Detecting access to sensitive data in software extensions through static analysis

Att upptäcka åtkomst till känslig information i mjukvarutillägg genom statisk analys

Johan Hedlin
Joakim Kahlström

Supervisor : George Osipov
Examiner : Ola Leifler
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmanens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Johan Hedlin
Joakim Kahlström
Abstract

Static analysis is a technique to automatically audit code without having to execute or manually read through it. It is highly effective and can scan large amounts of code or text very quickly. This thesis uses static analysis to find potential threats within a software’s extension modules. These extensions are developed by third parties and should not be allowed to access information belonging to other extensions. However, due to the structure of the software there is no easy way to restrict this and still keep the software’s functionality intact. The use of a static analysis tool could detect such threats by analyzing the code of an extension before it is published online, and therefore keep all current functionality intact. As the software is based on a lesser known language and there is a specific threat by way of information disclosure, a new static analysis tool has to be developed. To achieve this, a combination of language specific functionality and features available in C++ are combined to create an extendable tool which has the capability to detect cross-extension data access.
# Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of code listings</td>
<td>viii</td>
</tr>
</tbody>
</table>

1 Introduction 1
   1.1 Motivation 1
   1.2 Aim 2
   1.3 Research questions 2
   1.4 Delimitations 2

2 Theory 3
   2.1 Blacklist and whitelist 3
   2.2 Static and dynamic analysis 3
   2.3 Existing tools for static analysis 4
   2.4 Regular expressions 4
   2.5 Flex and Bison 5
   2.6 Databases of malicious domains 6
   2.7 Extensions in other platforms
      2.7.1 Chromium 6
      2.7.2 Firefox 7
      2.7.3 Android 7
   2.8 Security risks with extensions
      2.8.1 Reading/writing files 8
      2.8.2 Keylogger 9
      2.8.3 Poorly written extensions 9
   2.9 Evaluation
      2.9.1 Evaluating prototype methods 10
      2.9.2 Clustering of warnings 10

3 Method 12
   3.1 Static analysis 12
   3.2 Prototype
      3.2.1 C++ string matching 12
      3.2.2 Parsing using Flex and Bison 13
      3.2.3 Built-in language features 14
      3.2.4 Combining multiple approaches 15
      3.2.5 Extending modules 15
## List of figures

2.1 How a float is represented with regular expressions .................. 5
2.2 Two entries of malicious addresses from the URLhaus database ....... 6
3.1 Directory structure of extensions .......................................... 15
4.1 Default string matching output ............................................. 18
4.2 Colorized string matching output ......................................... 18
4.3 Bison prototype output ..................................................... 19
4.4 Bison prototype run on comment and variable declaration .......... 20
4.5 Example of report generated by the analyzer using language features 21
4.6 Example of report generated by the C++ string matcher as part of the combined analyzer .................................................. 22
List of tables

4.1 Bison prototype parser statistics ..................................................... 19
4.2 Evaluation results on externally developed extensions ......................... 23
4.3 Evaluation results on internally developed extensions .......................... 23
List of code listings

2.1 Example Flex scanner specification ........................................... 5
2.2 Example Bison parser specification ......................................... 6
3.1 Example detections with string matching .............................. 14
3.2 Example detections with parsing ........................................... 14
3.3 Example source code ............................................................ 14
3.4 Example IR ................................................................. 14
4.1 Ambiguous if-statement ......................................................... 18
4.2 Format of the path whitelist .............................................. 20
4.3 Format of the name blacklist ............................................. 20
1 Introduction

1.1 Motivation

The company where the work for this thesis was done develops a software with support for third-party extensions. The extensions are distributed by the company themselves and are published on their website. Since the company distributes the extensions, they become responsible for making sure that the extensions do not harm customers’ systems in any way, since any issue could harm the company’s reputation. This is similar to how the market of apps for smartphones works, where the distributor is responsible to remove potentially harmful apps. A similar problem could arise if two competing extensions were able to spy on each other when they are both on the same system. Sensitive data could then be collected and sent back to the developer in an automatic way, which is something that should be prevented.

However, their software does not make use of multithreading and extensions are not separated into different processes. This will limit the possibility to use the security functions of the operating system, as most of these only allow for restrictions to be applied to an entire process. This can be used to limit other programs’ access to the software’s memory or to set limits on resource usage, but cannot be applied to extensions that are loaded into the same process as the main program without affecting the main program itself.

An alternative to preventing malicious extensions from reading sensitive data is to instead attempt to detect malicious activity before the extension is published. With access to the source code of the extension this can be done through static analysis, where the source code is analyzed programmatically before it is compiled.

Dynamic analysis, where the extensions are run in a controlled environment where all performed actions are logged is another option for detecting malicious extensions, but the setup of this environment may be complicated and not representative of the actions an extension performs when running on customer systems. Malicious extension may also detect this environment and adapt their behavior to avoid detection.

Static analysis also has some disadvantages such as not being able to handle obfuscated code or extension that retrieves new instructions or code from the internet, but usually requires less time and system resources than dynamic analysis.

While the analysis of the source code can be performed manually by a trained analyst, this can take a lot of time for larger extensions and comes with the risk of human error causing
issues to remain unnoticed. While manual analysis could be more complete if performed by an experienced analyst, an automated approach to code analysis is a requirement if extensions are to be reviewed within a reasonable amount of time by a limited number of analysts.

If the software would have been developed in a more well-known language such as C++ or Java, there would already exist several different tools for static analysis. In that case a study of which one is the most suitable would be a good approach. But as the company makes use of a lesser known language that currently does not have any existing static analysis tool, a new one has to be developed. This can however be useful as it can be developed to find more specific threats, but might result in lower stability and coverage than that of an existing solution.

1.2 Aim

The purpose of this thesis is to utilize static analysis to detect when an extension accesses sensitive data from another extension. The main form of sensitive data that the company is currently concerned with is price information for different products. For example, extensions from two competing companies should not be able to access price information from each other.

1.3 Research questions

1. How can access to sensitive data be detected in source code using static analysis?

2. How can the performance of a static analysis method be measured?

1.4 Delimitations

Since the software only supports Microsoft Windows the analysis implementation presented in this work will only support Windows 7 and newer, although running it on other platforms will most likely work as well.

Due to the difficulty and complex nature of detecting many security-related issues, the work will be limited to a very specific problem. The work will focus on the threat where extensions could spy on each other and read sensitive data programmatically. Therefore, this thesis will not analyze threats where manual input from the user is required. Other malware-like behavior such as collecting data about the user or the system or running untrusted code is not the main focus of the work, however if checks for some common malware patterns prove trivial to add they might be included in the analyzer.

Because of the very limited time and the complexity of their software, the work will focus on a prototype that can later be extended with additional functionality. The prototype will not attempt to detect all kinds of security issues, only a set of those that can lead to sensitive data being exposed. These could be calls to functions that could reveal data from other extensions, extensions that attempt to directly access data or functions from other extensions, or the use of Windows functions that could be used to read memory directly.
2 Theory

2.1 Blacklist and whitelist

The terms *blacklist* and *whitelist* are both common when talking about security. Blacklisting a word or an address for example means that it is not allowed to occur at all, and that a request or text where it occurs may be blocked or removed. Depending on the implementation where the blacklist is used this could look different, but in general the presence of objects on the list should not be allowed. Objects not on the list are not impeded in any way. Whitelisting, in contrast to blacklisting, means that for example a word or address is allowed through for further processing or exempt from some other form of check. Objects not present on the list would generally be blocked or have to be processed further. This is for example beneficial if there exist some trusted domains (maybe the company’s own domain), words that should always be allowed through a filter or a specific set of items that are allowed. However, this should be used with care as whitelisting too many things may expose a larger attack surface for other vulnerabilities.

Which one that is best to use depends on the use case, sometimes the blacklist is more effective and sometimes the whitelist is more effective. In some cases, a combination of the two is the best choice.

2.2 Static and dynamic analysis

Static analysis is a technique where a given file, often source code or binary files, is scanned through without executing it [1]. The opposite of static analysis would be dynamic analysis, where programs are analyzed during runtime. This can make the process of analyzing encrypted or obfuscated data easier, since it will at some point have to be decrypted and available in memory. However, performing the analysis statically without executing the program has the benefit that potential malicious code never gets an opportunity to execute. It is also not necessary to have a compiler/interpreter with the correct libraries installed on the system, which makes the analysis tool more portable across different software environments.
Only performing dynamic analysis may lead to an incomplete picture of the code, as the program under analysis may detect analysis environments such as virtual machines and alter its behavior to avoid detection. Emulating an entire machine and processor can make it harder for the analyzed program to detect this, but measuring the execution speed in relation to the reported clock frequency could enable the malware to deduce that it is running under an emulator. Performance will also be lower on an emulated system, which may make it more difficult to test larger programs [2].

In contrast to manually auditing the code, which is very time consuming, an automatic tool could assist developers or code reviewers in finding common errors or problems. This can have multiple uses in several different fields, such as finding spelling errors, correcting grammar, finding bugs, finding unused code, malware patterns and malicious code. One of the most common example of static analysis is to find coding errors and vulnerabilities such as buffer overflow [3, 4], where a function could read or write to adjacent memory, which has been a common problem with C/C++ for a long time as it has the ability to control memory in a fine-grained way [5].

Static analysis could also be used for optimization, as it can quickly scan through the code base and create an accurate model of the program’s structure such as memory usage and loops that could be optimized. This information could then be used for creating documentation about the program in an automatic way. One example could be to automatically provide written documentation of a class’ functions and their usage by both analyzing the code and parsing comments.

As mentioned in the delimitations (see section 1.4), the analysis performed in this work will only target a very small set of malicious activities where an extension attempts to access data from another extension.

2.3 Existing tools for static analysis

There exists a large number of static analysis tools for different purposes and languages [6], both open source and closed source. Some of these are focused on finding security vulnerabilities from the programmer’s perspective, trying to identify unsafe usage of functions [7] [8] [9] [3] or improper handling of user-supplied data by analyzing the data flow through the program [10] [11].

Some tools look through compiled binaries for information such as imported functions, cryptographic operations or obfuscated code in order to try to detect suspicious behavior that could indicate that the binary contains malware [12].

2.4 Regular expressions

Using regular expressions, or regexes, is a common way to parse strings or a sequence of characters with specified patterns. It has its origin in mathematics, neuroscience [13] and computer science, with a history reaching back to around the 1950’s. Today it is used widely across the computer science field and many programming languages have support for regular expressions either as a library or built into the language itself [14]. As it is a technique that has evolved over a long period of time with no explicit standard to follow, multiple variations and extensions of the regular expression language has been developed [15]. These build on the same techniques and foundations but might not be compatible with each other.

A regular expression pattern is used to describe in a concise way what combination of characters to find in a sentence. An example of how a float is represented is shown in the figure 2.1. This pattern is also used in section 2.5.
2.5 Flex and Bison

Flex and Bison are a set of open-source tools that are commonly used together to parse different kinds of text-based input.

Flex is a scanner generator which takes in a scanner specification and emits an implementation as source code in C. A scanner (also called lexer) is used to convert raw character strings from a file into a sequence of tokens in a process called lexical analysis. A token represents a piece of data in a specific format, such as an integer, floating-point number, email address or keyword in a programming language. Tokens can be specified using regular expressions, which allows for complex data formats to be represented as a single data unit. Listing 2.1 contains an example of a scanner specification which converts integers and floats into tokens called INTEGER and FLOAT.

```
[0-9]+ { return token::INTEGER; } // 1, 8571, 125 ...
[0-9]+\.[0-9]+ { return token::FLOAT; } // 0.1, 155.167, 1.11 ...
```

Listing 2.1: Example Flex scanner specification

Bison is a parser generator which takes a parser specification as input and produces C++ source code implementing the specification. A parser uses a set of semantic rules (called a grammar) to process the sequence of tokens that a scanner produces when run on a text file. These rules define the semantic structure which the parser tries to construct out of the given token sequence. Every rule can also have an action associated with it, which consists of C++ code that will be executed whenever input that matches the rule is detected. An error message containing the name of the file, line number and expected input can be printed when the input does not match any rule.

Bison can for example be used to construct a simple calculator using rules stating that an expression consists of either a number by itself, or two numbers separated by an operator. Actions can then be used to perform the corresponding calculation and print the result to the output terminal. Rules can be written recursively, which for a calculator allows the input 1 + 2 + 3 to be broken into parts and parsed as (1 + 2) + 3. A partial example of a parser rule without actions is displayed in listing 2.2.

```
[0-9]+ \+ \[0-9]+ \{ return token::INTEGER; \} // 1, 8571, 125 ...
```

Listing 2.2: Partial example parser rule without actions

Bison can also be used to parse source code and other structured languages. The GNU Compiler Collection (GCC) used a parser based on yacc, an older parser generator which Bison is compatible with, until it was replaced in 2004 [16]. The PostgreSQL database management system uses Flex and Bison combined to parse statements [17].
2.6 Databases of malicious domains

There exists a few databases with known, blacklisted malicious sites and domain names, such as the URLhaus list provided by abuse.ch [18]. These are regularly updated with new threats and are a good source of known-bad domains that should not be present in legitimate programs. The database information also shows if the listed sites have been taken down or if they still are online.

The database providers commonly offer a web Application Programming Interface (API) to access the relevant information about each Uniform Resource Locator (URL) in a machine-readable format. In this case, a simple text file with updated dangerous addresses can be retrieved and incorporated into the analysis tool to prove the concept. A warning can then be emitted if the analyzed code contains a domain that is on the blacklist. While it is not expected that any extension will contain openly malicious domains such as these, they should be easy enough to detect without any large modifications to the analyzer. Even if this is not the main topic of this thesis, it is a good example of how the tool could be extended with additional functionality.

Figure 2.2 shows an extraction of two entries from the URLhaus database with the available data: the added date, domain name, online status, tags about the content, and source of the report.

<table>
<thead>
<tr>
<th>Date added (UTC)</th>
<th>Malware URL</th>
<th>Status</th>
<th>Tags</th>
<th>Reporter</th>
</tr>
</thead>
</table>

Figure 2.2: Two entries of malicious addresses from the URLhaus database

2.7 Extensions in other platforms

The use of extensions can be seen in several other platforms with different architectures and security measures. The following sections discuss some of them in more depth.

2.7.1 Chromium

Chromium, the open-source base project of the Google Chrome browser, has a large repository of extensions called the Chrome Web Store. Initially extensions could be installed by any website after getting confirmation from the user, but due to security issues with websites managing to bypass this prompt the default behavior was changed in 2014 to only allow extensions installed from the Chrome Web Store [19]. Google states that submitted extensions should
appear to users immediately, but that a manual review may be required if their automated systems detect probable malicious code or violation of developer policies [20].

Older extensions using native code relied on the Netscape Plugin API (NPAPI), which made the extension run with the same permissions as the user running the browser. Newer extensions based on the Pepper Plugin API (PPAPI) are run in a sandbox, an isolated environment with limited access to the rest of the system [21]. As of 2014 Google has removed all extensions based on the older NPAPI from the store [22].

Nav Jagpal et al. [23] provide an insight into the process of detecting malicious extensions at Google and the effectiveness of it. Some of the constraints present in their review system WebEval are to minimize the number of false-positives, minimize the amount of manual work required to flag an extension as malicious, ensure that a decision on a submitted extension is made within one hour, and to ensure that the system can be extended with additional rules for detecting new forms of malicious code.

Static analysis is also a part of the WebEval system. The permissions that an extension requests are scanned for unusual entries that allow control of sensitive browser functions. The extension code is scanned for evidence of obfuscation by detecting strings and code patterns common in obfuscated code, such as unusually long encoded strings and function calls. The directory structure and file contents are stored and used to detect duplicate extensions which share 80% or more of their code with another extension. Additionally, all files are scanned for known threats using multiple anti-virus engines [23].

WebEval utilizes dynamic analysis as well to log and monitor extensions as they are run inside of a simulated browser session. The simulated session consists of recordings of real browser sessions where a person visited several sites such as Google, Facebook, YouTube and Amazon while interacting with the sites, logging into accounts and placing orders [23].

Information about the extension’s developer is used in conjunction with the previously mentioned data. This information includes things such as email domain, location of logins, and ratings for the extension. Extensions that are flagged by the automated system are then sent to a human reviewer who makes the final decision about whether the extension should be removed.

### 2.7.2 Firefox

Mozilla Firefox supports extensions through their WebExtensions API, which aims to be a framework for developing extensions that can also be made compatible with Google Chrome, Microsoft Edge and Opera with only a few modifications required to the extension [24].

Firefox, like Chromium, initially allowed extensions to be installed from any source. In 2015 with the release of Firefox 43 extensions had to be signed by Mozilla as part of their review process in order to be installed in the browser [25].

The review process for Firefox extensions consists of an automatic analysis, a content review where metadata and resources are manually searched for spam and unwanted material, a code review where the code is looked through, and finally a testing stage where the extension’s advertised functionality is tested [26].

Mozilla have published a static analysis tool for extensions which can be used by extension developers and reviewers [27]. Some of the problems this tool checks for are known vulnerable libraries, calls to dangerous functions, hidden files, reserved names and missing or invalid metadata [28].

### 2.7.3 Android

Applications (apps) on platforms such as Android are built on a different architecture than extensions in web browsers, although they still face some similar issues to browser extensions regarding sensitive data leakage and risk of malware.
2.8. Security risks with extensions

By default apps can only be installed from the Google Play store, unless this behavior is overridden by the user. Apps are signed by their developers, but as a measure to simplify the process of acquiring a signing certificate that is trusted by all devices, there is no verification performed that the certificate is trusted and valid upon installation [29]. The signing process instead guarantees the integrity and authenticity of the app by aborting the installation if it has been tampered with or an update has been signed with a different certificate.

Android uses a permission-based system to restrict access to certain resources, which allows app developers to set required permissions which are requested when the app is installed [29]. The user can then decide if the list of permissions is acceptable or not before the app is installed on the device. Some of the actions which require permission before use are accessing the camera, reading and sending text messages, reading/writing to external storage and using account information stored on the device [29].

Each app is run as a separate user with its own private data directory, which allows the system to utilize the operating system (OS) kernel to enforce separation between different apps [29]. A developer can bypass these restrictions for their own apps by signing multiple apps using the same signing certificate. The Android OS will then run these apps with the same user id, allowing each app access to each other’s files and processes.

Parvez Faruk et al. [30] mention a possible attack that utilizes this feature by separating suspicious combinations of permissions into two separate apps, which can then call upon each other to perform permission-protected actions. As an example, they suggest that one of the apps can request permission to read text messages while the other has permission to access the internet. A user seeing the two permissions on a single app may become suspicious, but if the app can only read messages and not access the internet it may seem safer.

In 2012 Google announced that submissions to Google Play (called Android Market at the time) would be dynamically analyzed for malicious behavior using an internally-developed tool called Bouncer. Bouncer is stated to search for known malware and unusual behavior, while also comparing the app to previously analyzed apps. A trial run of the analyzer reportedly reduced the amount of downloaded malware by 40 % over the span of one year [31].

Since Bouncer performs dynamic analysis an app may try to evade detection by altering its behavior, since code that is not run cannot be detected by dynamic analysis. This issue is similar to the one mentioned in section 2.2, where the virtual environment can be detected by the analyzed program. The environment in which apps are run can be identified by factors such as Internet Protocol (IP) address, accounts added to the device and the number of contacts, which can enable an app to delay activation of malicious code until it is running on a user’s device [32].

2.8 Security risks with extensions

Extensions can put the system integrity at risk since these extensions have the same level of access to the system as the program they are running in. Due to them being run as part of the main program they can use the privileges granted to the program in order to compromise the system, which the user may not realize when installing the extension. Some of the more general risks which can arise from malicious or poorly written extensions are described in this section.

2.8.1 Reading/writing files

For extensions to be able to work efficiently, they might have to be able to read and write files on the disk because it provides more permanent storage than the Random Access Memory (RAM). This could however become a serious security vulnerability if the extension is allowed...
to read and write arbitrary files on the hard drive which it should not have access to. The enforcement of file access rights is something the operating system handles on a per-process level whenever a process tries to access a file or resource \[33\]. If the extensions are given the same level of access as the main program, which they could perhaps be given by default due to laziness during development, an extension could for example write a separate malicious program to the startup folder in Windows. This would cause the malicious program to be executed every time the computer boots into Windows, even if the extension itself is removed.

An extension with read access to arbitrary files could also read saved files containing sensitive data from the main program. If the user has documents containing order lists or similar saved these are also at risk of being exposed.

However, because all the extensions are running in the same process as the main program, this problem is hard to prevent without blocking file access for the entire program. But file read/write operations in code is something that could be detected through static analysis before the extension is executed or published.

### 2.8.2 Keylogger

Another method that an extension can use to access sensitive data is to implement a keylogger, which records keystrokes and sends all entered text to the extension’s creator. Keyloggers can access user information that is otherwise not stored in the system, such as credit card information or passwords. A common method of logging keystrokes is to utilize a technique called hooking that exists in Windows. Hooking allows a program to process an event (such as a key press or mouse movement) before other parts of the system can react to it \[34\]. The Windows API function "SetWindowsHookEx" which is used for hooking is a common component of keyloggers \[35\]. While it has some uses in legitimate applications, it can be considered a potentially dangerous function and should perhaps be flagged by analysis tools.

### 2.8.3 Poorly written extensions

Aside from the malicious behaviors mentioned above, poorly written extensions can also result in an unsatisfactory user experience. An unstable extension that is not run in a separate process can cause the entire program to crash if an error or crash occurs in the extension. An extension that performs slow or too frequent calculations, or is stuck in an infinite loop, can affect the rest of the system by using such a large share of the available resources that other programs cannot function correctly. Since the extensions are a part of the main program the operating system will notify the user that it is the main program that has stopped responding and not the specific extension. This can lead to bad publicity for the company that publishes the main program since the users will think that the main program is unstable, due to there not being any indication that the fault occurred in an extension.

### 2.9 Evaluation

Measuring the performance of a software solution is not always obvious, as the requirements and conditions may vary. To make the evaluation relevant, a decision of what to measure has to be made. For a static analysis program, both time and the amount of false-positives is relevant to look at.

The time it takes for the program to execute could be a very relevant aspect when evaluating the result. Depending on where the tool is implemented, execution time could be more or less critical. It is interesting to both look at the time of each technique used such as Flex and Bison for comparison but also at the prototype with combined techniques.

False-positives are relevant because if the tool reports too many false-positives, the user of the tool will be desensitized to the warnings and may as a consequence discard an actual
security problem. By eliminating these false warnings the user is more likely to consider the tool as a useful resource instead of an annoyance.

2.9.1 Evaluating prototype methods

Sarah Heckman and Laurie Williams\cite{36} use three metrics called accuracy, precision and recall to compare different methods of classifying warnings as either true-positives or false-positives depending on whether or not they correspond to a code defect.

The authors describe the accuracy as the number of correct classifications expressed as a percentage of all warnings. The precision is the percentage of identified true-positives that correspond to a defect in the code. The recall is the percentage of classified true-positives compared to the total number of defects in the code.

A high accuracy therefore means that the classifier correctly identifies defects, while the precision and recall shows how many of the classified defects were actually present in the code and how many of the defects that were correctly classified, respectively.

Together with predetermined classifications of all warnings, four intermediary statistics can be determined:

- The number of true-negatives (TN), where a warning is correctly identified as not corresponding to a defect.
- The number of false-negatives (FN), where the classifier treats a defect as a false positive.
- The number of true-positives (TP), where the classifier correctly identifies a warning as a defect.
- The number of false-positives (FP), where the classifier incorrectly assumes that a warning corresponds to a defect.

Using these four parameters, the authors give the following equations for calculating the accuracy, precision and recall:

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]
\[
\text{precision} = \frac{TP}{TP + FP}
\]
\[
\text{recall} = \frac{TP}{TP + FN}
\]

While the authors compare different strategies for classifying warnings, the same metrics could also be used for evaluating the code analyzer that produces the warnings. A true positive would then be a warning that corresponds to a security problem, while a false positive is a warning that upon inspection does not correspond to any defect in the code. A true negative would be a benign piece of code that does not generate a warning, while a false negative is a security problem that the analyzer does not warn about.

2.9.2 Clustering of warnings

Ted Kremenek et al.\cite{37} present evidence that suggests that the true-positives and false-positives reported by a static analyzer tend to cluster together in source code. They attribute this to the fact that programmers tend to repeat mistakes when working on a file or function, which can result in several true-positives warnings being generated for that file or function. A false-positive warning may likewise occur due to the programmer writing code that utilizes unusual language features, which they might continue to do elsewhere in the file or function.
A static analysis tool can utilize this behavior in combination with user-supplied information about whether a warning is correct or not in order to give lower or higher priority to other warnings according to the received feedback.
3 Method

3.1 Static analysis

As the first step in creating a static analytic tool, research was performed through published articles, books and online sources for information about existing approaches to static code analysis. Since some of the already existing static analytic tools are open source, inspiration and knowledge could be gathered from those projects as well. However, as this project is about creating a new static analysis tool for a lesser known language none of the existing ones are directly usable. The report will also investigate if the built-in functionality of the language can be used to achieve better results.

The creation of this tool involved creating prototypes to see which methods that could work for detecting unauthorized access to sensitive data. These prototypes are explained more in detail in section 3.2 below.

3.2 Prototype

3.2.1 C++ string matching

A basic but flexible method of detecting threats in source code is to utilize string matching, which is the process of detecting patterns in larger sets of text. While this method has some disadvantages, namely that it does not consider context and cannot understand the semantics of the language, it is easy to adapt to different languages and situations. If the set of patterns is restrictive enough that they are unlikely to be matched outside of the intended context, the technique can be useful for quickly analyzing large amounts of data.

String matching is related to lexical analysis (discussed in section 2.5) as both methods involve recognizing specific patterns in text. Lexical analysis is a core part of existing tools such as Flawfinder and ITS4, which use it for identifying function calls [7,3].

Due to the flexibility and simpler nature of string matching it can be useful for identifying URLs, email addresses or specific function names that are unlikely to appear in other parts...
3.2. Prototype

of the text. It can also serve as a fallback for files or specific lines that could not be analyzed using other techniques, which may lower the risk of missing a potential issue at the cost of an increased false-positive rate.

C++ was chosen as the implementation language for the prototype because some of the company’s other infrastructure is based on C++, which could ease integration in later stages. The goal of the C++ prototype is mainly to use regular expressions together with simple string matching as a general way to scan arbitrary files in any language. This has two purposes: finding suspicious words or patterns, and finding known malicious URLs. Even if it is not very likely that there are any malicious addresses present in an extension, it demonstrates the possibility to extend the tool and gives a slight peace of mind that extensions at least are free from some of the most obvious malware.

In order to know which addresses are malicious, the URLhaus database from section 2.6 is to be integrated into the prototype. To avoid using an outdated database it could for example be downloaded from the internet whenever the program is started and the local copy is older than a few hours.

The tool could also be used to locate any address or IP in the source code, which could be useful to a human analyst who can investigate what servers an extension might connect to and where in the code this occurs.

Each line would be analyzed by using regular expressions to find either an address (URL or IP) or a word that is on a predefined blacklist. It is only when an address is found that the program should try to match it with an entry from the database, which, compared to iterating over the database, is both faster and enables the feature of printing any address that appears within the scanned files. To achieve faster performance, the blacklisted URLs can be stored in a hash-based lookup table (unordered_set in C++), which has a constant lookup time on average. Thus a larger list will not noticeably impact the time required to scan through a file.

One drawback with this approach is that the address has to exactly match the one stored in the database. In the prototype this problem can be mitigated by ensuring that the stored URLs and the found addresses follow the same format regarding letter case and the presence of trailing URL characters such as '/' or '?'.

The blacklisted words should be stored in a text file on disk where each word is on a separate line, as this format would allow for easy extension of the blacklist without having to recompile the program. Each line is treated as a regular expression pattern, which means that it could be used for a simple word match or a more advanced pattern that depends on the surrounding characters.

Every time the program finds a match, it should be printed together with file name, line number, the word or URL and the whole line as context. This will make it easier to trace back what has been found and enables quicker analysis by the user since the relevant piece of code can be found easily.

3.2.2 Parsing using Flex and Bison

Parsing the source code of an extension into an abstract representation can be useful for performing more in-depth analysis by using information about types and defined symbols to differentiate between local and global variables and overloaded functions [38]. But for languages such as C/C++, where the behavior of statements may be altered at compile-time through use of the preprocessor, the parser may fail to analyze the code. In situations where real-time analysis is desired, such as in text editors when code is being written, parsing an entire file may take too long to be useful or fail due to unfinished code that does not yet compile [3].

Flex and Bison (see section 2.5) can be used to parse the source code of the extensions in order to find function calls and variable accesses. The advantage of using a parser to process the code instead of searching for patterns representing function calls is that the parser can
understand the context in which a match is found. This allows it to ignore comments and
variable declarations, both of which can contain the same pattern as a function call.

A comparison between string matching and parsing for the function `getPrice()` can be
seen in listing 3.1 and 3.2, where the underlined statements indicate which parts of the code
the analysis tool would warn about.

```
getPrice(); // Function call
/* getPrice(); */ // Comment
int getPrice(); // Variable
```

Listing 3.1: Example detections with string

```
getPrice(); // Function call
/* getPrice(); */ // Comment
int getPrice(); // Variable
```

Listing 3.2: Example detections with parsing

### 3.2.3 Built-in language features

The language that the company’s software is written in has some features that, when compil-
ing in debug mode, allows programs to interact with the compiler in order to gain access to
its intermediate representation (IR) of compiled code. The IR is a partially compiled version
of the source code that the compiler can later translate into machine code. An example of
how source code is represented in IR is displayed in listings 3.3 and 3.4. However, these
features are only available to programs written in the same language. Therefore, this part of
the prototype cannot be written in C++, which will likely limit the possibilities of interaction
with the components discussed above.

Having access to the IR makes it possible to first compile the extension that is to be
analyzed, then iterate over the instructions in the compiler’s IR for every function belong-
ing to the extension. Instructions representing function calls or accesses to variables can
then be further analyzed by asking the compiler for the file path where the requested ob-
ject is located. If the object is located in the same folder as the extension or in the folder
of the main program, the access is deemed safe. If the object belongs to another exten-
sion or is in a predetermined list of possibly dangerous functions, a warning can be emit-
ted from the analysis program. This results in that the analysis tool will ignore function
calls and variable accesses that occur within an extension, as these occur in every legiti-
mate object-oriented program. An example of the directory structure of the extensions is
displayed in figure 3.1. A function in `extensions/extension_1/core` should be able to
access its resources in `extensions/extension_1/resources`, but not the resources in
`extensions/extension_2/resources`.

```
int value = 100;
value = value / 2;
printf("%d", value);
---
getPrice();
/* getPrice(); */
int getPrice();
```

Listing 3.3: Example source code

```
set [local_var1 int] 100
div [local_var2 int] [local_var1 int] 2
push [local_var2 int]
push "%d"
call [void printf(str format, int data)]
---
call [int getPrice()]
set [local_var3 int] 0
```

Listing 3.4: Example IR
3.2. Prototype

3.2.4 Combining multiple approaches

If the prototype is to be integrated into other systems, it would be beneficial if all analyzer parts could be run as part of a single invocation of the program. This will also make it easier to run the analyzer manually as all functionality would be exposed through a single interface.

Combining multiple methods may also allow them to exchange information and complement each other if one method fails to process a file or code segment. For example, gathering a list of files which should be analyzed may be performed by the first analyzer, which can then pass this list to the other ones in order to reduce the file system lookup overhead.

3.2.5 Extending modules

An important part of the prototype is for it to be extendable with additional rules or functionality. As this work is limited in scope it does not include a study of which function calls and behaviors that should be reported. In order for the prototype to be useful it should therefore be able to be extended at a later time when such a study has been performed.

The integration with URLhaus serves as an example of how an external blacklist can be incorporated into the prototype, as it can easily be updated from an external source when becoming out of date.

3.2.6 Evaluation

Since the static analysis tool will not run as part of the main program, it does not have to be included when shipping the software to the end user. As a result of this, the tool will not impact the performance of customer systems or the user experience at all, due to it never being run on their hardware. The time required to run the analysis is therefore only of interest to the developers or reviewers that analyze extensions before they are published. If the rate at which extensions are submitted for publication is low enough, it might be acceptable for the analysis to run for a longer period of time. While not completely irrelevant, the execution time of the analyzer should perhaps not be prioritized over other factors such as detection rate or the number of false-positives.

Currently, the company does not have any examples of malicious extensions that read each other’s data. This creates some problems with evaluation since there are no known malicious code samples to test the prototype on. Attempting to create malicious extensions for testing purposes may give misleading results regarding the effectiveness of the prototype since it will not be possible to predict which methods a developer of an extension may use. But even if it may give a misleading result in regards to effectiveness, it could still be relevant to create an example of a malicious extension in order to demonstrate the prototype. This example extension should attempt to retrieve price information from other extensions and contain different forms of references to them, such as variable accesses and function calls. Since the prototype is to be developed by continuously testing it against the created extension and not
considered complete until all malicious parts are detected, there should not be any possibility for false-negatives to occur.

Not having any real-world examples of malicious code also creates problems with calculating the comparison metrics described in section 2.9.1.

Instead, the performance of the prototype will have to be mostly evaluated by the number of false-positives. Since there exists over a dozen extensions that have been developed internally, these can be used to test for warnings that do not correspond to an actual malicious bit of code. Since these extensions are developed internally without malicious intent, they should not produce any warnings about data security issues.

But as mentioned in section 2.9, time will also be used for evaluation as it is still relevant to see the how fast a solution is. It will both be used to evaluate the different techniques used in the prototype and the complete prototype itself.

As there is not currently any process in place for reviewing extensions before they are published, any prototype that finds some issues while not requiring unreasonable amounts of resources is worthwhile to the company. As long as all cross-references that were put into the malicious extension created for this thesis are detected and no incorrect cross-references are reported in the internally developed extensions, the prototype can be valuable for the company’s extension submission process.
4 Results

4.1 Prototype

4.1.1 C++ string matching

Implementing string matching using C++ as described in section 3.2.1 resulted in a very general and flexible tool. It is able to scan through all the files with a certain file extension within a given directory according to a user-defined list of file extensions. In theory, this means that it could scan through binary files as well if there is a suitable binary pattern specified which the program can use to match files. When the program is run, it reads a text file by default containing directory paths with corresponding file extensions of which to be searched. The program iterates through all the files that has the given file extension and adds them to a list. Each file in this list is then individually scanned, line by line and word by word, comparing against regex patterns which are explained in section 2.4.

The program is structured to find two different things. The first is blacklisted words or other text patterns, which are specified in a text file. This file contains regex patterns which can be used to find specific words or more advanced patterns in the analyzed source code. The reason for putting these in a separate text file is to make it easier to adjust the patterns or to extend the list with additional ones.

The second thing that the program searches for is URLs that are known for containing malicious content. The same technique with regexes is used, however, as the structure of these URLs will not change, a regex has been hardcoded into the program which matches both IP addresses and URL addresses. Every match is added to a list of findings, storing information such as what file, line number, the matching string, the surrounding text on the same line, and whether it was a word or a URL. To keep the individual findings separate but at the same time store all this information about a specific incident together, a data structure called suspects was created containing all the information listed earlier.

The known malicious URL file has its origin from URLhaus [18] which is downloaded upon execution if the current one stored is older than 1 hour. This is to make sure the list is always up to date without having to download the same file an unnecessary amount of times.
By default, the result is printed only with the total amount of searched files and total amount of matches in plain white as seen in figure 4.1. This is because it is supposed to be easy to integrate with other programs.

![Figure 4.1: Default string matching output](image)

The way the output is presented can be changed using command-line flags, allowing to see the output represented in color, see figure 4.2. As for now, the program supports showing all URLs found, the currently scanned file, print the matched word when they occur, print all findings, terminal color and to specify the path that should be looked at together with the file extension.

![Figure 4.2: Colorized string matching output](image)

### 4.1.2 Parsing using Flex and Bison

Creating a parser for source code using Flex and Bison proved difficult to implement with reasonable accuracy. Due to the flexible nature of the targeted programming language, where keywords such as `if`, `for` and `function` were not reserved, the parser failed to recognize more complex statements due to ambiguities between different language rules. For example, an ambiguity occurred in the code snippet shown in listing 4.1, which closely resembles actual code that was in production software. The statement `if (if)` performs a check equivalent to `if (if != null)` on the function named `if` (that is defined on the line above), while `if (value)` calls this function with `value` as an argument. Since an if-statement could be written without any actions, `if (value);` could both be a valid if-statement and a function call depending on whether or not there is a defined function named `if`.

```c
int value = 0;
Function if = ...;
if (if) if (value);
```

Listing 4.1: Ambiguous if-statement

When testing the parser on the entire codebase for the company’s software, ambiguities such as this and other complicated statements made the parser emit a large number of errors. Attempts to fix these would sometimes lead to other errors being generated elsewhere due to the newly added rules conflicting with other statements that could be successfully parsed before. For example, table 4.1 shows a situation where a change to the parser increased the number of files that could be parsed without errors, but the number of errors in the remaining files increased by a large amount.

While it is not necessary for the parser to be able to parse all possible statements, the process of recovering from an error involves having to ignore parts of the statement in order
for the parser to get back into a known state. Thus, every error brings the risk of missing an interesting function call if the error occurs on the same line as the function call.

<table>
<thead>
<tr>
<th></th>
<th>Version ( n )</th>
<th>Version ( n + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total files parsed</td>
<td>20798</td>
<td>20798</td>
</tr>
<tr>
<td>Files parsed successfully</td>
<td>11818</td>
<td>7534</td>
</tr>
<tr>
<td>Files with parsing errors</td>
<td>13179 (63.63%)</td>
<td>7534</td>
</tr>
<tr>
<td>Total errors</td>
<td>20733</td>
<td>35712</td>
</tr>
</tbody>
</table>

Table 4.1: Bison prototype parser statistics

An example of the output from the parser can be seen in figure 4.3. Some statistics about the analyzed files are printed at the top, followed by a list of reported warnings about potential issues. The part which the parser reported is highlighted in red. Each warning is displayed with accompanying information about file name, line number and some code context around the point where the parser found a suspicious function call. Note that the file name and file extension have been altered for this example, when run normally the analyzer will display the correct name and extension. Figure 4.4 shows the analysis result for code similar to listing 3.2, here the parser correctly reports the function call while ignoring the similarly formatted comment and variable declaration.

![Figure 4.3: Bison prototype output](image)

Figure 4.3: Bison prototype output
4.1. Prototype

Figure 4.4: Bison prototype run on comment and variable declaration

4.1.3 Built-in language features

Using the built-in features of the programming language in which the company’s software is written allowed for a more robust analyzer to be created in a shorter period of time than with Flex and Bison. Since the compiler must be able to parse all valid constructs in the language, all programs that can be compiled and run can also be analyzed by our analyzer.

As described in section 3.2.3, using the available information about in which file a function or variable was defined enables the analyzer to identify when a statement attempts to reference an object from a file belonging to another extension. This can occur automatically without having to specify which other extensions that exist, but legitimate function calls to features exposed to extensions by the main program are also flagged. In order to eliminate these false warnings a whitelist of paths that are allowed to be accessed from extensions can be specified in a text file, as seen in listing 4.2. Each line contains a regex which can match one or more paths on disk, in the listing below the paths `main/*` and `core/*` will be matched.

```
main /.*
core /.*
...
```

Listing 4.2: Format of the path whitelist

The analyzer can also detect references to functions or variables which match an entry in a blacklist of possibly dangerous object names. An example entry in the blacklist is shown in listing 4.3. Each entry contains a regex for the function name or variable, an optional regex of paths where the name can be matched, a severity level which the analyzer can use to format its report appropriately, and a message which is included in the report.

```
getPrice; paths=core/items/priceData/./.*| severity=warning| message=
Retrieves price information, acceptable if the referenced object belongs to the calling extension.
...
```

Listing 4.3: Format of the name blacklist

An example of a report generated by the analyzer is shown in figure 4.5. Issues with a severity level of critical, warning and info are displayed with red, orange, or blue text.
respectively. For each detected issue a line is printed with the location and function where the issue was detected, along with the targeted function or variable and its location. The message defined in the blacklist is also displayed to inform the user of what risks the issue could entail, as well as conditions that must be true in order for the issue to not be a false-positive.

The report is intended to be viewed in GNU Emacs, a popular text editor, where references to source code locations can be clicked on to open the relevant file and line in the editor.

Issues can be hidden from the report if they are below a certain severity level, which enables the user to only focus on the most important issues. For example, functions such as `getPrice()` are frequently called by extensions, but are only a security issue if called with another extension’s data as a parameter. These can be given a lower severity which allows them to be filtered out of the report.

Cross-reference analyzer:

Access to cross-package function in other extension
- `c:\program\extensions\test_extension\test.c(238, 57): test()
- `c:\program\extensions\other_extension\test.c(50, 33): void otherFunction()

Reference to data or function from another package may indicate bad behavior if the target is another extension.

Access to blacklisted cross-package function in core
- `c:\program\extensions\test_extension\test.c(242, 37): test()
- `c:\program\core\coreFunctions.c(81, 38): bool detectProcess(str name)

Functions in core can be used to start external processes or manipulate the system.

Access to blacklisted cross-package function in core
- `c:\program\extensions\test_extension\test.c(241, 25): test()
- `c:\program\core\coreFunctions.c(65, 21): bool asyncCmd(str cmd)

Functions in core can be used to start external processes or manipulate the system.

Blacklist analyzer:

Call to blacklisted function in core/items/priceData
- `c:\program\extensions\test_extension\test.c(218, 60): test()
- `c:\program\core\items\priceData\prices.c(214, 35): double getPrice(Item this)

Retrieves price information, acceptable if the object "this" belongs to the calling extension.

Progress: 30/30 (100%)
Warnings: 4
Critical: 1
Warning: 2
Info: 1
Hidden warnings: 0
totalTime: 4 ms
Done

Figure 4.5: Example of report generated by the analyzer using language features

4.1.4 Combining multiple approaches

As mentioned in section 3.2.4 the goal for the final analyzer prototype was to combine the three approaches, using them for their respective strengths and attempting to mitigate weaknesses by combining information from different modules or falling back on another approach when one fails. As shown in table 4.1 however, the Bison parser tended to fail a lot, which would have made it difficult to ensure that a certain file was properly scanned and did not contain
any malicious behavior. After evaluating the built-in features of the language and researching how they could benefit the prototype it became clear that the Bison parser would not add enough value to justify spending the time to improve the error rate.

Development efforts in the later stages of the prototype were instead focused on integrating the C++ string matcher with the analyzer that is based on language features, as these excel at different purposes. In the final prototype the string matcher is fed paths to analyze from the language-feature analyzer, and its results are printed as part of the complete report. This avoids having the user run two separate programs to complete a full analysis. An example of this output when run on specially crafted test data is shown in figure 4.6 where the string matcher has found a malicious URL and the __asm directive, which is used to insert low-level assembly code into the program.

Figure 4.6: Example of report generated by the C++ string matcher as part of the combined analyzer

### 4.1.5 Extending modules

The prototype was created with extensibility in mind, which makes it possible to add more features and functionality. Both the blacklist with words and the whitelist could easily be extended and tuned through a text file, making it possible to add any word or regex pattern without having to recompile the code.

The integration of URLhaus is a concrete example of how the software can be extended and integrated with other components. The same method used to download the URL blacklist could easily be used to download an arbitrary list of words or patterns. Other C++ programs could be created in a similar way and be added into the main analyzer, adding even more features.

### 4.1.6 Evaluation

The prototype was evaluated on two extensions belonging to some of the company’s largest customers, as the company had access to the source code of these for support purposes. In the future all third-party extensions must make their source code available to the company. The results are shown in table 4.2.

All tests were performed using an Intel i9-9900K @ 3.6 GHz with 32 GB of system memory and a Samsung 970 PRO Solid State Drive (SSD).
4.1. Prototype

<table>
<thead>
<tr>
<th></th>
<th>Extension E1</th>
<th>Extension E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total files</td>
<td>2847</td>
<td>1846</td>
</tr>
<tr>
<td>Functions parsed</td>
<td>122323</td>
<td>58797</td>
</tr>
<tr>
<td>Cross-references</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OS function calls</td>
<td>20</td>
<td>42</td>
</tr>
<tr>
<td>Calls to getPrice()</td>
<td>36</td>
<td>47</td>
</tr>
<tr>
<td>Malicious URLs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Time taken</td>
<td>6 min 52 sec</td>
<td>3 min 20 sec</td>
</tr>
</tbody>
</table>

Table 4.2: Evaluation results on externally developed extensions

In addition to the third-party extensions above, some internally developed extensions were also analyzed. The results are shown in table 4.3.

<table>
<thead>
<tr>
<th></th>
<th>Extension I1</th>
<th>Extension I2</th>
<th>Extension I3</th>
<th>Extension I4</th>
<th>Extension I5</th>
<th>Extension I6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total files</td>
<td>31</td>
<td>183</td>
<td>102</td>
<td>45</td>
<td>40</td>
<td>376</td>
</tr>
<tr>
<td>Functions parsed</td>
<td>347</td>
<td>4493</td>
<td>3082</td>
<td>715</td>
<td>775</td>
<td>14591</td>
</tr>
<tr>
<td>Cross-references</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>OS function calls</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Calls to getPrice()</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Malicious URLs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Time taken</td>
<td>2 sec</td>
<td>12 sec</td>
<td>12 sec</td>
<td>9 sec</td>
<td>4 sec</td>
<td>41 sec</td>
</tr>
</tbody>
</table>

Table 4.3: Evaluation results on internally developed extensions

The rows for time taken represent the total time required to run the combined analyzer (C++ string matching + language feature based). The time requirements for the C++ component and the language feature based component relate to each other by the ratio 1 : 39, i.e. the time required to run the C++-based analyzer constitutes 2.5% of the total time.

Comparing the different approaches presented in section 3.2 using the metrics mentioned in section 2.9.1 was not directly possible due to the different methods focusing on detecting different patterns in the code.

The lack of malicious extensions to test the prototype on also prevented the calculation of these metrics for the reasons discussed in section 3.2.6. They would have been useful for comparing the prototype with other existing products designed to detect similar issues, but as mentioned in section 1.1 such tools do not exist for the programming language which the extensions are written in.

As the company currently does not have any defined rules of what actions an extension is allowed to perform, evaluating the results according to the metrics in section 2.9.1 is difficult. Differentiating between a false-positive and a true-positive requires knowledge of whether the action represented by the reported piece of code is permitted or not, which has not been clearly defined by the company yet. Clustering warnings such as mentioned in section 2.9.2 is also not possible without knowing which warnings are true-positives and false-positives.
5 Discussion

5.1 Results

5.1.1 Output from C++ tool

The output from the C++ tool is quite straightforward, showing total files searched and the total number of findings. As it is a tool that should be able to run both as a standalone tool and to be integrated and run within other programs. The program was designed to be able to scan the files even if the other methods failed. This meant that it had to be general enough to read any file, and not be too complex in its default state. The program should however be able to be extended with more functionality.

One advantage with the C++ tool is that it is fast and can be run on different file types. It is however not as intelligent in its current state as the other methods tested, since it cannot differentiate between code, comments and strings.

5.1.2 Output from using built-in language features

The language feature-based analyzer performed quite well regarding the detection of cross-extension references. As discussed in section 3.2.6, the true-positive rate of the analyzer is hard to evaluate due to the lack of existing malicious extensions, but observing the number of cross-references when run on two external extensions showed that there were no false-positives reported. Some warnings for OS functions and calls to getPrice() are to be expected as the analyzer cannot determine the intent of these calls, but as the main purpose of the analyzer is to detect cross-references a lack of false-positives in these tests is important for the tool to be usable.

The fact that the analyzer was able to detect cross-references in extension i6 (see table 4.3) could be considered a false-positive, as it was developed internally and should therefore not be malicious. However, as these references are not that common and should not occur in third-party extensions it can be argued that every instance of this should be treated as a critical warning. Internally developed extensions should perhaps not be subject to the same restrictions as externally developed ones would be, and instead be allowed to interact with
5.1. Results

Other extensions. This could be extended to third-party extensions as well, where access to another extension’s data is allowed if both extensions are developed by the same company.

5.1.3 Using a combination of tools

Combining the different tools was not as straightforward as initially planned due to the inclusion of the analyzer using language features, which could not be integrated into the C++ prototype. Instead the compiled version of the C++ program had to be launched from the language feature analyzer, which meant that the interaction between them was limited to sending text data back and forth.

Despite this limitation the C++ analyzer could still receive paths to analyze and return the analysis in text form, although the colored text from figure 4.2 cannot be displayed correctly when output as part of the report from the language feature analyzer.

5.1.4 Evaluation

As shown in table 4.2, both of the evaluated extensions that were developed externally consisted of a large number of files and functions. Neither of these contained any references to other extensions, which was to be expected as they were developed by reputable external companies. Both made use of the OS and \texttt{getPrice()}, with E2 having more than three times the amount of warnings per function compared to E1. E1 had approximately 1 warning per 2200 functions while E2 had 1 warning per 660 functions.

The majority of the OS function calls were used to open a URL in a browser or to start an external program. Opening a URL is considered an OS function and is therefore reported, but it might not present a risk to the system or the user, especially since all URLs are compared to the URLhaus blacklist described in section 2.6. The functions related to URLs could perhaps be added to a whitelist or reported with a lower severity to focus on the more important warnings.

Starting an external program may indicate that an extension relies on a background service for fetching data, or that it attempts to hide functionality or run arbitrary commands on the system. Each call to the relevant OS function will have to be manually analyzed to determine the purpose of the call.

The reason for why OS functions are reported is that these are very powerful and can be used for malicious purposes, some of which are explained in section 2.8. Since the company has not yet determined which calls should be allowed, every use of OS functions is reported by the analyzer. By doing this it will be easier for the company to go through all produced warnings and decide whether or not that specific behavior should be allowed.

Unlike the OS functions, where the name of the function may indicate whether or not it could be used maliciously, the \texttt{getPrice()} function is commonly used in benign extensions as well. The usage of this function can only be considered malicious if the data passed to it belongs to another extension. The calls to \texttt{getPrice()} will therefore have to be manually analyzed to determine which extension the passed in data belongs to. Like with the OS functions, all occurrences of \texttt{getPrice()} are reported in order to simplify the manual analysis and to provide a worst-case figure of the number of possibly suspicious calls to this function.

From the results presented in table 4.3 it can be seen that extension I2 performs a call to \texttt{getPrice()}, which can quickly be confirmed as safe by manual review as the function is only called on data internal to the extension. Extension I6 performs 6 OS function calls. These will have to be further reviewed as they launch external programs which cannot be automatically analyzed.

Extension I6 also contains 25 references (function calls or variable accesses) to other extensions, which is the type of warning that this work considers to be the most important. By looking through the generated warnings it became clear that all these were to a single extension. This other extension contained developer tools for modifying objects that are used in the
software, while extension I6 contained developer tools for publishing collections of objects. In this case the cross-references were legitimate and expected, as the metadata for I6 stated that it depended on the other extension in order to function.

5.2 Method

5.2.1 Reason for excluding Flex and Bison

As mentioned in section 4.1.2, the Flex and Bison parser could not successfully parse enough files to be considered useful in its current state. Spending time on improving the parser could likely increase the number of files that are parsed without issues, but this figure will probably never reach the same level as the compiler which must be able to parse all statements that are valid in the language.

Detecting accesses between extensions would also have required large additions to the parser and perhaps multiple passes through the code in order to be able to resolve references and differentiate between local and global variables and functions.

A benefit of improving the parser is that it would have provided an excellent foundation for future expansion, as information about symbols, types, control flow and data flow could be used to implement some of the features present in tools such as the ones mentioned in section 2.3.

5.2.2 Things that could have been done differently

Defining acceptable behavior

To make this prototype more useful, there has to be concrete rules about what should be allowed and what should not. As it is now, no such rules existed which made it hard to detect real violations.

As mentioned in section 3.2.5, there does not currently exist any definitive rules of what an extension is and is not allowed to do. Analyzing and mapping which functionality extensions currently use, what should be allowed and what privileges they should have could help improve the ruleset used in the analyzer to increase the quality of the warnings and prioritize them better.

Likewise, the blacklist of dangerous function calls and words could be extended through a similar study, which would allow the analyzer to provide more specific warnings and comments about unsafe behavior. This could also help reduce the large amount of warnings about OS functions shown in table 4.2, as functions determined to be safe could be removed from the blacklist.

Request more clear specifications

The requirements for the prototype were not clearly defined, thus some assumptions had to be made about how the prototype would have been used. For example, the language feature-based analyzer needed to be run in debug mode, which could seemingly only be done through the GNU Emacs text editor. As such the output format of the prototype was designed to take advantage of the editor’s ability to display colored text and clickable references to source code. If the prototype is to be run as part of the extension submission process it has to be run automatically, which may make the current output format a poor choice if the report is to be displayed as part of a web page or saved in a database.

Requesting more clear specifications about how the prototype was intended to be used would perhaps have made it easier to integrate it into other systems, as machine-readable formats such as XML or JSON could have been added as output options.
5.2.3 Source criticism

Many of the sources used are from online documentation or from GitHub\textsuperscript{1}, mainly because that is, in many cases, the most updated source for information about software projects. Especially when it comes to open source projects, but also for documentation from companies such as Microsoft, Mozilla and Google.

These sources were used together with published scientific articles from publishers such as the ACM and IEEE. The articles were selected with consideration given to relevance, the number of articles which reference the chosen article, and the time since the article were published. Some of the articles included are around 10 years old but may still be relevant as most of the underlying issues with software security and the methods of combating them have not changed drastically during that period.

A few books have been used as references for background information in the theory chapter, since they provide a good summary about the history of a technology or a description of a platform.

5.3 The work in a wider context

In a wider context, static analysis tools are a great way to detect vulnerabilities and bugs in large amounts of code without manually auditing it. Several other companies which provide programs with extensions use static analysis as a part of their auditing for malicious code, as mentioned in section\textsuperscript{27}. Static analysis provides a consistent and reliable auditing method and can minimize the amount of errors or mistakes. Automatically scanning code using static analysis and then manually auditing the result could increase the overall security for the entire code base. However, no matter how good the analyzer tool is, you have to understand what to look for and what the tool does.

"A fool with a tool is still a fool" - Ron Weinstein

As the language used by the company is a lesser known language, existing tools for static analysis cannot be used as they do not support it. The development of this tool could help the company to analyze their code in an automatic way.

The tool could be used as part of the extension submission process, so that the source code of every submitted extension is scanned automatically. An analyst could then review the produced warnings manually and evaluate if the extension is suitable to be published. The source code can then be compiled and the resulting binary can be published.

The societal aspects our work and static analysis as a whole are not that pronounced, although there is a possibility that the overall security can increase for any program if the right tool for the right language is used. Buffer overflows in a program could for example be found very early and fixed before it is published.

But depending on the implementation of a static analysis tool and what policies it follows, there could be some ethical problems. The technique could be used as an upload filter, to filter out certain documents or code of any kind. It would be an ethical problem if it would be used for political gain or censoring on a larger scale. The policies used to define analysis rules should therefore be justified and used with responsibility.

\textsuperscript{1}GitHub is a repository of open source projects with version control
The purpose with this thesis was to produce a working prototype that uses static analysis to detect when an extension accesses sensitive data from another extension. Because the software in question is based on a lesser known language, a new tool had to be developed as there were no existing ones available. This tool fulfills the aim of the thesis, as it is capable of detecting cross-references in extensions, which may indicate that an extension is trying to access sensitive data.

While this thesis does not contain any major advancements within the field of static analysis, it can hopefully serve as an example of how such an analysis tool can be constructed and what the advantages and disadvantages of different methods are.

Through researching possible methods for performing static analysis and through developing the prototype, the first research question could be answered.

- How can access to sensitive data be detected in source code using static analysis?

The presented prototype can search through existing extensions and find cross-references between them, classify the severity of the event, and then print a report for manual review. It utilizes both the lesser known language and C++, letting each component focus on different problems and then collecting the results. It is able to detect and locate potential threats such as cross-references, blacklisted URLs and blacklisted words.

The Flex and Bison parser provides an alternative approach which requires more setup and studies of the language that is to be parsed before it can be useful, but would provide more flexibility when completed. The C++ analyzer is an example of a simpler approach to the problem, which has a more generic implementation and runs faster, but at the cost of an increase in false-positives.

- How can the performance of a static analysis method be measured?

For the second question, the most representative methods of measuring the performance seemed to primarily be by true/false positive rate and secondly by execution time. In the scenario where the prototype will be used, as part of the submission process for extensions, the time was not a crucial factor, although still interesting to some degree. This is because the number of incoming files to search through is not expected to be that great due to extensions
being submitted at a low rate, and that the prototype is supposed to be run as part of a final check of the code, not continuously during development.

The true/false positive rates were considered the more relevant factor, as these will be a crucial part of whether the tool will be useful or considered annoying due to reporting too many false-positives. Due to the lack of malicious extensions however, the true-positive rate will be more difficult to test.

6.1 Future work

6.1.1 YARA

One possible approach to take in the future is to integrate YARA rules, which define text-based or binary patterns to search for in the code or in binary files [39], into the analyzer. However, this approach is used more within the field of malware analysis and detection. If a new tool was to be developed with focus on malware threats, YARA rules would be a good module to include.

6.1.2 Extending the parser

Improving the parser grammar specification and extending it to consider the additional information described in section 5.2.1 could be beneficial if the dependency on the compiler is to be reduced. As the analyzer currently depends heavily on the language compiler, the code of an extension must be partially compiled before it can be analyzed. By extending the parser the functionality of the analyzer could be improved in situations where the compiler is not installed or when parts of the source code are incomplete.

6.1.3 Taint analysis of sensitive data

An alternative to identifying cross-extension function calls could be to instead track the flow of sensitive data through the program through a method called taint analysis. David Yu Zhu et al. [40] present a dynamic approach to this, where the taint-tracking code is run before each instruction that is executed by the program. This allows for sensitive data to be marked as tainted, and for this tainted state to be propagated throughout the program whenever an operation that depends on a tainted value is performed. This could perhaps be extended to include the source of the tainted data, which could then be checked when the data is used by an extension. If the data originates from another extension a warning could be presented to the user of the analysis tool.

Taint analysis can also be performed statically, as demonstrated by Steven Arzt [11] who created a tool which performs taint analysis on a simplified intermediate representation of Java code.

6.1.4 Clustering warnings

As mentioned in section 3.2.6, Ted Kremenek et al. [37] have shown that false-positives and true-positives commonly occur in clusters. By using input from the user regarding which warnings correspond to an actual issue and which are false-positives, the analyzer could adjust the priority of a warning if the user has provided input on another warning located in the same function or file.
6.1.5 Tracking warnings across versions

An improvement to the prototype that could give insight into the lifespan of detected issues would be to save old reports and compare to the current one, which would enable the analyzer to identify which issues that have been reported previously. Jaime Spacco et al. [41] present a method for this which uses information about the variable names and function names related to the issue to generate a hash value, which can then be compared across versions to identify where two issues from different reports point to the same statement in the code.

This method could be used in combination with the clustering and user provided input discussed above to assign an issue the same priority as the previously detected instance, or to prioritize new issues that were not present in a previous version of the same extension.


