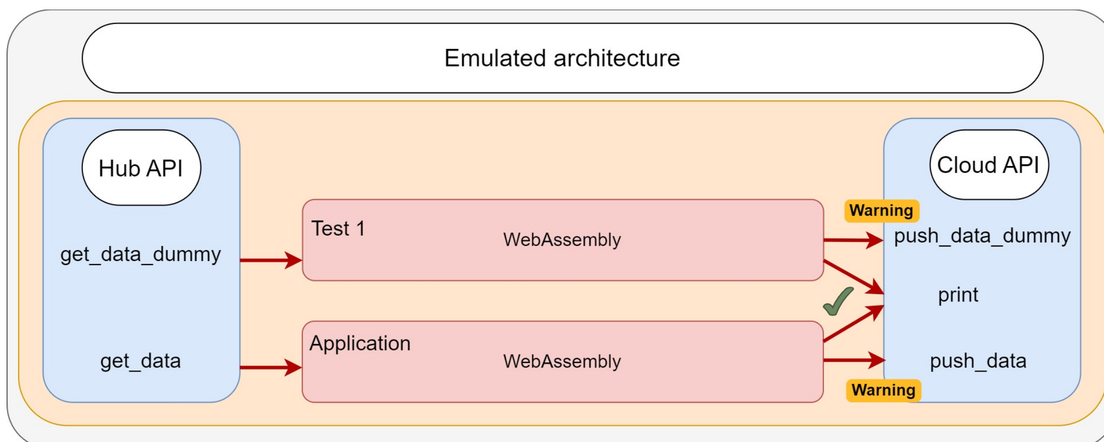

Figure 5.3: Illustration of our emulated theoretical architecture environment

5.3 TaintWasm dynamic taint analysis structure

As per the information provided in Chapter 3, a dynamic taint analysis system works by tainting data at a taint source, propagating taints according to propagation policies then checking taints at a taint sink. In this section, we shall provide a comprehensive overview of how we executed the aforementioned aspects for TaintWasm.

Taint sources As explained in the section about the environment of TaintWasm. We decide to implement TaintWasm for WebAssembly binaries embedded inside applications for a smart home. Those applications can access sensor data only through the API given by the system. Thus taint sources in our systems are situated at the API calls to get sensor data.

Taint sinks In direct link with how we choose to implement our taint sources, we implemented our taint sinks at the API calls to "push" data out of the application.

5.3.1 Taint propagation policies

In our taint propagation methodology, we employ three distinct policies: the explicit policy, the implicit policy, and the potential implicit policy.

Explicit policy The explicit policy represents the simplest and most direct approach to taint propagation. It encompasses data propagation through direct interactions such as binary and unary operations, as well as variable assignments. When data is tainted using this policy, it unequivocally signifies that the tainted data is derived from sensitive data. This straightforward policy ensures unambiguous tainting of relevant information.

Implicit policy The implicit policy comes into play when data is influenced by sensitive information but does not undergo direct transformation. Consider, for instance, a conditional statement, as exemplified in code snippet 5.3. Here, the exported value is not a direct modification of the imported value but depends on it. When data is tainted using the implicit policy, it indicates that the data conditionally relies on sensitive information. However, this policy recognizes that such dependency might not always have significant implications.

Potential implicit policy The potential implicit policy addresses nuanced scenarios, particularly edge cases. It accounts for situations where code transformations, involving conditional blocks and return statements, can modify the tainting status of data. For instance, in code snippet 5.4, the code following the "if" statement is conceptually equivalent to an "else" block, but distinct in terms of tainting. Consequently, the implicit policy would not taint it. This policy tends to over-taint, ensuring a cautious approach that avoids under-tainting. The absence of tainting by the potential implicit policy provides a clear indication that the data is unrelated to sensitive information.

In summary, our taint propagation framework incorporates these three policies, each serving a distinct purpose: explicit tainting for direct derivations, implicit tainting for conditional dependencies, and potential implicit tainting for handling edge cases. These policies collectively enhance our ability to analyze and track the flow of sensitive data within the codebase.

To provide a comprehensive view of the risks associated with each taint propagation policy, we present a risk assessment in Table 5.2. This assessment evaluates

the over-taint and under-taint risks inherent to each policy. Notably, the explicit tainting policy exhibits a low over-taint risk but a high under-taint risk, making it suitable for identifying direct data flows but potentially missing conditional dependencies. The implicit tainting policy balances risks with a medium level of over-taint and under-taint risk. On the other hand, the potential implicit tainting policy prioritizes capturing edge cases with a high over-taint risk, while maintaining a low under-taint risk, which is essential for ensuring comprehensive data leak detection in complex scenarios.

Policy	Over taint risk	Under taint risk
Explicit	Low	High
Implicit	Medium	Medium
Potential implicit	High	Low

Table 5.2: Risk Assessment of Taint Propagation Policies

5.4 Challenges

The objectives we have set ourselves comes with their challenges when we implemented TaintWasm. The first was to understand WebAssembly in enough detail to create a taint analysis system for its binaries. The following challenge was to install, run and debug Wasabi, as it is a prototype and our use of it was very sharp we identified and corrected some very specific bugs. Another significant challenge was to find examples to test our DTA exhaustively, indeed since the theoretical architecture we consider is not widely used yet, it was not possible to find applications to test our technique. We then add to make our testing as precise as possible to correct TaintWasm. The last challenge was to deal with the compiler optimizations. As for most compilers, the one of WebAssembly performs a lot of optimization in comparison with the source code, thus we had to identify and manage every compilation optimization for our analysis to work.

Chapter 6

TaintWasm implementation

This chapter delves into the technical details of TaintWasm implementation. As the context in which it operates is new and quite complex, we need to first explain some basics. Firstly, we will explain how WebAssembly execution works, followed by the WebAssembly text format (WAT), which is a human-readable format of WebAssembly binaries. During the implementation of TaintWasm, we used this format to write our tests as analysis is performed during runtime on binaries, requiring a deep comprehension of how WebAssembly is executed. We will then return to Wasabi and explain the hooks used in our dynamic taint analysis (DTA) implementation and how Wasabi performs its instrumentation. This will lead us to the implementation of TaintWasm itself. We will begin by explaining the principle behind the analysis and then illustrate it through a step-by-step execution. For simplicity, we will only cover explicit flows in this execution and then move on to an example of implicit and potential implicit flow. Finally, we will discuss implementation challenges.

6.1 Technical prerequisite

Inside the section we will explain the important technical concept to comprehend to understand the implementation of TaintWasm. This section is not exhaustive on the discussed topics, it aims to give the minimum matters to understand to following sections. For further information about WebAssembly and Wasabi refer to the official documentation [47] [7].

6.1.1 WebAssembly Execution

TaintWasm relies heavily on the execution model of WebAssembly. It is a crucial aspect that cannot be overlooked. A comprehensive understanding of WebAssembly

execution is crucial for the successful implementation of dynamic taint analysis within this context.

WebAssembly serves as a virtual machine, providing a secure and efficient runtime environment for web applications. It enables the execution of code within modern web browsers while maintaining essential principles of security, portability, and performance. The execution process of a WebAssembly module encompasses several pivotal stages:

1. **Module Loading:** Upon retrieval from a server or local cache, the WebAssembly module, typically in binary form, undergoes validation to ensure conformance with the WebAssembly specification, adhering to safety and security constraints.
2. **Compilation:** Following successful validation, the module is compiled into a lower-level representation akin to machine code, forming the basis for subsequent execution.
3. **Instance Creation:** The instantiation of a WebAssembly module results in the establishment of an execution instance. This instance encompasses functions, memory, tables, and other relevant resources necessary for the module's operation.
4. **Execution:** The core execution of WebAssembly code occurs within a stack-based virtual machine. A structured call stack is maintained, accompanied by a set of instructions that operate on this stack. This design enhances the efficiency and compactness of the execution environment.

6.1.2 WebAssembly Text Format (WAT)

For the sake of human-readable representation and testing, WebAssembly introduces the WebAssembly Text format (WAT). This textual representation serves as a more accessible counterpart to the binary WebAssembly format, enabling enhanced comprehension, debugging, and developer interaction. This format is demonstrated in Listing 6.2 and the equivalent C code in Listing 6.1.

To work with TaintWasm, it is important to have a deep understanding of the WebAssembly text format. The extensive use of WAT is a crucial element in creating tests and enables dynamic taint analysis while running WebAssembly binaries. This understanding of WebAssembly's textual representation forms the base upon which the TaintWasm framework operates.

Listing 6.1: Simple C code Example

```
1  int add_two(int value) {
2      return value + 2;
3  }
4
5  int main() {
6      int value = add_two(5);
7      if (value > 2) {
8          return 15;
9      }
10     return value;
11 }
```

Listing 6.2: Equivalent WebAssembly text format code

```
1  (module
2      (func $add_two (param $value i32) (result i32)
3          local.get $value
4          i32.const 2
5          i32.add
6      )
7      (func $main (result i32)
8          (local $value i32)
9          i32.const 2
10         call $add_two
11         local.set $value
12         local.get $value
13         i32.const 2
14         i32.gt_u
15         if
16             i32.const 15
17             return
18         end
19         local.get $value
20         return
21     )
22 )
```

The following table 6.1 outlines key categories of instructions within the WebAssembly Text format along with representative instructions and their explanations:

Table 6.1: WebAssembly Text Format instructions list

Category	Instruction	Explanation
Control Flow	block	Establishes a code block with a defined scope.
	br	Unconditionally jumps to a specified label.
	call	Invokes a function by index.
	drop	Discards the top stack value.
	end	Marks the end of a block, loop, or if statement.
	if	Conditionally executes a block if the top stack value is true.
	loop	Creates a loop that repeats a block.
	return	Returns control from a function to its caller.
	select	Selects a value based on the condition on the stack.
Numeric Instructions	const	Pushes a constant value onto the stack.
	eq, neq, gt, lt, ge, le	Perform comparisons between values on the stack.
	add, sub, mul, div, rem	Basic arithmetic operations.
Memory Instructions	load	Loads a value from memory to the stack.
	store	Stores a value from the stack into memory.
Variable Instructions	set	Sets the value of a local or global variable.
	get	Pushes the value of a local or global variable onto the stack.
	tee (local only)	Copies the value of a local variable, leaving it on the stack.

6.2 Wasabi instrumentation and hooks

Here we will provide a brief insight into the functioning of Wasabi's instrumentation approach. Wasabi's instrumentation involves the incorporation of hooks within the WebAssembly binary. These hooks serve as entry points that enable the integration of analysis logic during the execution of WebAssembly code.

Wasabi introduces a set of hooks strategically positioned within the WebAssembly binary to facilitate the seamless integration of analysis. Each hook is designed to correspond with specific instructions or control flow constructs, allowing for the invocation of analysis logic at key points in the code's execution. The table 6.2 below provides an overview of some essential hooks employed by Wasabi, their associated instruction or control flow constructs, and a brief description of their functionalities:

Hook Name	**Instruction Name**	**Functionality Description**
start	n/a	Represents the module initialization point.
if	if	Initiates a conditional branch based on a value.
br	br	Unconditionally transfers control to a labeled target.
begin	block, if, function, loop, else	Indicates the beginning of various structured control constructs.
end	end	Marks the conclusion of structured control constructs.
drop	drop	Removes the top value from the stack.
select	select	Conditionally selects a value from the stack.
call_pre	call	Signals the pre-invocation of a function.
call_post	call	Signals the post-invocation of a function.
return	return	Transfers control back to the caller of the current function.
const	i32.const, ...	Pushes a constant value onto the stack.
unary	i32.eqz, ...	Performs unary operations on values in the stack.
binary	i32.eq, i32.add, ...	Executes binary operations on stack values.
local	local.set, local.get, local.tee	Manages local variable's values in the stack frame.
global	global.set, global.get	Interacts with global variables in the module.

Table 6.2: Essential Hooks Employed by Wasabi Instrumentation

6.2.1 Instrumented Code and Analysis

The code in WebAssembly text format in listing 6.3 has two distinct functions: "*add_two*" and "*main*." The former is a straightforward addition operation involving a given value and the constant 2, while the latter, function "*main*," orchestrates a sequence of computations based on value comparisons. Notably, the core functionality of the code remains consistent across both instrumented (listing 6.3) and non-instrumented (figure 6.1) instances.

The process of code instrumentation is done thanks to Wasabi. Within the context of the instrumented code, Wasabi strategically introduces hooks, positioned in correspondence with specific instructions embedded within the codebase. The strategic positioning of these hooks embodies a dual aspect: the function number, which uniquely identifies the corresponding function, and the instruction number, signifying the position of the instruction within that function.

Upon initiating function execution, a "begin_function" hook comes into play. This hook is denoted by an instruction number of -1, effectively indicating its positioning before any instruction within the function. The "*add_two*" function includes a "return" hook even though there is no explicit "return" instruction. This highlights how this function naturally generates a return value. In contrast, the "return" hook nested within the "if" block aligns with a return instruction, thereby underlining its specific inclusion. Another observation is on the "end_function" hook. This particular hook consistently maintains the same instruction number within a given function. We obverse that behaviour at the end of the "if" block where the "end_function" hook is placed and has an instruction number of 11.

Each hook triggers the activation of a corresponding JavaScript function nested within the analysis framework. This is TaintWasm's core principle. The TaintWasm framework is implemented upon this intricate interplay of instrumentation and analysis.

In brief, the instrumented code effectively integrates Wasabi's hooks to implement the dynamic taint analysis. These hooks, aligned with specific instructions, serve as pathways to JavaScript functions embedded within the analysis structure. This instrumentation and analysis enable the tracing of data flow across the code.

Listing 6.3: WebAssembly text format code before instrumentation

```
1  (module
2      (func $add_two (param $value i32) (result i32)
3          local.get $value
4          i32.const 2
5          i32.add
6      )
7      (func $main (result i32)
8          (local $value i32)
9          i32.const 2
10         call $add_two
11         local.tee $value
12         i32.const 2
13         i32.gt_u
14         if
15             i32.const 15
16             return
17         end
18         local.get $value
19         return
20     )
21 )
```

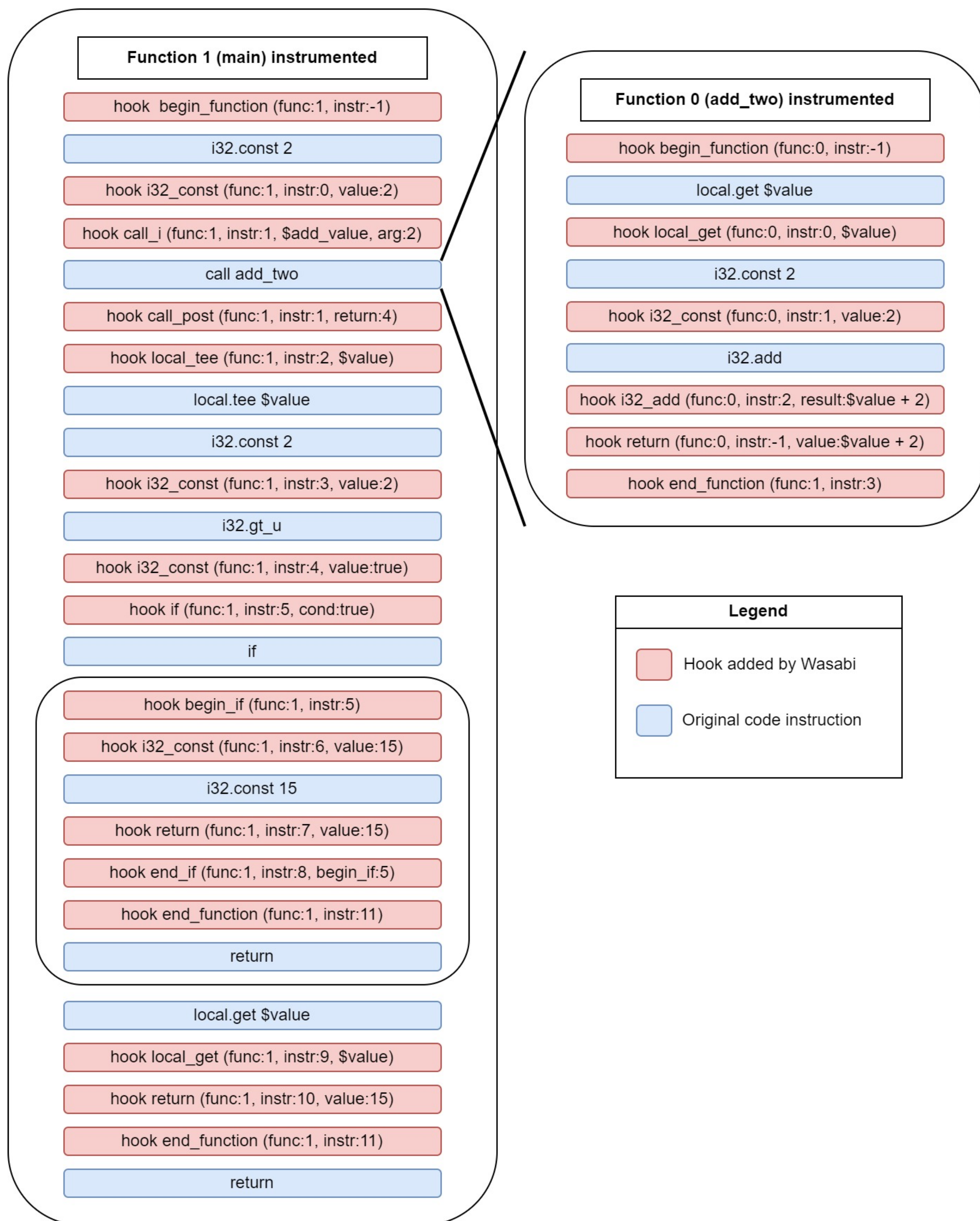

Figure 6.1: Instrumented code

6.3 TaintWasm implementation

To implement TaintWasm, we drew inspiration from a simple example in the Wasabi documentation [7]. Our approach involves emulating the execution of the WebAssembly virtual stack machine within the context of the Wasabi analysis. Unlike certain other Dynamic Taint Analysis (DTA) implementations such as TaintDroid [6], we opted not to modify the interpretation of the binaries. Instead, we leverage code instrumentation provided by Wasabi to achieve our objectives.

In this pursuit, we meticulously replicated the execution stack of the currently analyzed binary using dynamic analysis within the Wasabi framework. Rather than duplicating the exact actions of the binary, we focused on comprehending the behaviour of each WebAssembly instruction, specifically how they interact with the stack. Our emphasis was on identifying the push and pop operations performed by each instruction, subsequently translating them into corresponding actions on our mirrored stack. Within our stack, we exclusively store taint values associated with the respective data. As WebAssembly operates as a stack machine, our approach mirrored this behaviour, but instead of executing operations with concrete data, we emulated the push and pop interactions with the stack while maintaining taint information. This taint information within our mirrored stack is subject to our defined taint propagation policies.

For instance, consider the **add** instruction. When executed, this instruction pops two values from the stack, performs an addition operation, and then pushes the result back onto the stack. Following our taint propagation policy, any binary operation involving at least one tainted data results in a tainted outcome. To incorporate this policy into our analysis, we employ hooks that indicate the occurrence of an **add** instruction. This enables us to extract two taints from our mirrored stack, evaluate if either of them is tainted, and subsequently push a taint, that is tainted if necessary. This process is outlined visually in Figure 6.2, illustrating the sequence of events for this example.

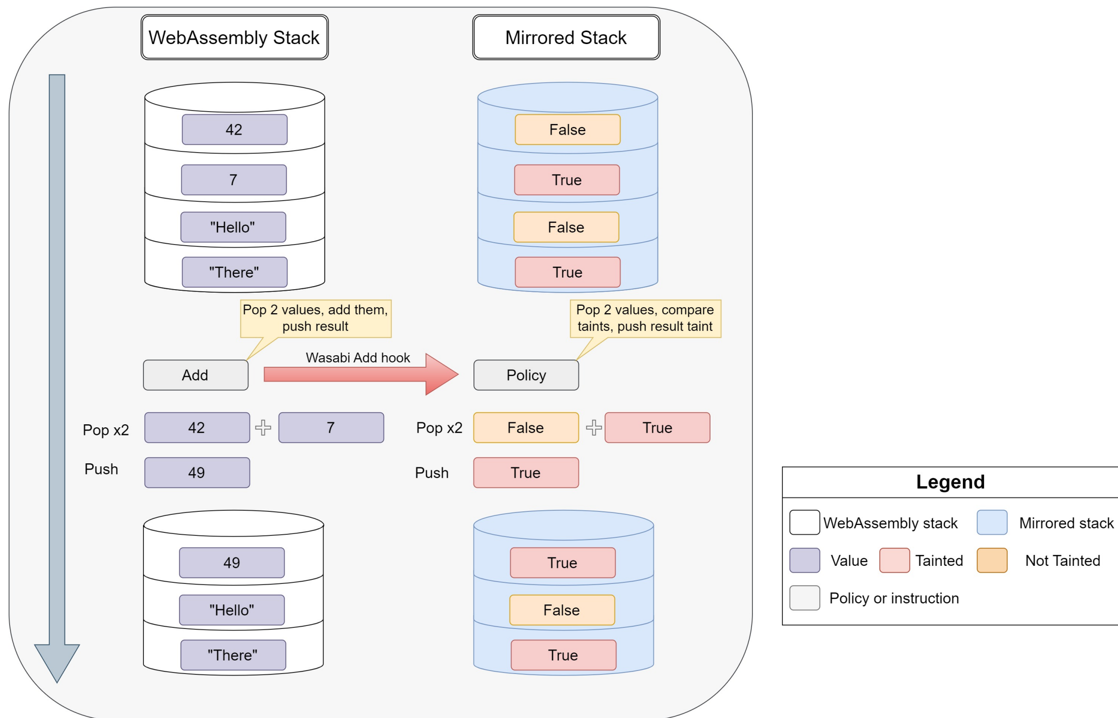

Figure 6.2: High-level single analysis step example

Further in this section we will give more complete examples to illustrate our taint propagation policies for explicit flows, then implicit flows and finally for potential implicit flows.

6.3.1 Mirrored Stack implementation

Now we will explain the concrete implementation of the TaintWasm mirrored stack. Since we are not redoing the entire program execution, we do not retain the variable values and other information. Instead, we only keep track of the taints associated with those values. Doing so we can track and propagate taints according to certain rules that we detailed above. Finally, we can see if tainted values reach the sink or a critical zone and if so signal it.

As we observe in the following code snippet 6.4, the stack itself is an array containing function scopes. Those function scopes are JSON containing an attribute for blocks, locals, a function index and the index of the calling function. Those values are necessary to represent correctly the WebAssembly environment. With the stack, we have an array for global variable taints, and another for dynamically

allocated memory. All those elements represent the WebAssembly stack itself. Other fields in the code below are there for the taint analysis. Indeed, since we implemented the analysis with Wasabi hooks and those have sometimes different possible behaviour depending on the compilation or other factors, we had to add fields in the stack to handle a lot of edge cases. We will explain in more detail those in the challenges section.

Listing 6.4: Complete mirrored Stack implementation

```
1     constructor() {
2         this.stack = [
3             // a function scope
4             {
5                 blocks: [],
6                 // a block scope
7                 // {
8                 //   blockTaint : Taint
9                         //   values : []
10                //   breaking : bool
11                // }
12                locals: [],
13                funcIdx: −1,
14                fromFuncIdx: −2,
15            },
16        ]
17        this.memory = [];
18        this.globals = [];
19
20        this.historyValue = null;
21        this.blockHistory = [];
22
23        this.savedScopeBeforeReturn = null;
24        this.lastFuncToBePop = null;
25        this.lastFuncIdxToBePop = −1;
26        this.idxFromTheSavedScope = −1;
27    }
```

6.4 Flows analysis examples

To clarify our method we will now illustrate the execution of our analysis for explicit flows, implicit flows and potential implicit flows. For the sake of simplicity, we will cover an entire example only for explicit flows with a simplified stack structure. Indeed, a lot of elements in the stack presented in the previous section were introduced to resolve some specific edge cases. We will cover those inside the implementation challenges section. We will not cover complete code analysis

for implicit and potential implicit flows since it would be redundant with explicit flows. Instead, we will show the specific differences between those. Regardless, even though we show those three cases separately with a simplified stack structure to be more understandable and clear, the real TaintWasm implementation performs the three analyses together within the same stack by storing a taint for each case, to know an array with three taints.

6.4.1 Explicit flow complete analysis example

We will quickly review the code in listing 6.5, we are about to analyse with TaintWasm principle. This is a very simple code using an API represented by two imported functions *get_data* and *push_data*, which will respectively be our taint source and sink. In this example, we will observe a clear progression from source to destination, and within this scenario, there is also a manifestation of leakage at the destination point. It is important to note that the API is simplified to easily follow the flow of the taints in these simple examples. Here it only pushes and gets a single value. Now, we will go through the code itself. The main function named *f* will first call *get_data* which will give a value that is considered sensitive. This acquired value is then summed with another value and the result is stored in the local scope of the function. Then this value is pushed on the stack to be used as an argument for the following function call to *propagateArgToSink*. Arguments are directly stored in the local scope of its function. The function will then get back the argument value from its local store and push it to the stack then will call the function *push_data* which will use this value as an argument too. This call to the API is considered to be a leak since the argument of the function is a tainted value.

Listing 6.5: Complete WebAssembly Text Format simple code

```
1  (module
2      (import "env" "get_data" (func $get_data (;1;) (result i32)))
3      (import "env" "push_data" (func $push_data (;2;) (param i32)))
4
5      (func $propagateArgToSink (param i32)
6          local.get 0
7          call $push_data
8      )
9
10     (func $main (local $loc i32)
11         call $get_data
12         i32.const 33
13         i32.add
14         local.set $loc
15
16         local.get $loc
17         call $propagateArgToSink
18     )
19
20     (start $main)
21 )
```

6.4.2 Step-by-step illustrated analysis for explicit flow

Now, we will proceed with the detailed step-by-step analysis to illustrate the explicit taint flow within the code snippet. We will visualize the stack structure and taint propagation at each key point in the execution process. Through these illustrations, we will demonstrate how TaintWasm captures the flow of taints, identifying potential data leaks and enforcing our defined taint propagation policies.

The initial stage of our analysis involves setting up the environment for the *main* function. As depicted in Figure 6.3, we observe the preparation of the execution stack. A function scope and a block scope are pushed onto the stack, laying the foundation for the subsequent execution steps.

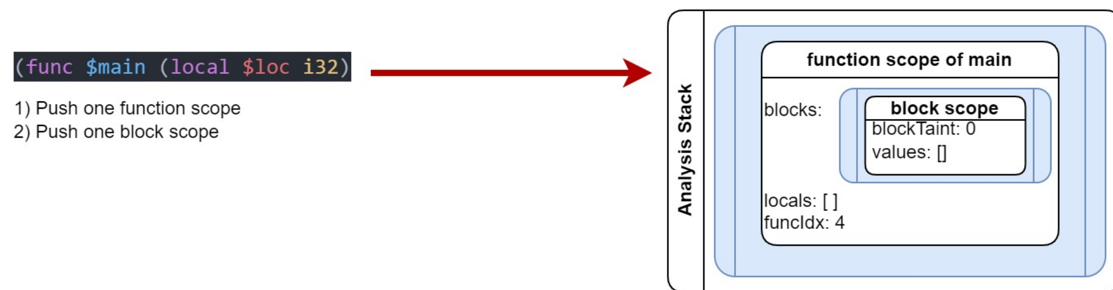

Figure 6.3: Environment setup for the *main* function

Upon invoking the API function *get_data*, the execution process unfolds as follows: it first pushes a function scope and a block scope to establish the function's environment. Subsequently, as the call concludes, the function scope is popped, maintaining the block scope intact. Finally, the function places a value into the calling function's scope. Notably, this value is tainted due to its origin from a taint source. This sequence of events is depicted through the series of figures shown in 6.4 and 6.5.

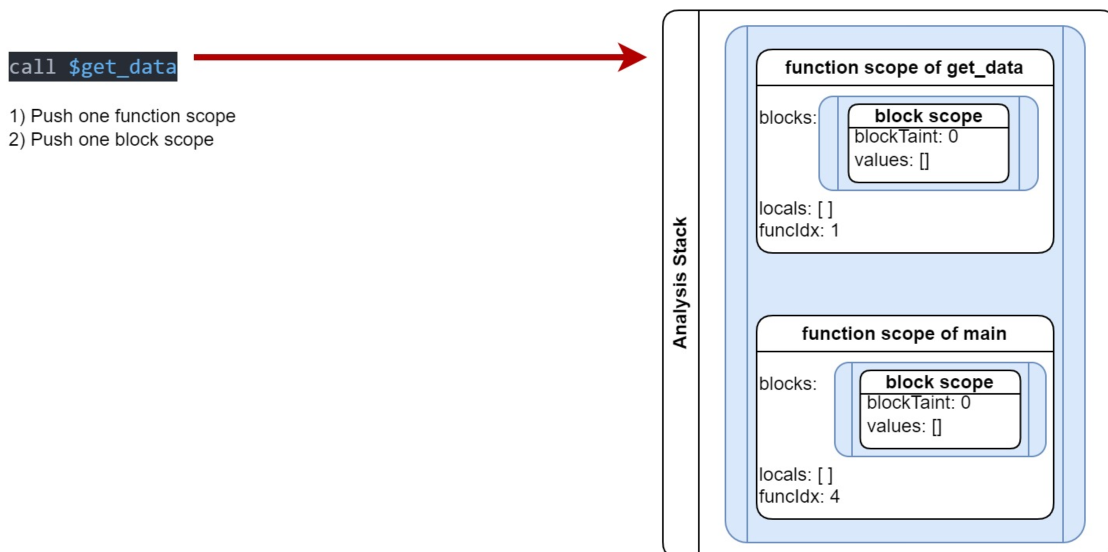

Figure 6.4: Function scope and block scope being pushed during the execution of the *get_data* API call.

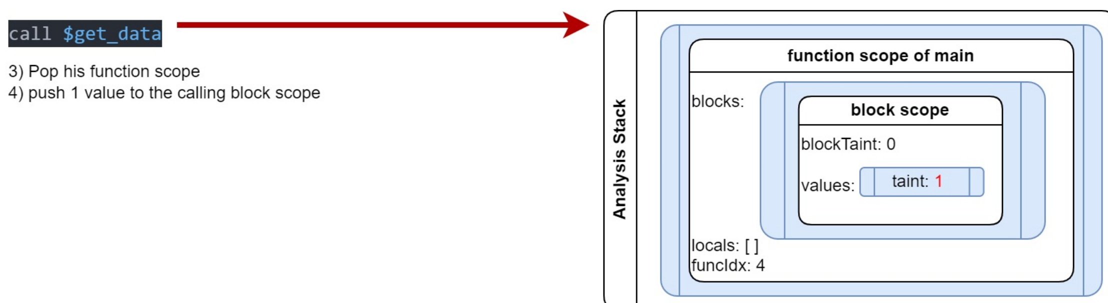

Figure 6.5: Function scope being popped after the execution of the *get_data* API call is completed.

At this juncture, we perform a straightforward action of adding a fresh value to the values present within the block scope. Importantly, this value remains untainted, as it originates from a non-taint source. Refer to 6.6 for a visual representation of this operation.

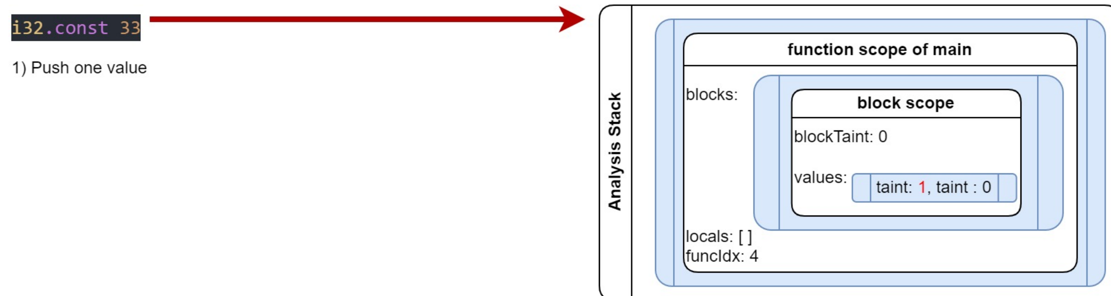

Figure 6.6: Depiction of a new untainted value being added to the values of the block scope.

Subsequently, an *add* instruction is invoked. This instruction entails the removal of two values from the stack, followed by the addition of a new value into the calling block scope. According to our taint propagation policy, if either of the two extracted values is tainted, the resultant value is also considered tainted. Consequently, the value pushed onto the block scope is tainted. For a visual depiction of this process, refer to Figure 6.7.

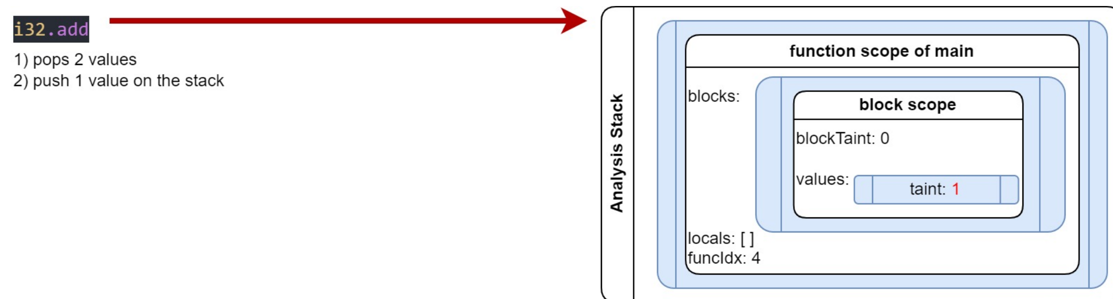

Figure 6.7: *add* instruction's operation

In the subsequent step, the value residing within the block scope is extracted, and then subsequently inserted into the local variables of the main function scope. For a graphical representation of this transition, consult Figure 6.8.

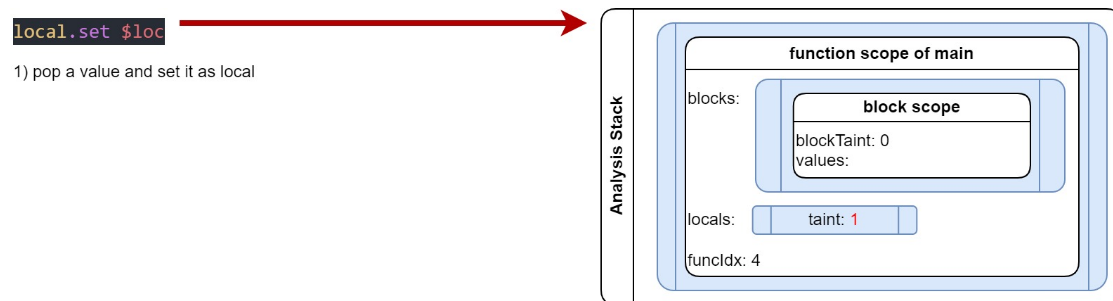

Figure 6.8: Value pushed to the local variables

Subsequently, the `local.get` instruction is executed, serving to duplicate a value from the local variables of a function scope into the values of a block scope. The process is visually depicted in Figure 6.9.

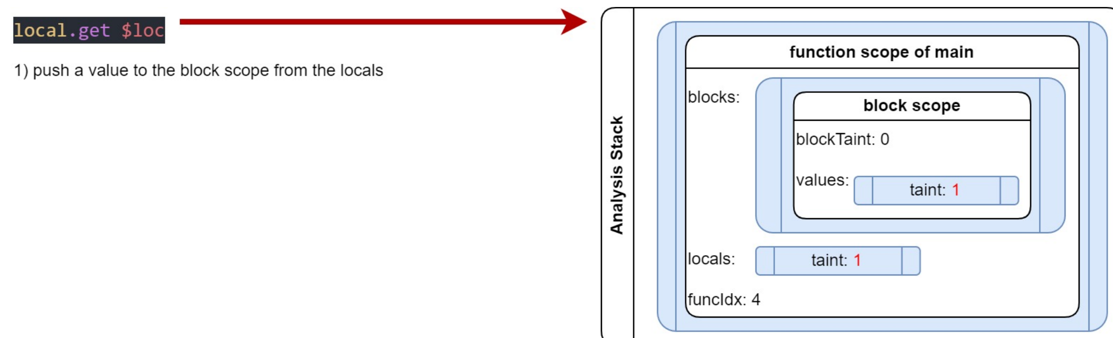

Figure 6.9: `local.get` instruction, copying a value from the local variables to the values of a block scope.

Moving forward, we proceed to the execution of an instruction that initiates a call to the *propagateArgToSink* function, requiring a single argument. As a result, the instruction pops a value from the stack and then proceeds to establish the necessary environment for the *propagateArgToSink* function scope. A visual representation of this process can be observed in Figure 6.10.

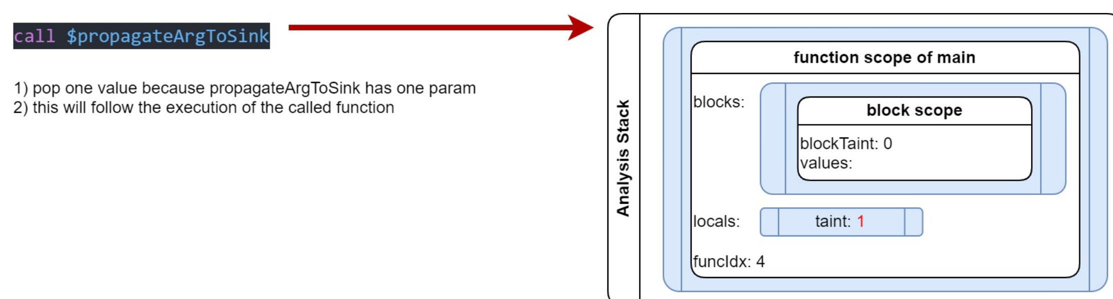

Figure 6.10: Call of the *propagateArgToSink* function

Proceeding further, we transition to the process of establishing the execution environment for the previously invoked function. This involves a sequence of actions: pushing a function scope onto the stack, followed by a block scope, and ultimately inserting the argument's taint into the local variables of the newly created function scope. Visual insight into this mechanism is provided in Figure 6.11.

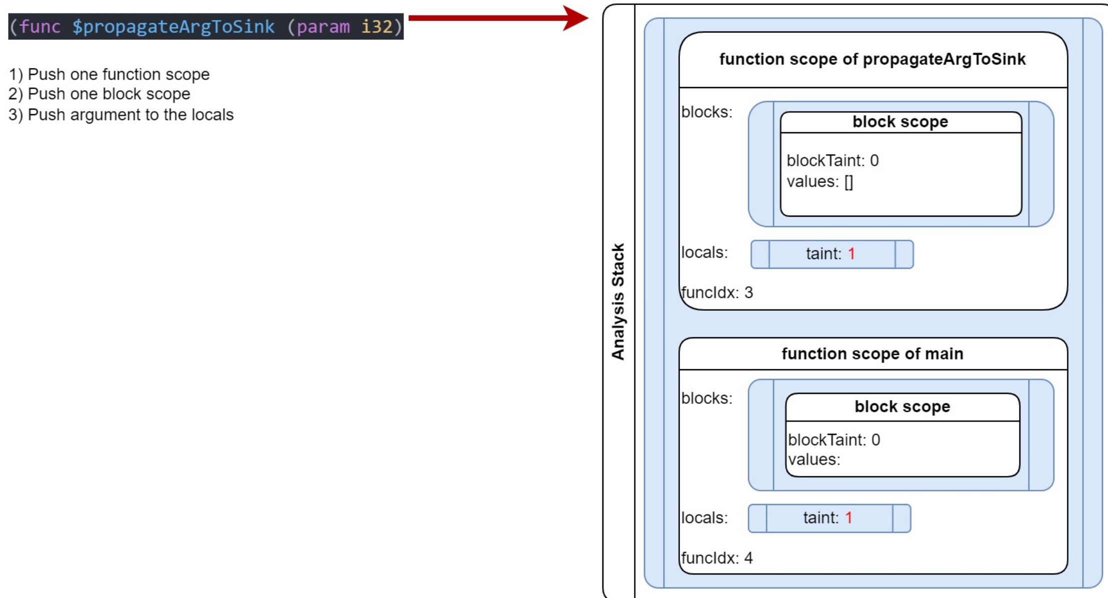

Figure 6.11: Preparing execution environment for the called function.

In this step, the instruction pushes a value from the locals to the block scope of the function, as shown in Figure 6.12.

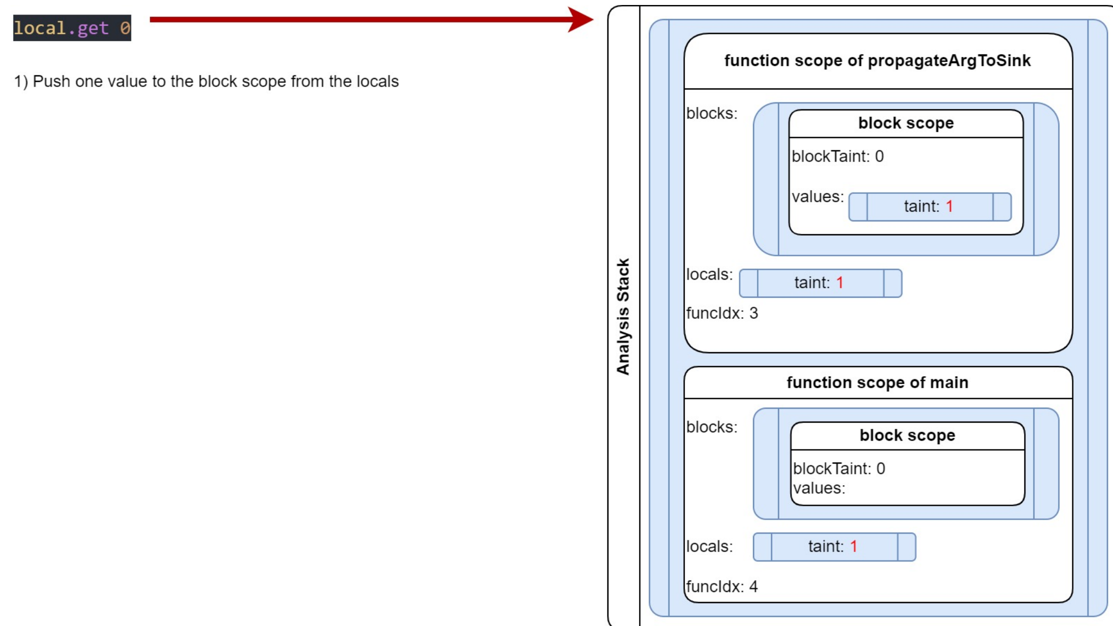

Figure 6.12: Pushing a value from locals to the block scope of the function.

At this stage, the function proceeds to call the API method *push_data*, which serves as the taint sink in this example. As a consequence of this call, one value is popped from the stack, corresponding to the method's argument, and a function scope along with a block scope is pushed. Although we do not trace the detailed execution of the API method, as it is imported from JavaScript, it is important to note that since the argument is tainted, TaintWasm will respond by triggering a warning. This process is illustrated in Figure 6.13.

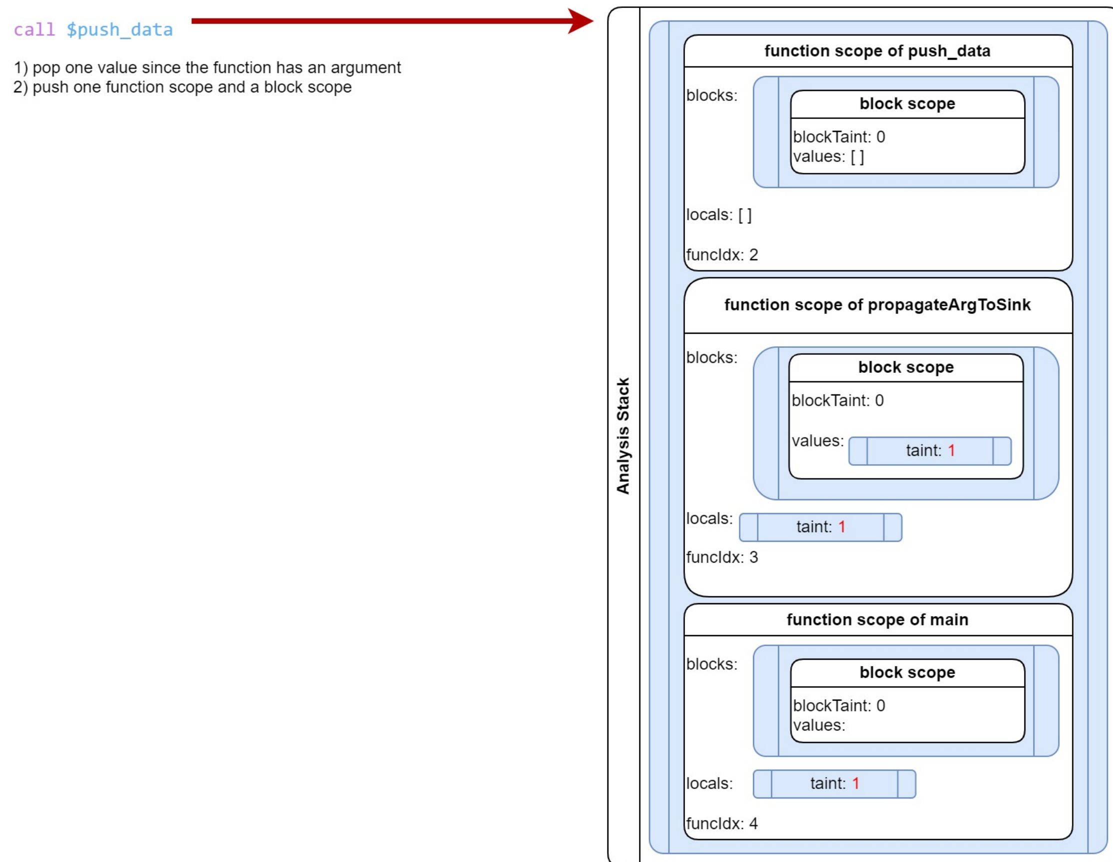

Figure 6.13: Calling the *push_data* API method, the taint sink.

With the API method call completed, the execution proceeds to pop the block scope and the function scope of the call, as depicted in Figure 6.14.

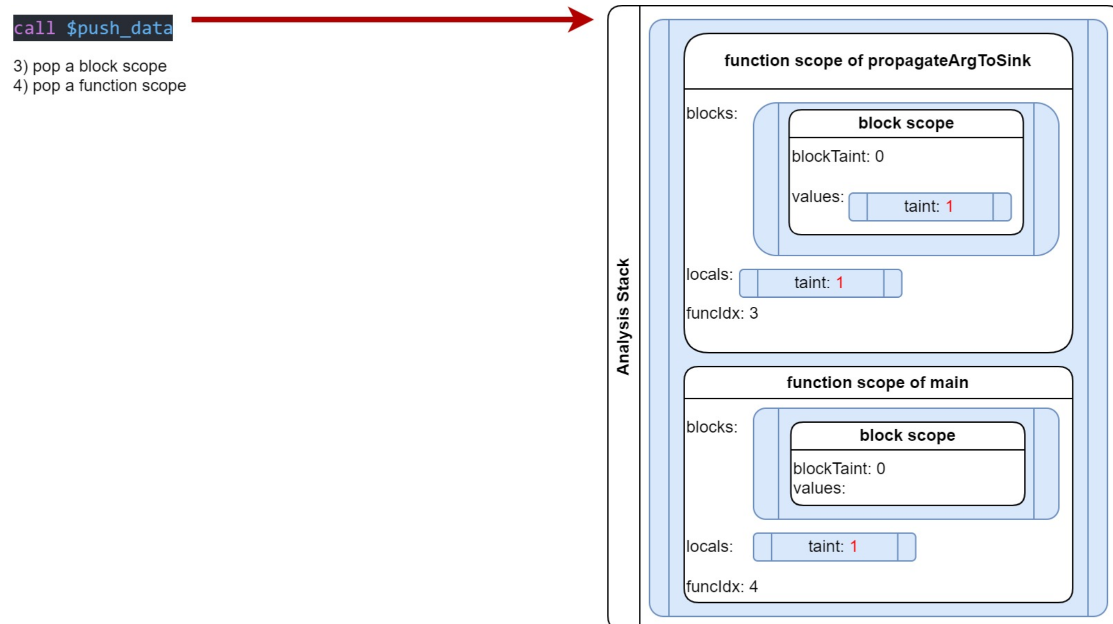

Figure 6.14: Popping the block and function scopes after the completion of the API method call.

As we conclude the function call, the execution proceeds to pop the environment of the function, including both the block scope and the function scope, as illustrated in Figure 6.15. Subsequently, the execution returns to the calling function, which in this case is the *main* function.

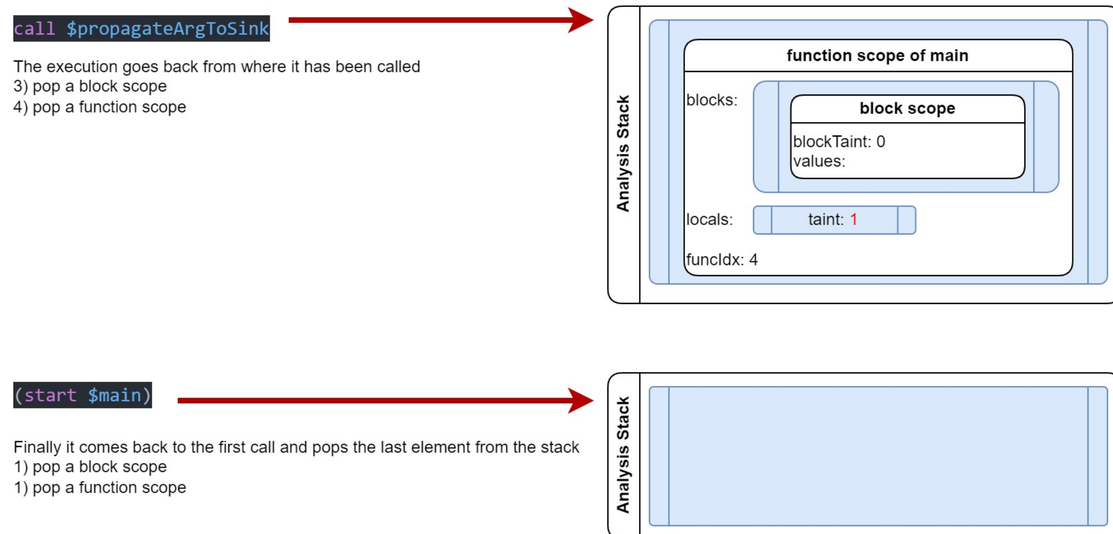

Figure 6.15: Popping the environment of the function at the end of the function call.

6.4.3 Implicit Flow and Potential Implicit Flow Handling

In this subsection, we will illustrate an example of taint analysis for implicit and potential implicit flows. The presented example involves a simple WebAssembly code snippet, followed by an image depicting the key steps of the analysis with the mirrored stack representation on the right. For this demonstration, we will assume that the condition within the code is true.

Listing 6.6: WebAssembly code example

```
1  (module
2      (import "env" "get_data" (func $get_data (;1;) (result i32)))
3      (import "env" "push_data" (func $push_data (;2;) (param i32)))
4
5      (func $main (;3;) (local $var i32)
6          i32.const 0
7          local.set $var
8          call $get_data ;; source
9          i32.const 100
10         i32.eq
11         if
12             i32.const 1
13             local.set $var
14 end
15         local.get $var
16         call $push_data ;; sink
17     )
18
19     (start $main)
20 )
```

Figure 7.1 visually outlines the taint analysis steps for the provided code snippet. Note that the illustration begins directly at the `i32.eq` instruction for brevity, excluding the preceding steps. Each step showcases the analysis stack, which encompasses function scopes, local variables, and block scopes. Notably, the figure presents the taint status of various elements within the stack.

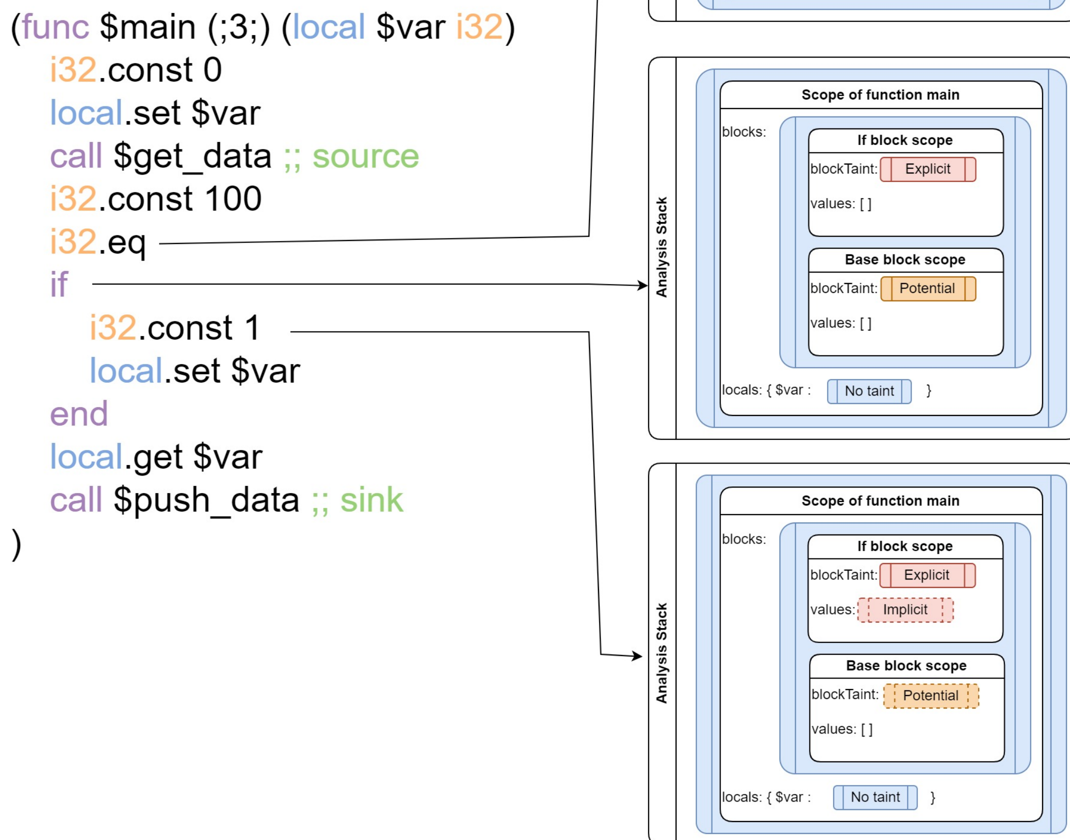

Figure 6.16: Illustration of taint analysis steps with mirrored stack

Starting at the `i32.eq` instruction, the mirrored stack reveals the taint propagation within the execution context. The local variable `$var` contains a clean taint status, as it has not yet been influenced by the value of `$get_data`. Similarly, the block scope exhibits a clean taint status.

However, within the values section, an "explicit" taint emerges. This taint results from the combination of the taint from `$get_data`'s value and the constant 100, as the taint propagates through the computation.

Subsequently, as the execution proceeds to the `if` condition, the value within the initial block scope is consumed. A new block scope is introduced within the block stack. Although these operations generally occur as separate steps, they are combined here to enhance clarity.

Within this step, the taint statuses evolve further. The block scope of the original scope gains a "potential" taint due to the tainted condition. Simultaneously, the newly added block exhibits an "explicit" taint, resulting from the union of the current block's taint and the condition's taint.

In the final step, after the `i32.const 1` instruction following the `if` block, a new taint emerges within the `if` block's values. This taint is characterized as "implicit" due to the tainted status of the block itself. It is important to note that a "potential" taint would propagate if the block had a "potential" taint, but for an "explicit" taint it would propagate an "implicit" taint in the block.

6.5 Implementation challenges

6.5.1 Managing Function Stack Popping

Within the TaintWasm framework, wasabi gives three distinct hook operations to signify the end of a function's execution: "return", "end", and "call_post". Each hook has specific implications in the context of WebAssembly's function execution.

The "return" hook occurs when a function returns a value. In TaintWasm's implementation, the "return" hook is unique—it peeks the function scope and preserves it, rather than popping it. This approach was chosen due to the order of subsequent operations: after "return," an "end" operation is encountered, followed by a "call_post" operation. While all three hooks may not always materialize, it is essential to account for each possibility.

The challenge arises in accurately maintaining the mirror stack to replicate WebAssembly's stack behaviour. Specifically, managing the function scope and ensuring its proper removal from the stack becomes crucial, given the presence of three potential hooks symbolizing the end of a function.

The "end" hook marks the conclusion of a block, if-condition, loop, or function. In this context, we focus solely on the "end" of a function. Additionally, the "call_post" hook emerges after a function completes execution, the scope exits, and control returns to the caller function.

To overcome this challenge, we aim to precisely synchronize our mirror stack with the stack of WebAssembly. The primary concern is managing the function scope's removal from the stack correctly and just once. While either the "end" or "call_post" hook will always occur after the "return" hook, the problem occurs when both "end" and "call_post" hooks are present. To tackle this issue, a logical solution is required to ensure that the function scope is popped only once, maintaining consistency in the stack's representation.

The delicate management of these hook operations is essential to achieving a faithful representation of the WebAssembly stack and ensuring the accurate execution of TaintWasm's dynamic taint analysis.

6.5.2 Challenges in Compiling C Code with Emscripten

Emscripten, a comprehensive compiler toolchain for WebAssembly, empowers the transformation of C and C++ code into formats that can be executed across various platforms, including web browsers, Node.js, and WebAssembly runtimes. By default, Emscripten generates a WebAssembly module, JavaScript code acting as a bridge between the WebAssembly module and the web environment, and an HTML page to orchestrate the execution. However, the requirements of TaintWasm necessitated a more streamlined compilation process, focused solely on producing a stand-alone WebAssembly module.

The initial objective was to extract the WebAssembly module without the accompanying HTML, a relatively straightforward task. Nevertheless, a persistent challenge emerged due to the presence of attached JavaScript code. Over several weeks of dedicated research, characterized by trial and error, a solution was ultimately unearthed, allowing the generation of a self-contained WebAssembly module. The rationale behind this endeavour was rooted in the necessity to exert precise control over the execution of the WebAssembly module and the imported functions, which was hindered by the presence of the JavaScript bridge.

Furthermore, integrating the project's JavaScript API into the C code without relying on conventional glue proved to be intricate. Negotiating this challenge required an in-depth exploration of viable strategies to seamlessly incorporate the JavaScript API within the C codebase.

A notable hurdle surfaced in the form of dead code elimination, a component that systematically removes code sections not directly linked to the main function

or those that are not directly exported. This feature inadvertently posed difficulties, as critical code segments risked removal due to their distinct execution paths.

Following the resolution of compilation challenges, the focus shifted to establishing communication between the JavaScript API and the C codebase. A significant amount of time was allocated to comprehending the intricacies of passing complex data structures, such as strings or other objects. Notably, the WebAssembly environment primarily supports integer and floating-point data types. To overcome this limitation, a breakthrough came in the form of utilizing shared memory to facilitate data exchange between the JavaScript environment and the WebAssembly module. This approach enabled the creation of pointers to shared memory regions, thereby enabling the seamless transmission of more intricate data structures.

In summary, the journey to successfully compile C code with Emscripten for the TaintWasm project was riddled with challenges. From refining the compilation process to extracting the WebAssembly module sans unwanted code, integrating the JavaScript API, and addressing data exchange intricacies, each obstacle demanded rigorous investigation and inventive solutions. The eventual triumph over these challenges underscores the innovative spirit and tenacity that underscored the development of the TaintWasm tool.

Chapter 7

Results, limitations and possible improvements

In this work we try to evaluate if pure dynamic taint analysis is a viable solution to identify and prevent sensitive data leaks inside a hybrid smart home. As a reminder our hybrid Iot architecture has a hub on which most calculations are done and we consider the cloud to be an untrusted zone, however, we keep the cloud to avoid the limitations of hub-only smart homes. To achieve this objective we identified several key points that the dynamic taint analysis system must respond to. Is it able to spot any kind of data leaks and not only explicit ones? Does the overhead induced by the analysis prevent it to be used in real time? And finally, can the system be deployed easily on an IoT system thanks to its portability? We made our benchmark and research to answer those questions. Since we had no suitable system on which to develop TaintWasm we had to create our benchmark. We try to be as exhaustive as possible on the testing and cover every edge case. We also implemented an example concrete application to test the overhead of the analysis. In this chapter, we will go through our results before discussing those through the scope of the previous questions. Then we will give possible improvements to mitigate the identified limitations.

7.1 Our Benchmark

In this section, we present the benchmark we designed to evaluate the effectiveness and performance of the TaintWasm dynamic taint analysis tool. Our benchmark suite encompasses a comprehensive range of test cases aimed at assessing the capabilities of TaintWasm in identifying explicit, implicit, and potential data leaks within WebAssembly code. To construct these tests, we carefully crafted scenarios targeting various types of leaks to challenge the dynamic taint analysis provided

by TaintWasm. The primary objective was to identify edge cases that produce under-tainting. This is to answer if yes or no dynamic taint analysis can on its own detect every data leak even if it produces overtainting.

The benchmark table 7.1 presents the outcomes of our evaluation, demonstrating the results of the taint analysis and the corresponding performance overhead ratios for each test case. In this table, we outline the categories of explicit flows, implicit flows, and potential flows, along with cases where no leaks are expected. Each test is identified by a descriptive name that characterizes its nature and purpose. The proportion of explicit and implicit flows does not represent reality since those are for most edge cases. Indeed, in a real application, we expect to find a large proportion of potential and implicit flows, while having few explicit flows. This is because potential implicit flows cover a very wide range of cases, thus the developer of the application should be very cautious to avoid those, while explicit flows represent fraudulent behaviours.

The benchmark tests were formulated in the WebAssembly text format (wat) to encompass various instructions and instruction arrangements. Our approach ensured that nearly all fundamental instructions were evaluated in isolation, leading to a comprehensive assessment. Moreover, we explored various combinations of instructions, including complex arrangements, aiming to exercise the capabilities of TaintWasm to its fullest extent. It is important to note that certain situations yielded overtainting, but these instances were not extensively included in the table to maintain readability and conciseness.

A considerable number of edge cases were unveiled during the test generation process. Specifically, we leveraged the compilation of C code using Emscripten while also deliberately tuning the optimization levels to reveal intricate instances of potential data leakage. This method provided a valuable source of real-world cases that challenged TaintWasm. These cases, when combined with our meticulously crafted tests, contribute to a comprehensive evaluation of TaintWasm's capabilities.

The benchmark results exhibit a high degree of consistency. The average overhead ratio across all test cases approximates 2.19, presenting a minimal ratio of 1.81 and reaching a maximum of 3.59. It is important to note, however, that these outcomes diverge from the observations documented by Lehmann and Pradel in their comprehensive analysis of the Wasabi framework [7]. This disparity in results serves to emphasize a key distinction, wherein our benchmark evaluation is based on a distinctive set of circumstances.

Namely, our test cases encompass relatively straightforward scenarios characterized by a limited instruction range of 10 to 50. This inherent simplicity accounts for the comparatively diminished overhead ratio, as opposed to the Wasabi study where more intricate and expansive codebases were examined. Consequently, the modest overhead ratio of 2, when applied to succinct code segments, masks the actual magnitude of the computational burden that dynamic analysis would impose on larger and more intricate codebases. The inverse relationship between code complexity and apparent overhead ratio underscores the significance of accounting for this factor when interpreting the benchmark results.

Furthermore, these observations find alignment with our C code. In these cases, we observed an overhead ranging from 10 to 40, reaffirming the notion that the complexity and scale of the code significantly influence the overhead introduced by dynamic analysis.

In summary, our benchmark suite serves as an evaluation of the TaintWasm tool's proficiency in dynamic taint analysis for WebAssembly code. By systematically creating diverse scenarios and drawing inspiration from real-world code compilation, we have provided a comprehensive assessment that highlights both strengths and potential areas for improvement. The alignment of our findings with established research further bolsters the reliability of our evaluation.

Type of flows	Test Name	Overhead Ratio	Explicit	Implicit	Potential
Explicit flows	arg_to_sink_api_leak_explicit	2.02	✓	✓	✓
	break_taint_leak_explicit	2.24	✓		
	drop_leak_explicit	2.13	✓		
	empty_leak_explicit	2.09	✓		
	fct_call_api_leak_explicit	2.42	✓		
	fct_call_with_multi_arg_api_leak_explicit	2.10	✓		
	fct_call_with_return_api_leak_explicit	2.37	✓		
	global_copy_api_leak_explicit	2.09	✓		
	local_copy_local_api_leak_explicit	2.47	✓		
	memory_copy2_api_leak_explicit	2.61	✓		
	memory_copy_api_leak_explicit	2.22	✓		
	op_binary_api_leak_explicit	2.08	✓		
	op_unary_api_leak_explicit	2.27	✓		
	select_add_leak_explicit	1.93	✓		
	select_leak_explicit	2.12	✓		
Implicit flows	fct_ret_if_api_leak_implicit	1.93		✓	✓
	fct_ret_if_else_api_leak_implicit	2.33		✓	✓
	if_else_with_block_leak_implicit	2.31			✓
	if_if_api_leak_implicit	2.10		✓	✓
	op_binary_api_leak_implicit	2.23		✓	✓
	op_unary_api_leak_implicit	1.87		✓	✓
	select_leak_implicit	1.93		✓	✓
Potential flows	loop_api_leak_potential_implicit	1.93			✓
	block_br_api_leak_potential_implicit	2.08			✓
No leaks	arg_to_sink_api_clean	2.08			
	block_block_br_api_clean	2.13			
	block_pop_clean	2.02			
	drop_clean	1.94			
	fct_ret_if_api_clean	2.31			
	global_copy_api_clean	2.26			
	if_api_clean	2.13			
	local_no_copy_local_api_clean	2.51			
	memory_copy_api_clean	2.03			
	op_binary_api_clean	2.12			
	op_unary_api_clean	3.59			
	select_clean	2.21			
	unreachable_clean	2.18			

Table 7.1: Taint analysis results and performance overhead ratios for benchmark tests

Furthermore, we developed a concrete application in C to complement our evaluation. This application serves as a practical illustration of how TaintWasm operates within a real-world scenario. The application functions by monitoring sensor values through an API, which simulates sensor data by generating random values. Subsequently, the application processes these values and directs them to various destinations based on their magnitudes.

In the application design, each sensor value retrieved through the API is sent to the "temperature_log" endpoint. This allows us to assess TaintWasm's behaviour in scenarios where data flows to a log or storage facility. Moreover, the application generates personalized messages corresponding to the sensor values and forwards

these messages to the "temperature_alert" endpoint. This dynamic message customization showcases TaintWasm's capability to handle diverse data types, such as strings. The ability to process and transmit string objects exemplifies the tool's versatility in detecting and managing data leaks across various data structures.

As we observed in our experimentation, the instrumented version of this application exhibits an overhead of approximately tenfold in comparison to the non-instrumented variant. The substantial increase in overhead underscores the resource requirements of dynamic taint analysis, an aspect that needs careful consideration in real-time or resource-constrained environments. This reinforces the trade-off between enhanced security through taint analysis and the associated performance cost, particularly in scenarios where real-time responsiveness is crucial.

7.2 Discussion

In this section we will discuss the three points we specified earlier in accordance with the results we presented. The first one was to evaluate if TaintWasm can identify every sensitive data leak and not only explicit ones. As we saw we implemented TaintWasm to have three levels of detection. Thus it can detect explicit flows, implicit flows and potential implicit flows during a single analysis. However, as we saw in Chapter 3, dynamic taint analysis systems can not identify implicit flows without over-tainting. TaintWasm does not derogate the rule, that is why we implement these three levels of analysis, by doing so the system using TaintWasm knows every possible data leak and can take action according to the sensitivity of the analysis. Indeed, if we take for example an application that detects the presence of a person in rooms using camera feeds. The camera feeds are considered to be very sensitive data and the system wants to be sure that only notifications and absolutely no data even derived from the camera feed are forwarded to the cloud. Then it can decide to discard every flow detected at the taint sink even potential implicit ones. On the other hand, if the application only uses temperature data and it considers that those temperatures are not that sensitive but still wants to avoid raw streams of data being pushed to the cloud the application can discard only explicit flows. In brief, TaintWasm as a pure dynamic taint analysis tool cannot avoid false positives, however, we implement three levels of granularity to let the IoT system decide how to handle those. Furthermore, to our knowledge and in accordance with our benchmark TaintWasm does not produce false negatives (under-tainting).

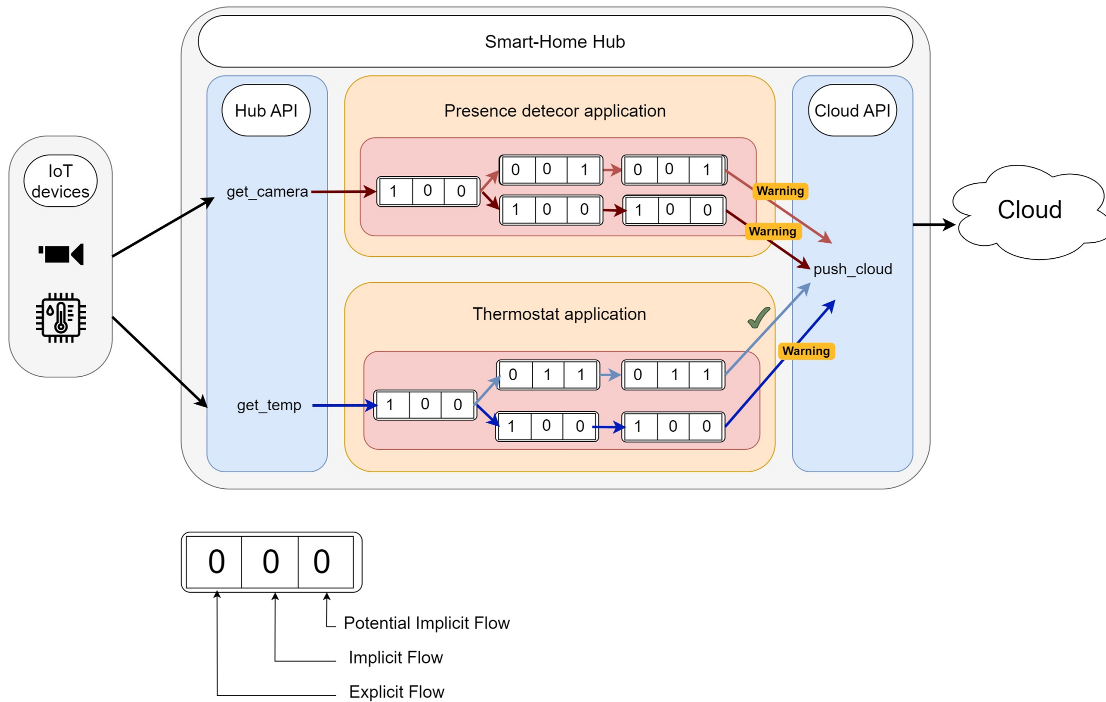

Figure 7.1: Illustration of system choice toward sensitivity level of the analysis

In the table 7.1, we observed that the overhead induced by TaintWasm ranged from 2 to 3.5 in comparison to the non-instrumented test. This is lower than the results announced in the Wasabi paper [7], where the overhead ranged from 49 to 163. The smaller size of our codes is the reason for this difference, but higher overhead has been observed on compiled c code. For small applications, we found that the analysis can still be used in real-time, however, for larger ones this might make them unusable. A straightforward solution to this overhead would be to implement the analysis by modifying the interpreter or another solution than code instrumentation. However, doing so will make the analysis not portable anymore. Even though the analysis is difficult to be performed in real-time, the tool can be used for developers or inside a sandbox before the deployment of a third-party application. However such use of TaintWasm needs some modification to its layout.

Regarding the final aspect of our discussion, we consider the portability of the analysis. Given the inherent portability of WebAssembly, the dynamic taint analysis enabled by TaintWasm extends beyond its traditional web context and can be executed on various platforms, including Node.js and other WebAssembly/JavaScript runtimes. Nevertheless, it is important to acknowledge that while WebAssembly

offers portability, the original Wasabi framework was primarily designed for web-related contexts. Although subsequent modifications were introduced to Wasabi to incorporate support for Node.js, this extension has not undergone extensive testing, as documented in an issue report [51]. The fundamental concepts underlying TaintWasm, however, possess the potential for replication within different frameworks and contexts. Despite the limited availability of alternative frameworks at present, TaintWasm was developed with the Wasabi framework due to its alignment with the existing ecosystem.

7.3 Possible improvements

As we saw in the previous section TaintWasm does not meet every requirement it needs to handle data leak identification and prevention in an Iot system on its own. Although it has room for improvement and it is still a solid foundation.

The first big improvement to add to TaintWasm would be a complementary analysis to differentiate malicious and innocent implicit flows. The approach of dta++ [40] looks promising for our case, using symbolic execution to identify implicit flows that are functionally dependent on tainted data would decrease the number of overtainted values. A similar approach could be developed to handle potential implicit flows but this might be more challenging. Some static analysis techniques could do this job too, but they might require the source code or reverse engineering to be functional.

To reduce the overhead some optimization might be developed. A first idea that might be advanced would be using fewer hooks but that seems to be very challenging since this might cause possible undertainting which we want to avoid, indeed, using fewer hooks will make the mirrored stack not sound with the stack of WebAssembly. The overhead is mainly because TaintWasm is implemented over code instrumentation, but this is necessary for its portability. This makes the overhead very difficult to reduce. Areas of improvement are thus found in how to execute the TaintWasm analysis not in real-time while continuing to prevent data leakage. This might be responded to by the system that would deploy TaintWasm. Using it only in some executions or inside a sandbox might be an area to develop.

7.4 Conclusion

In this master's thesis, we addressed the challenge of enhancing privacy within smart home environments. As the proliferation of IoT systems continues, the risk of data leakage and privacy breaches becomes more noticeable. Our focus to give vulnerable users more control over sensitive data led us to explore dynamic taint analysis (DTA) as a potential solution to identify and prevent sensitive data leaks.

We examined the landscape of IoT systems and their security concerns, particularly in the context of smart homes. We aimed to test if dynamic taint analysis by itself is sufficient to identify data leaks in a hybrid smart home environment. We identify that hybrid smart homes are a good architecture to compromise between privacy and keeping the advantages of the cloud.

We then explored dynamic and static analysis to lay the foundation for our approach. We mainly focused on dynamic taint analysis since we try to make a proof of concept around it. We introduced TaintWasm, our implementation of dynamic taint analysis for WebAssembly binaries, built upon the Wasabi framework. We choose WebAssembly since it is emerging as a great solution to have good performance, security and portability further than the web environment. And thus revealed itself as a promising technology for IoT.

Through testing and research about dynamic taint analysis, we evaluated the capabilities and limitations of TaintWasm. While the results revealed the potential of dynamic taint analysis to identify various types of data leaks and enhance privacy in smart homes, challenges emerged. Notably, the overhead introduced by the analysis causes a significant complexity for the real-time deployment of the analysis on applications.

In conclusion, our research contributes to the ongoing discourse on privacy within IoT environments, specifically in the context of smart homes. Dynamic taint analysis demonstrates its relevance in identifying data leaks and enhancing responsibility for sensitive data usage. However, to realize its full potential, it must be complemented with other techniques that address inherent limitations.

Bibliography

[1] Marie Chan et al. "A review of smart homes—Present state and future challenges". In: *Computer Methods and Programs in Biomedicine* 91.1 (2008), pp. 55–81. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2008.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0169260708000436.

[2] Luca Greco et al. "Trends in IoT based solutions for health care: Moving AI to the edge". In: *Pattern Recognition Letters* 135 (2020), pp. 346–353. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2020.05.016. URL: https://www.sciencedirect.com/science/article/pii/S0167865520301884.

[3] Phani Kishore Gadepalli et al. "Challenges and Opportunities for Efficient Serverless Computing at the Edge". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 2019, pp. 261–2615. DOI: 10.1109/SRDS47363.2019.00036.

[4] Elliott Wen and Gerald Weber. "Wasmachine: Bring IoT up to Speed with A WebAssembly OS". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2020, pp. 1–4. DOI: 10.1109/PerComWorkshops48775.2020.9156135.

[5] Pankaj Mendki. "Evaluating WebAssembly Enabled Serverless Approach for Edge Computing". In: *2020 IEEE Cloud Summit*. 2020, pp. 161–166. DOI: 10.1109/IEEECloudSummit48914.2020.00031.

[6] William Enck et al. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *ACM Trans. Comput. Syst.* 32.2 (June 2014). ISSN: 0734-2071. DOI: 10.1145/2619091. URL: https://doi.org/10.1145/2619091.

[7] Daniel Lehmann and Michael Pradel. "Wasabi: A framework for dynamically analyzing WebAssembly". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 1045–1058.

[8] Mohammad Hasan. *State of IOT 2022: Number of connected IOT devices growing 18 to 14.4 billion globally*. June 2022. URL: https://iot-analytics.com/number-connected-iot-devices/.

[9] Igor Zavalyshyn et al. "SoK: Privacy-enhancing Smart Home Hubs". In: *Proceedings on Privacy Enhancing Technologies* 4 (2022), pp. 24–43.

[10] Amazon. *Amazon Alexa*. URL: https://alexa.amazon.com/.

[11] Samsung. *Samsung SmartThings*. 2023. URL: https://www.samsung.com/be_fr/apps/smartthings/.

[12] Somfy. *Somfy - solutions domotiques pour une maison connectée somfy*. URL: https://www.somfy.be/fr-be/.

[13] *Empowering the smart home*. URL: https://www.openhab.org/.

[14] Derick Schweppe. *The Open Source Privacy-focused voice assistant*. URL: https://mycroft.ai/.

[15] Jasmin Guth et al. "A detailed analysis of IoT platform architectures: concepts, similarities, and differences". In: *Internet of everything: algorithms, methodologies, technologies and perspectives* (2018), pp. 81–101.

[16] Serena Zheng et al. "User Perceptions of Smart Home IoT Privacy". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018). DOI: 10.1145/3274469. URL: https://doi.org/10.1145/3274469.

[17] Lakshmisri Surya. "Security challenges and strategies for the IoT in cloud computing". In: *International Journal of Innovations in Engineering Research and Technology ISSN* (2016), pp. 2394–3696.

[18] Serena Zheng et al. "User Perceptions of Smart Home IoT Privacy". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018). DOI: 10.1145/3274469. URL: https://doi.org/10.1145/3274469.

[19] Lo'ai Tawalbeh et al. "IoT Privacy and Security: Challenges and Solutions". In: *Applied Sciences* 10.12 (2020). ISSN: 2076-3417. DOI: 10.3390/app10124102. URL: https://www.mdpi.com/2076-3417/10/12/4102.

[20] George Demiris et al. "Senior residents' perceived need of and preferences for "smart home" sensor technologies". In: *International Journal of Technology Assessment in Health Care* 24.1 (2008), pp. 120–124. DOI: 10.1017/S0266462307080154.

[21] Karen L. Courtney et al. "Needing smart home technologies: the perspectives of older adults in continuing care retirement communities." In: *Informatics in primary care* 16 3 (2008), pp. 195–201.

[22] Vitor A. Cunha et al. “A Network Service for Preventing Data Leakage from IoT Cloud-assisted Equipment”. In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. 2019, pp. 1–7. DOI: 10.1109/ISCC47284.2019.8969719.

[23] *Mathworks, Polyspace*. URL: https://nl.mathworks.com/learn/training/polyspace-for-c-cpp-code-verification.html.

[24] *COVERITY SCAN STATIC ANALYSIS*. URL: https://scan.coverity.com/.

[25] *Klocwork 2023*. URL: https://help.klocwork.com/.

[26] Pär Emanuelsson and Ulf Nilsson. “A Comparative Study of Industrial Static Analysis Tools”. In: *Electronic Notes in Theoretical Computer Science* 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Veri[U+FB01]cation (SSV 2008), pp. 5–21. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. URL: https://www.sciencedirect.com/science/article/pii/S1571066108003824.

[27] *Splint static analysis for C*. URL: http://splint.org/.

[28] David Evans and David Larochelle. “Improving security using extensible lightweight static analysis”. In: *IEEE software* 19.1 (2002), pp. 42–51.

[29] Jens Palsberg and Michael I. Schwartzbach. “Object-Oriented Type Inference”. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’91. Phoenix, Arizona, USA: Association for Computing Machinery, 1991, pp. 146–161. ISBN: 0201554178. DOI: 10.1145/117954.117965. URL: https://doi.org/10.1145/117954.117965.

[30] Tim A Wagner et al. “Accurate static estimators for program optimization”. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 1994, pp. 85–96.

[31] Andreas Moser, Christopher Kruegel, and Engin Kirda. “Limits of Static Analysis for Malware Detection”. In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 2007, pp. 421–430. DOI: 10.1109/ACSAC.2007.21.

[32] Xiaolu Zhang et al. “Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations”. In: *Forensic Science International: Digital Investigation* 39 (2021), p. 301285. ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2021.301285. URL: https://www.sciencedirect.com/science/article/pii/S2666281721002031.

[33] *Valgrind*. URL: https://valgrind.org/.

[34] Bas Cornelissen et al. “A Systematic Survey of Program Comprehension through Dynamic Analysis”. In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702. DOI: 10.1109/TSE.2009.28.

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)”. In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[36] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. “The concept assignment problem in program understanding”. In: *[1993] Proceedings Working Conference on Reverse Engineering*. 1993, pp. 27–43. DOI: 10.1109/WCRE.1993.287781.

[37] Dapeng Liu et al. “Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 234–243. ISBN: 9781595938824. DOI: 10.1145/1321631.1321667. URL: https://doi.org/10.1145/1321631.1321667.

[38] James Newsome and Dawn Xiaodong Song. “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software.” In: *NDSS*. Vol. 5. Citeseer. 2005, pp. 3–4.

[39] Bas Cornelissen et al. “A systematic survey of program comprehension through dynamic analysis”. In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702.

[40] Min Gyung Kang et al. “Dta++: dynamic taint analysis with targeted control-flow propagation.” In: *NDSS*. 2011.

[41] James Clause, Wanchun Li, and Alessandro Orso. “Dytan: a generic dynamic taint analysis framework”. In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 196–206.

[42] Rezwana Karim et al. “Platform-independent dynamic taint analysis for javascript”. In: *IEEE Transactions on Software Engineering* 46.12 (2018), pp. 1364–1379.

[43] Ali Davanian et al. “{DECAF++}: Elastic {Whole-System} dynamic taint analysis”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019, pp. 31–45.

[44] Michael D Ernst. “Static and dynamic analysis: Synergy and duality”. In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. 2003, pp. 24–27.

[45] Shiyi Wei and Barbara G Ryder. “Practical blended taint analysis for JavaScript”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, pp. 336–346.

[46] Andreas Haas et al. “Bringing the web up to speed with WebAssembly”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

[47] *WebAssembly official website*. URL: `https://webassembly.org/`.

[48] Niko Mäkitalo et al. “WebAssembly modules as lightweight containers for liquid IoT applications”. In: *International Conference on Web Engineering*. Springer. 2021, pp. 328–336.

[49] *First pull request*. URL: `https://github.com/danleh/wasabi/pull/38`.

[50] *Second pull request*. URL: `https://github.com/danleh/wasabi/pull/39`.

[51] *Wasabi nodejs issue*. URL: `https://github.com/danleh/wasabi/issues/31`.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**