Bachelor thesis

Design and Implementation of Obfuscation Techniques with Predictable Size and Execution Overhead

Leif Brötzmann

Kiel, Germany 2021

Kiel University
Department of Computer Science
Distributed Systems Group

Supervisor 1: M.Sc. Patrick Rathje
Supervisor 2: Prof. Dr. Olaf Landsiedel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 27. September 2021

Abstract

Predicting the overhead of existing obfuscation techniques and tools is difficult and imprecise. Non-determinism is a key property of most obfuscation techniques because it makes them more effective. This results in the protected programs being internally different for each run of the tools. When working with resource constraints like on embedded devices, this causes programs to have varying resource costs, which may not always fit the restraints. Furthermore, it becomes hard to predict how modifications to the original program affect the resource cost of the obfuscated program. This work presents methods of making existing established techniques predictable and designs and implements a framework to achieve byte precision for size prediction and instruction count precision for the runtime overhead. The resulting implementation achieved the precision goals at the cost of using multiple times the resource cost compared to existing tools for larger programs.

Acknowledgements

I want to thank everyone who supported me in finishing this work, especially my supervisor Patrick Rathje for the long discussions and great feedback in the theme finding, development, and writing phase. My thanks also go to Prof. Olaf Landsiedel for making it possible to write my thesis on this topic even though it isn't within the usual fields his workgroup researches.
Special thanks to all my friends and internet strangers, with whom I had the pleasure to discuss my work. Yulan, without your advice on writing and your very short notice proofreading, this work would look quite a bit worse than it is now. I learned a lot while applying your tips and thank you a lot for all the help! Thank you Olivier and Alex for being there and giving feedback on my presentation practices, even though they took way longer than I initially promised. And of course, thanks to everyone I didn't mention by name for helping me out in any way or even just being there to listen.

Contents

Chapter 1

Introduction

Security through obscurity is rightfully generally considered bad practice in software development because it only hinders attackers but ultimately does not prevent them from extracting information [1]. The main topic of this work focuses on doing exactly this through software obfuscation techniques. Likewise, tamper-proofing, used to prevent attackers from changing secrets or code, only introduces an additional level of difficulty to attacks, which however makes their task harder. In contrast, protecting secrets through proper methods can actually prevent the extraction of information. This could be achieved by using analyzed and strong cryptographic algorithms or making use of only trusted infrastructure.

So why should we bother with obfuscation methods?

Real-world software, especially commercial products, often include secrets but cannot avoid shipping them in one way or another to the customer. In such cases, increasing the hurdle to tamper with the software is in the vendors and often the customer's good. Products that ship all their features and files with all releases but lock most of them behind specific licenses are a common example. Often, the development process does not allow the product to be split into different builds for each type of available license. Such *thick clients* are susceptible to attackers unlocking more features or extracting files [2]. If attackers or malicious competitors release a method of reenabling these features without a valid key, the company may lose revenue. Copyrighted content may be illegally extracted and redistributed, which would hurt existing or planned sales. For offline software, one worst-case scenario is that attackers figure out the license key generation and sell their own keys. While there are methods to prevent cases like the above from happening, most also restrict the software in one way or another. For example, validating licenses against servers owned by the software vendor requires internet access. If the features are all executed on the client's computer, attackers may find ways to circumvent the online verification completely. In the case of expensive software, trusted hardware dongles are also a popular approach, but may not be adequate for every set of customers and product price class. Even choosing to use such devices is not a foolproof solution. The hardware instead may be targeted and could be vulnerable. While no attack may be known at the point of shipping them, at a later point one might be discovered. Updating the hardware for each customer is often not feasible, or at least expensive. Another example would be shipping a product that in one way or another contains valuable information, like access keys, important user information, or propitiatory algorithms. In embedded devices, this is especially relevant, as they directly save and process user information on the physically accessible device. Be-

cause of all the constraints around them, they often can not connect to the Internet and rely on trusted infrastructure that way. Even for devices that are capable of internet connectivity, they often cannot keep constant connections or connect at any time.
In all of these cases, delaying the attackers and consequently increasing their expenses is in the vendors' favor. It reduces the number of attackers that have enough incentive and provides enough time to sell products or to update them.

Another useful method in combination with obfuscation and tamper-proofing is to diversify the critical parts of programs meant to be protected. This way, attacks on one specific version will not work on another. Especially on physical devices, flashing different versions on each device makes attacks less portable. Research has shown that this method increases the time required to break a system linearly to the number of unique versions [3].

A problem with most of the techniques used for obfuscation and tamper-proofing, especially when performing it in a diversifying way, is the difficulty predicting the resulting size and speed of the programs. For many methods, estimates can be made, but because of their often fundamentally non-deterministic nature, exact changes are only seen after completing the obfuscation process and are different between each run of the used tools. Because many methods interact with each other, the ordering of the techniques used matters a lot. This makes deriving precise combined output values just from individually processed results impossible. This may not be a problem in situations where resources are not constrained and the deviations of size and runtime overhead do not matter. However, on embedded devices where resources are limited, this can cause problems. If the obfuscated programs get close to the resource limits, there would be no way to know if each build fits on the device as no exact upper bound is known. Expansions to the input program would also be difficult because changes to it would cause unknown changes to the output, which may or may not fit anymore. Large programs might physically exceed the program memory of the device. Programs with high runtime overhead may instead miss deadlines, or interfere with other processes that need to be executed at regular intervals.

Contribution The contributions of this work are the design and implementation of a framework for obfuscation techniques that changes existing obfuscation methods to have a predictable overhead. The existing techniques of Mathematical Operation Encoding (5.1), Literal and Variable Encoding (5.2), Fake Dependencies (5.3), Bogus Control Flow (5.4), Control Flow Flattening (5.5) and Virtualization (5.6) are looked at specifically. Rules are developed for these established techniques to produce deterministic changes in binary size and execution speed while retaining their non-deterministic nature. In most cases, these rules require the output of each technique to be padded to have the same overhead as the worst-case overhead. These changed techniques are implemented in a framework that guarantees byte precision for the size and instruction count precision for the runtime overhead.

Outline This work is divided into 9 chapters. Following the introduction, chapter 2 provides information about what obfuscation is and what it is trying to achieve. Chapter 3 presents an overview of existing frameworks and the existing literature. Chapter 4 introduces the design of a framework for precise obfuscation techniques. Chapter 5 lists obfuscation techniques and methods of making them predictable. Chapter 6 displays some implementation details and what is possible to do thanks to the predictable behavior. Chapter 7 contains the evaluation of the framework implementation and how it compares to existing frameworks. Chapter 8 discusses possible future work and Chapter 9 concludes the results.

The implementation of the framework and techniques can be found at
https://github.com/Pusty/Obfuscat

Chapter 2

Background

To know how to effectively protect the valuable and sensitive information in programs, a definition of what exactly should be protected is needed. It is also important to look at who the potential attackers are and how they operate, as well as define the general concepts this work is based around.

2.1 Secrets

Almost all commercial software contains potentially sensitive information in the shipped product. This information can range from hardcoded cryptographic keys in client-side applications to media assets or proprietary code snippets. The importance of certain secrets depends on both the attacker's objectives and the context of the software. Some secrets may seem relevant at first glance but provide no value on further inspection, while others may seem irrelevant but are very precious to attackers. Common types of secrets include external application programming interfaces and their keys, fixed cryptographic keys, license keys and their authentification code, proprietary algorithms, and copyrighted material.

Externally used Application Programming Interfaces (APIs) might be valuable to attackers when these interfaces are not documented or otherwise available but are of little value if they are public. Even publicly available, APIs might still be a risk when keys are shared between releases, as attackers could use them for denial of service attacks when these APIs are rate limited. Especially risky would be keys to external interfaces that are paid per access or automatically upgrade payment plans after receiving a certain amount of requests. In those instances, disclosure of these keys could lead to high financial costs. In some situations, it may be possible to provide users with individual access keys, but for these to work, the user would first need to authenticate themself to the service. This initial authentication itself would then be a potential risk when used by non-authorized third-party programs. For applications with numerous copyrighted assets, the associated download keys may also be valuable to externally access them or obtain them before they would be regularly distributed.

The risks of using hardcoded cryptographic keys in the product provided to users can be quite severe as well. In cases of symmetric encryption, attackers having access to the keys compromises the whole system. For asymmetric encryption, this is only the case when the private keys are leaked. Depending on which keys the application itself uses, even external access to some of the keys could be a risk, as attackers might be able to encrypt arbitrary content to trigger bugs in the decoding process or impersonate the party whose keys they obtained. Examples with financial implications in the missed potential sales are mobile video games when through the disclosure of the keys the encrypted game states get modified in ways normally only possible through purchases. Protected media assets like streaming movies, or game assets, that could be extracted and illegally published or reused are another example. Even outside of monetary risk, known keys can provide risks, as seen in smart devices securing personal data where attackers might gain access to sensible individual information of customers.

Although diminishing, another popular example of secrets can be found in license keys for offline software. Compared to software that relies on an internet connection, it is not possible to verify the keys on trusted infrastructure. In those cases, there is also the risk of legitimate customers sharing their license keys. This is something that can be mitigated both legally and through revoking their access to the program in later versions though. The much greater secret is the license validation algorithm that is part of the product code to enable the product. If attackers gain full knowledge about its inner workings, they can generate their own licenses using so-called "keygens". These attacker-generated keys are indistinguishable from real ones if such an attack was not considered during the license system design. As such, it is hard to revoke the access of these illegitimate users without affecting paying customers, if their existence is even noticed. Software that does not have such restrictions and uses internet access to connect to trusted license verification servers does not have this problem. Instead, attackers might target the license checks directly. As long as a feature is already contained in the program the user has, it could be possible to remove the license checks around it. Thus, the secret would be locations in the program handling licensing.

Other sensitive information includes proprietary algorithms, copyrighted assets, specific implementation details, or personal customer information stored within a product.

2.2 Attacker and Attacks

To effectively protect these secrets, an understanding of the attackers' motivation and the type of attacks they might employ is required. Collberg et al. [1] provide three assumptions that provide us further information:

1. The attackers are creative humans trying to work around our defenses.
2. They have no time or access limit on the products.
3. All defenses only hold up for a certain amount of time.

The third point is especially relevant. The conviction that the attack is worth the attacker's time and energy is the only thing we can work against through obfuscation and tamper-proofing. All methods inherently try to discourage these attackers from their attack or make them spend as much time as possible on the defense itself instead of compromising the protected software. Because the attackers are inventive and do not follow set procedures, techniques that solely focus on being algorithmically hard to remove are not necessarily a problem for the attacker to manually workaround. Focusing on methods that are difficult to manually workaround may not be a problem for the attacker if it is simple for them to develop tools to algorithmically circumvent them.

Schrittwieser et al. propose a model for the specific objectives analysts may have [4]:

1. Finding the location of data
2. Finding the location of program functionality
3. Extraction of code fragments
4. Understanding the program

The table below categorizes the previously mentioned examples following this model:

Scenario	Objectives
Extraction of user data	Location of data
Extraction of keys	Location of data
Extraction of assets	Location of data
Extraction of keys at runtime	Location of program functionality
Bypass license checks	Location of program functionality Extraction of code fragments
Proprietary code reuse	Extraction of code fragments
Key generator development	Understanding the program
Stealing of proprietary algorithms	Understanding the program

Table 2.1: Categorization of previous examples following Schrittwieser et al.'s model

To achieve these goals attackers will either analyze and attack the product statically or dynamically. Static attacks are done on the program files themself without executing them, whereas dynamic attacks focus on the specific runtime behavior of a program.

Part of static analysis is to disassemble the code of a program to analyze the intended behavior from the compiled code. Control Flow Graphs and Call Graphs can also be generated from the disassembled code to analyze branch conditions and the overall flow of the program. For some targets, the attackers are also able to decompile the code. In this step, the tools will try to reconstruct high-level code by optimizing and inferring high-level artifacts out of the assembly to be as readable and easy to understand as possible. For Java programs, the decompilation step is especially relevant as these programs provide bytecode with significant metadata annotations, which makes high-level code recovery very precise [5]. The same goes for .NET programming languages like C#, F#, and Visual Basic.

In contrast to static analysis, dynamic analysis focuses on the runtime behavior of the target. This includes following the actual executed code and analyzing externally called functions, the register state, and memory at different points in the program. As this type of analysis requires the execution of the targetted products, the analyst either has to provide a compatible platform that can run the program or has to be able to emulate such a platform through software. This also makes the analysis susceptible to defense techniques that detect differences in the execution platform or the tracing of the program and react to it. The focus on runtime behavior allows gaining specific information about the program for the analyzed set of inputs. However, compared to static analysis, dynamic analysis does not provide global knowledge about all possible behaviors of the program.

Another type of analysis method is Symbolic Execution proposed by James King [6]. Instead of executing programs normally, the inputs are replaced with symbols representing arbitrary values. Instead of working on concrete values, the program now computes formulas on these symbolic inputs, and when encountering branches evaluates all possible destinations. This technique can be used to automatically analyze code [7] and gain knowledge about how code execution and inputs correlate. The automatic nature of this type of analysis makes it especially relevant for secrets where this connection is meant to be protected or encoded secrets can be recovered.

All these previously mentioned methods are passive because they do not change the actual program, but try to extract information out of it. Attackers may also want to perform active tampering attacks where they change parts of the product to let it perform actions it would not under normal circumstances. Trying to circumvent license checks by removing them or replacing parts of the product with malicious assets would fall under this category, and need additional care when trying to defend a program.

2.3 Obfuscation

Obfuscators formally defined by Barak et al. [8] are algorithms that transform arbitrary input programs into output programs that follow the following constraints:

1. The functionality of both the input and output programs must be the same for all inputs.
2. The output program may at most be polynomially slower than the input program.
3. The output program must fulfill the virtual black-box property, which means that analyzing the output program with full internal access should be as hard

as analyzing the input program as a black box.

While the same paper presents that it is impossible to develop algorithms fulfilling all of these constraints, this definition and the concept of Obfuscation still has practical application. While the black-box property cannot be met, obfuscation algorithms do make analysis more complex and subsequently, force the malicious analysts to spend more time and resources on their attacks on the obfuscated product. Depending on the methods used, obfuscation can help hide statically otherwise easily findable data, slow down static and dynamic analysis of code, and mask especially relevant properties of a program.

In addition, obfuscation techniques may allow for watermarking of binaries with certain version-specific magic values to identify the source of illegitimate shared releases of a piece of software.

2.4 Tamperproofing

Tamperproofing, or introducing tamper resistance into software, means implementing mechanisms into the product that prevent or hinder attackers from modifying it. These techniques may verify the correctness of code at runtime, repair modified sections or simply modify the executed functionality when detecting a tampered environment to mislead the attacker.

When these methods are implemented purely through software they may be easily spotted and removed. For this reason, it is important to additionally apply obfuscations or other protection techniques on the tamper-proofed code to be effective.

2.5 Diversifying

The idea behind diversifying software is to distribute multiple versions of the same software, which are functionally equivalent but differ in implementation, cryptographic keys, or the way they were compiled. This reduces the impact of bugs and makes analysis and exploitation less scalable, as each version is internally build up differently. These diversified binaries ideally are constructed such that the analysis of one version only provides information about how the other versions are functionally constructed but not how they work in detail. Through this approach, automated tooling which modifies, analyzes, or exploits all versions of the software would be required to be more complex, and the design of such tooling would be more time and cost-intensive.

N-Level obfuscation goes one step further and proposes to introduce functionally nonequivalent versions of a program to achieve these goals. Through this method, even the functionality of the source program is protected against the analysis of individual versions [3].

2.6 Deterministic Compilation

Deterministic compilation and reproducible builds are a process of ensuring that compilation output for a given input program is exactly the same when performed on any machine. By ensuring that the results are always the same, analysis on one build is ensured to be true for all builds. In such a build environment it is also possible to verify that neither the source code nor any compilation tool has not been tampered with. In reality building such a system is difficult because even at the compiler level, non-determinism may result from the build environment and compiler optimization techniques [9].

Chapter 3

Related Work

3.1 Obfuscation Techniques

In this work, existing obfuscation techniques are looked at and modified in a way where their overhead can be precisely calculated. Banescu et al. [2] and Collberg et al. [10] [11] provide overviews of a large amount of obfuscation techniques and classifications methods. On the theory aspect, Barak et al. work focus on developing a mathematical model for obfuscation [8]. The most simple techniques replace parts of expressions with equivalent ones. Hacker's Delight contains many tricks that can be helpful for obfuscation. Especially interesting are smart equivalences that are hard to identify [12]. However, because popular obfuscators make use of them, tools have been developed to specifically remove those tricks [13]. Because some algorithms are identifiable just by constant values appearing in them [14], techniques have been developed to encode them. Among them, Mixed-Boolean-Archimetics and their algorithmic construction look at efficient methods of producing many equivalences and encodings that use both logic and arithmetic operations [15]. To make program analysis more difficult, Collberg et al. propose opaque predicates to make it difficult to know which paths in a program are taken [10]. Another proposed method to do this is by flattening the control flow graph [16]. Cappaert et al. especially put in work to strengthen this method [17].
In the evaluation, the strength of the techniques is tested through symbolic execution [6], which can be used to automatically attack obfuscated code [7].

3.2 Software Obfuscators

Related to this work are both commercial and free obfuscation tools. In the realm of freely available tools, Tigress [1] and Obfuscator-LLVM [18] stand out.
Tigress is an academically free but closed source, source-to-source obfuscation tool that works on C and C++ code. Compared to this work, the possible input programs can be way more expressive. With its focuses on source-to-source obfuscation, it can be directly used or integrated with native code projects. With a lot of available transformations and configurations available, it is very versatile. In contrast to this work, Tigress does not give any guarantees on size or runtime overhead and causes unpredictability in that regard as it modifies the program on a source code level, which can lead to small changes in the source code to lead to massive changes in the native binaries or the other way around.
Obfuscator-LLVM is an academic open-source obfuscation tool that operates on the LLVM Intermediate Representation. While providing fewer features than Tigress,

[1]https://tigress.wtf/

Obfuscator-LLVM can obfuscate code from all language frontends that support the LLVM ecosystem. This makes it very flexible in the types of programs it accepts as input. Similar to this work the Intermediate Representation, internally used after being generated from the source code, is then compiled to native code by the respective backends. The LLVM ecosystem has the benefit of providing numerous output targets which are directly supported though. As the obfuscation techniques, themself get applied on the Intermediate Representation, the size and amount of executed instructions fluctuate less. The overall predictability of the modifications is still prone to later optimizations and other compiler passes that influence the resulting native code. An additional problem with the LLVM Intermediate Representation is the number of supported commands and types of control flow changes. Many different models are supported to account for the large variety of languages and platforms supported. Because of this, writing passes that work reliable on the whole internal format becomes complex, especially regarding precise estimating the overhead without losing functionality.

Chapter 4

Framework Design

The main goal of this work is to come up with a methodology to give precise guarantees about the size and runtime overhead of obfuscation techniques. To achieve this work proposes a framework design for the techniques and calculations. Within this framework, the concepts of program representation and obfuscation techniques will be defined as well as a method to generate native code.

4.1 Design Goals

The design should not limit the processor architectures it will be possible to be used with. To limit the scope of the actual implementation, this work will focus on a small subset of programs. The design should also be modular enough to allow for further expansions and not be too limited in expressiveness.

More precise, the main goals are as following:

Special Purpose Programs as Input: Only programs specially written as input for the framework are to be considered. The design will leave open how exactly these limited programs are parsed and processed to be usable by the framework. However, it will define an internal program representation and methods to manage it.

Executable Code as Output: To be as precise as possible about the predictions of the size and instructions executed, the framework will output native machine code usable by the targeted platform. The processing of the IR should be independent of the output target.

Flexible Framework for Obfuscation Techniques: The design should be modular and allow for new obfuscation techniques to be added to the framework. These techniques should have proper access and methods for the level of the program they are manipulating (Instruction, Basic Block, Method, Program). Each obfuscation method must precisely define functions to calculate the overhead it will cause for a given input program.

Ensure Precise Overhead Calculations for Size and Runtime: Each program holds a set of properties which represent it. The overhead formulas of the obfuscation techniques may only rely on these properties to calculate the changes in size and runtime overhead. These changes must also be a set of program properties which represent the obfuscated program. Formulas for these calculations must exist for both the size overhead and the runtime overhead. For the runtime overhead, the overhead calculated represents the overhead for a specific trace of the program. This way it can be used to calculate best-case, worst-case and average-case performance from the same formula by applying different program traces. The output formulas for the size and runtime must precisely calculate the size and instructions executed when provided to an output native code generator. These formulas themself must be independent of the targeted binary code.

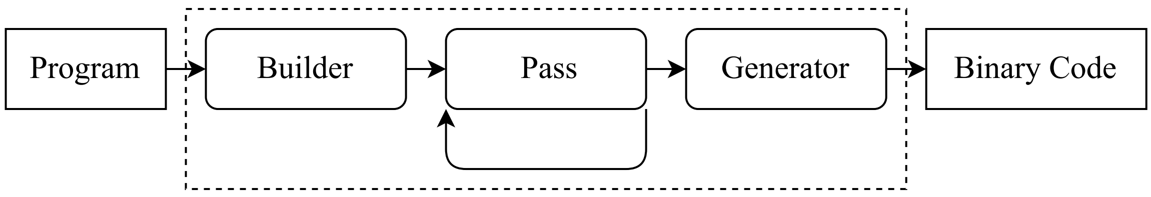

Figure 4.1: Obfuscation Pipeline: The Builder converts the Input Programs to the Intermediate Representation (IR). The IR is then passed to one or multiple obfuscation passes and finally compiled to native code by a Code Generator

4.2 Program Representation

Programs get represented through an internal Intermediate Representation (IR) within the framework. The converter to generate this IR from an input program will be called Builder. Obfuscation passes manipulate the IR of a program and output the modified IR representing the program with the obfuscation pass applied. The IR contains all required information to synthesize machine code from it, which the Code Generators are responsible for.
The IR is constructed as follows:
The smallest element is a Node which represents a single operation. A Node may require input Nodes for its operation. It may also have other immutable data attributes which stay constant during runtime, such as operation size or memory slots targeted. Possible types of Nodes are Constant Data, Load, Store, Arithmetic Operation, Logical Operation, Memory Allocation, and similar. It is important to note that some Nodes may not be used as input for another Node, because their evaluation yields a global action instead of a value, for example being Store Nodes.

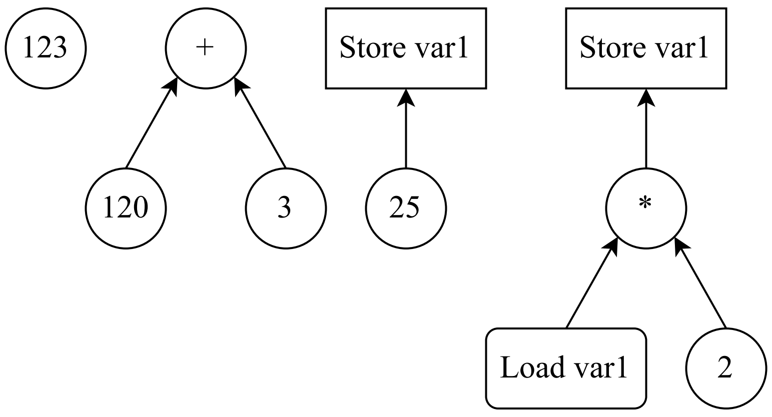

Figure 4.2: Examples of Nodes: A constant 123, the expression $120 + 3$, storing the value 25 into the variable var1, storing the result of var1 $\cdot$ 2 into the variable var1

The next larger unit is Basic Blocks. A Basic Block consists of a sequential list of Nodes. Nodes that are inputs for other Nodes may be part of the sequential list themself but are not required to be. A Node may only appear in a single Basic Block and may not connect to any Node outside of the Basic Block. When compiled or simulated, Nodes are executed exactly once per Basic Block execution. The order of execution is sequentially picking the Nodes in the list, recursively executing all the children of the current Node, and then proceeding with the next in the list. A Basic Block has to either declare another Basic Block as the default successor block executed after it or provide a function return value. Optionally, multiple conditions for other Basic Blocks to succeed the current may be present. If one of these conditions holds during the runtime, the Basic Block connected to that condition is executed next instead of the default successor. If no such checks exist or all conditions are evaluated as being false the required default successor is executed next, or the function the Basic Block is in returns with the return value provided.

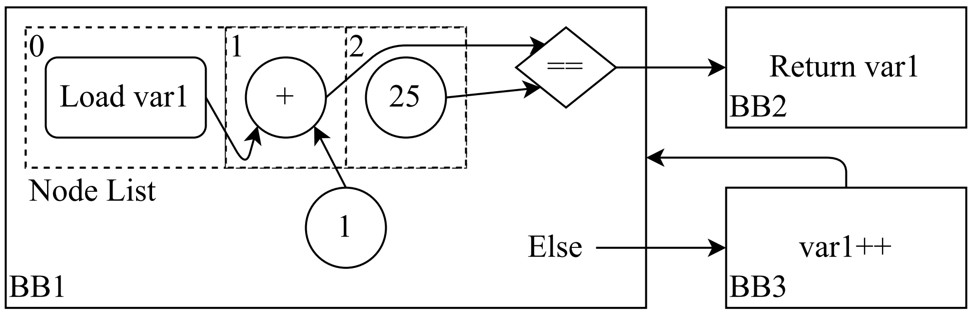

Figure 4.3: An example program with three Basic Blocks: The most left one shown with the inner details which illustrate the sequential Node List and how not all Nodes are required to appear in it. The equality comparison shows how conditional code flow can be modeled through Basic Blocks.

The largest unit in the IR is the Function. Functions have a list of Basic Blocks of which one is the entry point, which gets executed at the beginning. Additionally, functions need to declare how many parameters they take as input. The static data, used by Basic Blocks within the functions, is also required to be part of the structure.

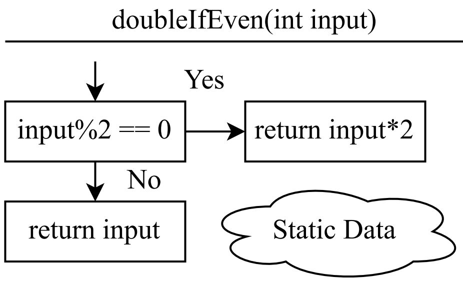

Figure 4.4: A diagram of a Function: When the input is even, return double the input, otherwise return the original input

No larger unit to express a program is required, as multiple functions can be merged into a single Function. This can be done by combining all the lists of Basic Blocks into a single list, adding one additional argument and Basic Block to selecting which set of Basic Blocks to execute, and using recursion to call subroutines.

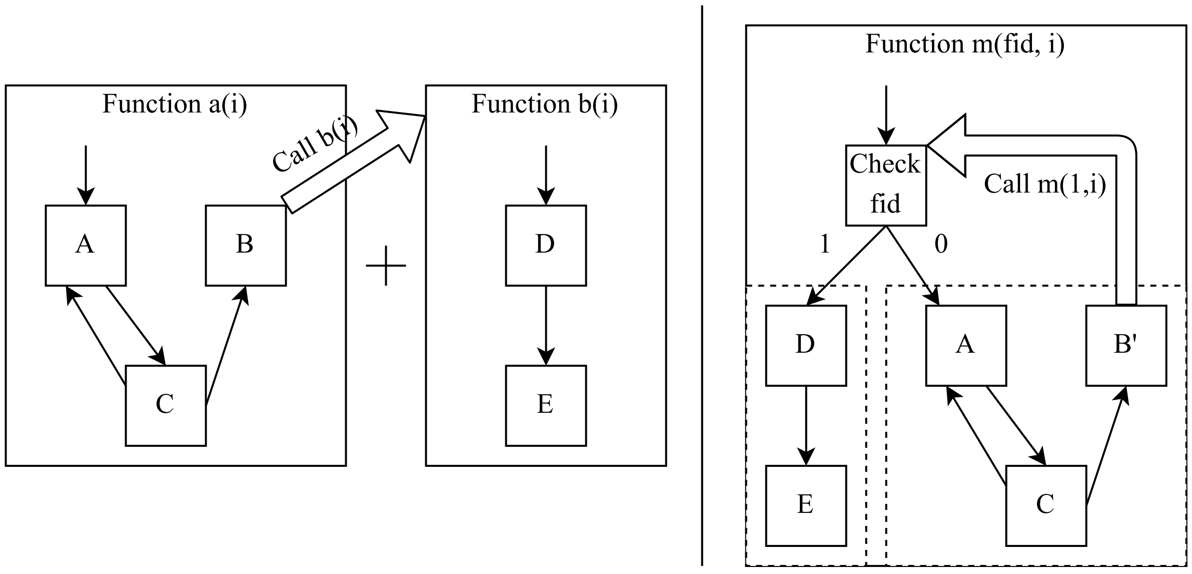

Figure 4.5: A diagram of multiple Functions being merged to one: The Functions a and b can be merged to m by adding an additional parameter and letting it decide which code is executed. Function calls are then changed to recursive calls with the appropriate function identifier added.

4.3 Obfuscation Pass

Generally, obfuscation techniques are not very predictable or deterministic by design. Deterministic output is often not even helpful as it minders the effectiveness of the methods used. But as predictability is the focus of this work, existing techniques have to be adjusted through restrictions. The obfuscation techniques in this design run as modular passes on the Intermediate Representation of a program. Every valid program must be able to be obfuscated. The resulting output IR does not have to be fully deterministic based on the input. Randomness within the predictability restrictions may be applied and is encouraged. But the applied pass must be precisely describable, both in size and runtime overhead, by static formulas that take variables representing properties as input that abstractly describe the IR.

The Intermediate Representation has a set of properties Props that describe the attributes of the IR that have an impact on the size and runtime behavior of actual Programs. As such a program in this definition is a map of properties $\mathit{Prop} \in \mathit{Props}$ to the number of occurrences $v \in \mathbb{N}$. The set of Programs supported by the framework is Progs. Every Program $\mathit{Prog} \in \mathit{Progs}$ has a function $\mathit{Prop}\colon \mathit{Progs} \to \mathbb{N}$ for each $\mathit{Prop} \in \mathit{Props}$ that represents the occurance of the property Prop in the Program Prog. The set of predictable Obfuscation Passes is Passes.

Every Obfuscation Pass $Pass \in Passes$ has a formula $PassF_{Prop} \colon Progs \to \mathbb{N}$ for each $Prop \in Props$ that describes the change the Obfuscation Pass $Pass$ has on the specific property $Prop$ given the input program. For all Obfuscation Passes $Pass \in Passes$ the function $Pass \colon Progs \to Progs$ is the actual applying of the Obfuscation Pass $Pass$ on a Program $Prog \in Progs$ to get the obfuscated Program $Prog' \in Progs$.

For every Obfuscation Pass $Pass \in Passes$ and every Program $Prog \in Progs$ the equasion $\{Prop(Pass(Prog)) \mid Prop \in Props\} =$
$\{PassF_{Prop}(Prog) \mid Prop \in Props\}$ must hold.

As an example:
For an IR with $Const_{value}$ and $Math_{operation}$ Nodes.
A Pass $O \in Passes$, that adds an addition that does not change the result to all constants.
A Program $pinProgs$ that consists out of $Math_{+}(Const_5, Const_7)$ = 5+7.
Let the set of properites $Props$ be $\{Const, Math\}$ describing the amount of Nodes of that type in a program.

Initially for p:

$Const(p) = 2$
$Math(p) = 1$

When applying O on p:
$O(p) = Math_{+}(Math_{+}(Const_5, Const_0), Math_{+}(Const_7, Const_0))$ which is a program equal to the expression $(5 + 0) + (7 + 0)$
so
$Const(O(p)) = 4$
$Math(O(p)) = 3$

The changes in both size and execution by O can be expressed through the following formulas, as the pass adds exactly one constant ($Const_0$) and one mathematical operation ($Math_{+}$) to the output for each constants in the input:
$OF_{Const}(p) = Const(p) + (Const(p) \cdot 1)$
$OF_{Math}(p) = Math(p) + (Const(p) \cdot 1)$

when applying the Formulas of O on the values of p:

$OF_{Const}(p) = Const(p) + (Const(p) \cdot 1) = 2 + (2 \cdot 1) = 4$
$OF_{Math}(p) = Math(p) + (Const(p) \cdot 1) = 1 + (2 \cdot 1) = 3$

as such

$OF_{Monst}(p) = Const(O(p))$
$OF_{Math}(p) = Math(O(p))$

which means:

$\{Prop(O(p)) \mid Prop \in Props\} = \{OF_{Prop}(p) \mid Prop \in Props\}$ holds,
so O is a predictable obfuscation technique.

For both size and runtime overhead, separate formulas need to be defined (though they may be the same for simple Node replacement techniques). For calculating the size overhead, only the property values of the input program are required. For calculating the runtime overhead, the property values of a specific execution trace need to be supplied. The predictability for the overhead is determined by the formulas giving exactly the same output property variables as the output IR for any program.

Through this design, the overhead can be calculated through external means without running a program through the actual passes, as the results of the formulas will match the obfuscation pass behavior. Additionally, it is possible to calculate the overhead of chained obfuscation passes by nesting the formulas. So assuming there was a second pass B the nested overhead of first running pass A and then B can be expressed as $Prop(B(A(p)) = BF_{Prop}(\{InnerProp \to AF_{InnerProp}(p) \mid InnerProp \in Props\})$ for a given property $Prop \in Props$. It is important to note that properties in these Formulas depend on each other, which means before the properties of BF can be calculated, all properties of AF need to be calculated first.

4.4 Code Generator

The task of the Code Generators is to take the IR from the Builder or the Obfuscation Passes and convert it into native binary code. For the predictability requirements, the generator needs to be able to calculate the exact native code size and amount of instruction from the properties of a program. This way, generators can predict the exact output from just the properties calculated using the formulas of obfuscation passes. The easiest way to achieve this is to ensure that the compiled size of a Node is independent of the context it is in, and for each type, the size and amount of instructions need to be precisely the same. The same must hold for all properties related to control flow and the appended static data.

So assuming we have the properties $Const$ and $Math$ and for a given Program p with:

$Const(p) = 2$
$Math(p) = 1$

with a Code Generator G in which all Nodes are exactly 16 bytes long and made out of 4 instructions:

$Size_G(p) = Const(p) \cdot 16 + Math(P) \cdot 16 = 2 \cdot 16 + 16 = 48$
$Instructions_G(p) = Const(p) \cdot 4 + Math(P) \cdot 4 = 2 \cdot 4 + 4 = 12$

Chapter 5

Obfuscation Techniques

There are multiple approaches to obfuscating operations on a basic block level. One of the initially more simple-looking approaches is to replace specific expressions with equivalent but more complex expressions. More complexity, in this case, means adding more terms to the replacement expressions. Ideally, the equivalence is not easily identifiable after the replacements, though judging the strength objectively is hard. Against manual analysis on instruction-level, this is achievable through mixing redundant sub-expressions into the original expression (e.g. out of $5 * 3$, the equivalent expression $1 * 5 * 3 + 0$ is constructible). Also, simple equivalent transformations, like replacing a single subtraction with the negation of a swapped argument subtraction-term, can be done to provide some hindrance and confuse the low-level analysis.
Techniques such as these fail to stop sophisticated analysis and tooling that works on higher-level representations, as these make heavy use of simplification and optimization techniques. This way, the redundant expressions, and simple equivalences get removed and reverted to the original expression or possibly something even more compact.
In contrast to these simpler operation obfuscations, control flow obfuscation targets the function level of the program to protect. They modify basic blocks directly but also may append or remove them. Most importantly, they change how the execution flows from block to block, or at the very least make both automated tooling and manual analysts think that it changed. Compared to the previously mentioned techniques, the following techniques greatly change how a program is interpreted by automated tooling. The downside is that the overhead increases rapidly when trying to provide robust obfuscation.

5.1 Mathematical Operation Encoding

A simple obfuscation approach is to encode mathematical operations by replacing terms with fixed non-obvious equivalent complex expressions. Using fixed replacement terms makes it simple to assess the overhead they will induce, and by padding them with redundant terms to the same size, the cost per replacement becomes constant. Because the replacement expression will be executed instead of the original term, the runtime overhead is equal to the size overhead as well and calculatable through $overhead_{oe} = operations_{math} \cdot replacementSize_{max}$. Not all replacement expressions consist only of mathematical terms but may introduce constants as well, depending on the granularity of the cost calculation. This may need to be considered in the padding.
The book Hacker's Delight [12] by Henry S. Warren provides a rich list of such possible replacements. In the chapter *Addition Combined with Logical Operations*,

multiple equivalences for arithmetic addition, subtraction, and negation as well as all logical operations are explicitly presented and proven.

$$x + y = x - (\neg y + 1)$$
$$x + y = (x \vee y) + (x \wedge y)$$
$$x + y = (x \oplus y) + (x \wedge y) \cdot 2$$

All these replacements have in common that the equivalent complex versions contain both logical and arithmetic operations. This is common among more sophisticated techniques, as it is generically difficult to undo mixed logical and arithmetic expressions without building a value table.
For multiplication equivalent replacement expressions that fulfill these mixing conditions can also be found:

$$x * y = (x \vee y) \cdot (x \wedge y) + (x \wedge \neg y) \cdot (y \wedge \neg x)$$
$$x * y = (x \vee y) \cdot (x \wedge y) + \neg(\neg x \vee y) \cdot (x \wedge \neg y)$$

The strength of these replacement expressions depends on the equivalences known to the analysis tools and the optimization techniques implemented. By using equivalences that mix logical and arithmetic terms, this is fulfilled to a certain degree. However, because all these expressions are fixed, these equivalences may be simplified through pattern-matching approaches. They can be directly undone by hardcoding the set of complex replacements used in the deobfuscation tool. Especially for the techniques presented in Hacker's Delight and other well-known expressions, such programs already exist [13], but are not integrated by default into the commonly used tools.

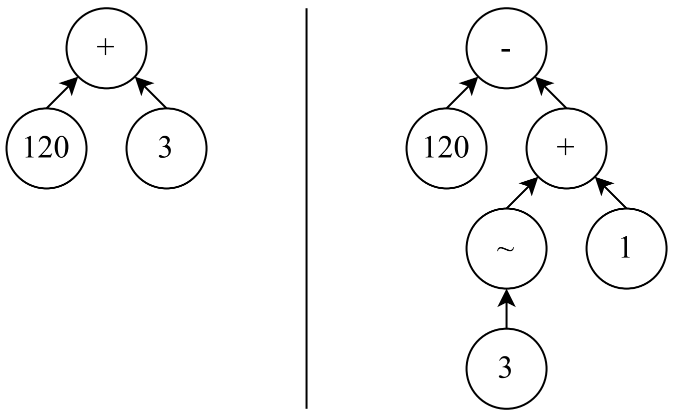

Figure 5.1: Example of Operation Encoding: Encoding of 120 + 3 to the equal equation $120 - (\neg 3 + 1)$

5.2 Encoding Literals and Variables

Literal values (constants and fixed strings) provide analysts with information about the code they are used in. Debug strings are the most obvious case, as they directly describe what a section of code is supposed to do. Information text strings expose which part of the code is responsible for which output. Constants might provide information on where loops start or end. Even worse, algorithms with known constants could be identified without having to be analyzed. Especially for cryptographic algorithms, this is a common analysis approach, and a multitude of tooling exists for malware analysis in particular [14]. By encoding these constants and strings, direct identification through simple magic values or fixed signatures becomes impossible. One method for encoding constants, proposed by Zhou et al., is to encode them using mixed-boolean-arithmetic and invertible polynomial functions [15]. Although less effective, even encoding the literals with a polynomial function and decoding them at runtime through the inverse function achieves the goal of not having them be present in a directly identifiable way. The same technique can be used to harden against dynamic analysis of local variables. By encoding all variables while they are stored, and only decoding them when they are needed, it is not possible to directly extract the full local state at a single moment of execution. To extract all local variable values, multiple points in the execution need to be hooked and pieced together. As the encoding and decoding of variables are independent of each other, each variable can have its unique function and inverse function pair. This makes automatic analysis of the encoding functions more difficult. For parameters or values that are accessible outside of the obfuscated function scope, it is important to either not encode and decode them, or encode the parameters at the beginning and decode them at the end of the function. This is required because the external code would otherwise be responsible for encoding and decoding the parameters without knowing the encoding functions. This method also provides some hindrance to static analysis. Not so much for literal values, as high-level analysis tools will statically deduce the original value by emulating the decode function. This is possible because they do not contain any variable inputs, which makes their actual value inferable by constant propagation. Encoding and decoding of local variables can contain non-statically deductible variables though. Thus, analysis tools may not be able to directly simplify these operations without specialized complex analysis.

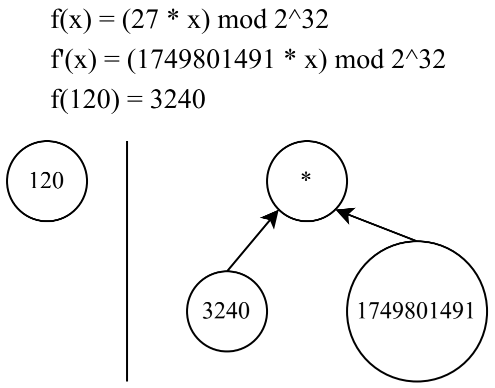

Figure 5.2: Example of Literal Encoding: Encoding 120 by using a reversible linear polynomial function

5.3 Fake Dependencies

The problem of high-level analysis tools being able to emulate code that does not make use of any variables is solvable through another obfuscation technique. This is done by replacing literal terms with expressions that make use of variables or ideally even parameters. The main objective of this method is to build expressions fulfilling this property without static analysis tools being able to deduce it. Injecting these fake dependencies helps to disrupt static analysis by breaking these optimizations and symbolic analysis by increasing the expression complexity. Additionally, taint analysis is hindered, as it becomes harder to deduce which values really depend on which variables. The previously mentioned method for encoding literals of Zhou et al. [15] does this through using both polynomial functions for encoding the literals and mixed-boolean-arithmetic predicates to inject difficult-to-identify fake dependencies to other variables. Hacker's Delight [12] also provides numerous different types of predicates that can be used to create connections to unrelated variables.
A very simple example for such a injection could be the following:

$$x = (x \wedge x) \wedge (x \vee fake)$$

These expressions are obviously equal as $x \wedge x = x$ and $x \wedge (x \vee fake) = x$. By including context from larger expressions or even other basic blocks these expressions can be additionally harderned.

5.4 Bogus Control Flow

Bogus Control Flow, as proposed in [11, Chapter 4.3.4], obfuscates the actual control flow of a program by inserting conditional branches that are never taken. These branches may point to other random basic blocks or fake blocks. The methods that generate fake basic blocks make them look similar to the basic blocks that get executed normally. Compared to the correct basic block, they introduce wrong behavior that is not straightforward to spot. When done correctly, this leads attackers to spend time statically analyzing the correct and fake basic blocks or force them to analyze the program dynamically. The strength of this type of obfuscation mainly relies on making it hard to deduct that these added conditional branches are never taken. To achieve this, opaque predicates are used. These predicates output the same truth value, independent of the input they receive. Collberg et al. propose multiple methods of building strong opaque predicates for which it is hard to automatically compute that they are opaque predicates [10].
For simplicity's sake, the approach looked at in this work does not add any fake basic block and uses trivial opaque predicates that are solvable within the scope of a basic block. When calculating the overhead of this implementation, the runtime and size overhead is the same, as the predicate is always evaluated and the appended branch is never taken. The overhead can be modeled as $overhead_{bcf} = (operations_{opaquepredicate} + conditionalBranch) \cdot unconditionalBranches$ where all possible opaque predicates must have the same amount of operations. This is achievable by padding the shorter predicates to the length of the most complex one. For

an implementation that adds fake basic blocks, the runtime overhead would need to be adjusted to exclude the never-taken jumps from the calculations.
Notice that while there are more complex methods (e.g. two-way opaque predicates where both branches may be taken) opaque predicates that rely on constant truth values may be easily identified through manual analysis or automated dynamic analysis.

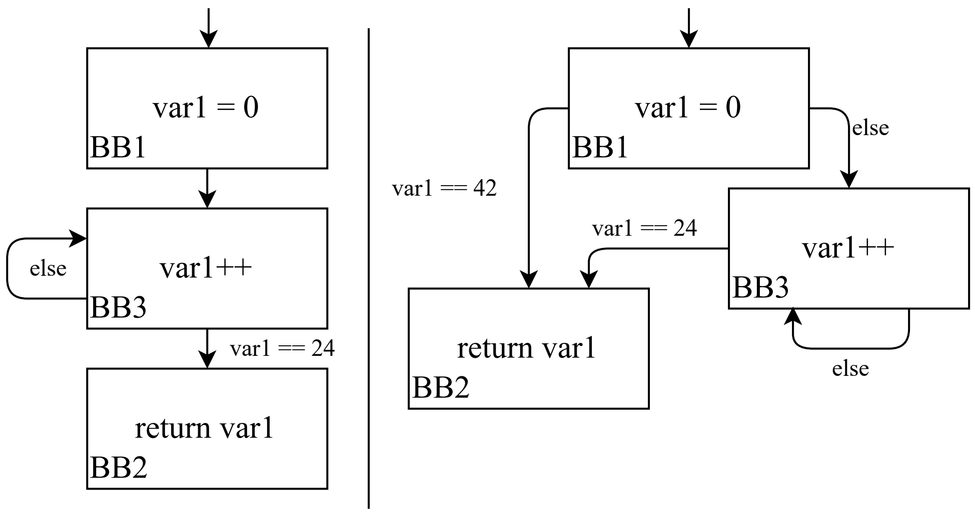

Figure 5.3: Example of Bogus Control Flow: Bogus Control Flow adding a very simple impossible conditional jump.

5.5 Control Flow Flattening

Similar to Bogus Control Flow Obfuscation, Control Flow Flattening aims to obfuscate the control flow of a program [16]. The control flow is flattened by appending a new basic block as a dispatcher and having all obfuscated basic blocks jump to it. The new appended basic block is responsible for choosing which basic block is executed next. The idea behind this method is that all obfuscated basic blocks originate from the same source and point back to it, which means no information can be inferred from just the control flow graph.
One possible implementation is to have a global ‘next block‘ variable. The dispatcher uses this variable to determine the next basic block. At the end of each obfuscated basic block, the block sets the variable to the successor block. For each block that previously conditionally branched, the ‘next block‘ variable is set conditionally instead. Using multiple basic blocks to implement the conditional ‘next block‘ assignment would expose the original control flow within the obfuscated control flow graph. As that would be counterproductive, formulas that achieve the conditional behavior without branching should be used instead. Such formulas can be found in [12] for all comparison operations. Another method of strengthening this obfuscation method [17] is to make the changes to the ‘next block‘ variable relative to the current basic blocks value.
The overhead for this technique must be divided in both size and runtime overhead because the dispatcher block will only be counted once towards the program size

but will be executed for each basic block. Additionally, to initialize the ‘next block‘ variable to jump to the first obfuscated basic block an additional entry point basic block needs to be added.
As such
$sizeOverhead_{flatten} = entryBlock + dispatcherBlock + nextBlockEpilog \cdot basicBlocks - removedConditionalJumps \cdot conditionalBasicBlocks$
and
$runtimeOverhead_{flatten} = entryBlock + dispatcher \cdot executedBasicBlocks + nextBlockEpilog \cdot executedBasicBlocks - removedConditionalJumps \cdot executedConditionalBasicBlocks$

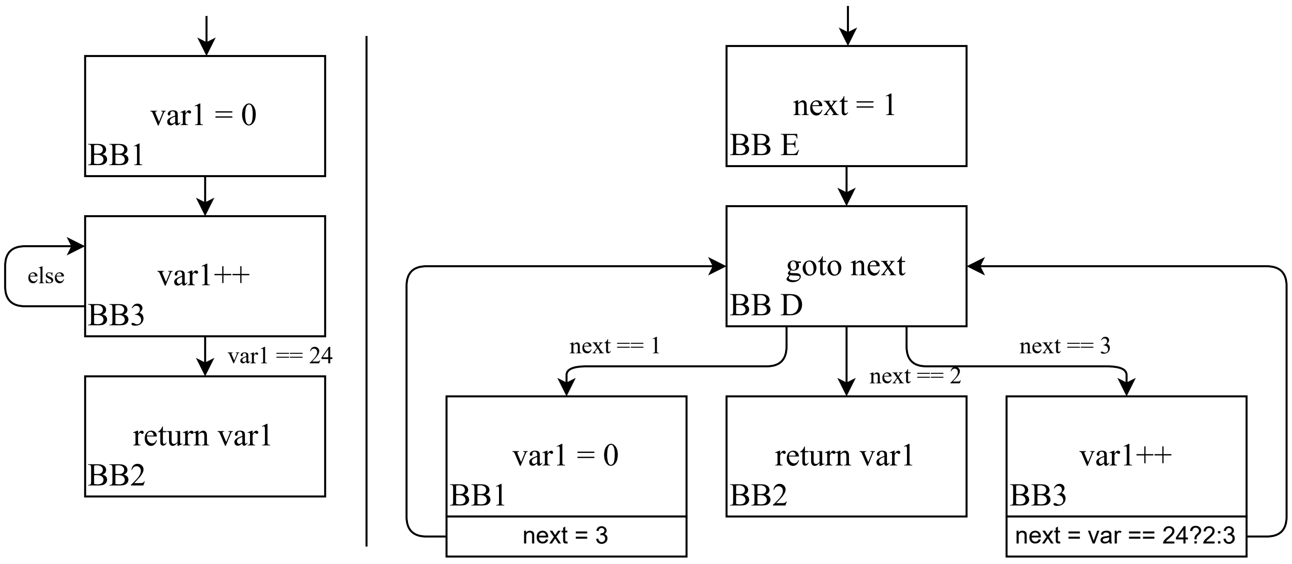

Figure 5.4: Example of Control Flow Flattening: Flattening the example program

5.6 Virtualization

Virtualization Obfuscation helps to protect code by compiling it to a custom architecture instead of the native target. This custom architecture code is then packed together with an interpreter that decodes and executes it natively. By doing this, the native code only contains the interpreter for the custom architecture, while the protected program itself can not be analyzed directly with tooling designed for the native platform. Static analysis is then only possible if tools are written or adjusted to work with the custom architecture [19] . A drawback of this approach is that reusing the same or similar custom architecture makes it possible for attackers to also reuse their tooling.
Calculating the overhead of this type of obfuscation is complicated and dependent on the custom architectural design. A very simple design would map each node from the intermediate representation to exactly one custom architecture instruction. Compared to the previous mentioned techniques, these techniques do not produce a clear overhead but change the size and runtime output completely: $size_{simplevm} = (entryPoint + dispatcher + \sum_{opcode}^{opcodes} handler_{opcode}) + size_{simplevm-compiler}(protectedCode)$.
As the compiler for the custom architecture must follow the previously defined design

rules for a compilation backend, the size of the compiled protected code is predictable as well. The runtime behavior can be expressed as $runtime_{simplevm} = entryPoint + dispatcher \cdot executedInstructions_{all} + \sum_{opcode}^{opcodes} handler_{opcode} \cdot executedInstructions_{opcode}$. The amount of executed instructions of the protected program will always be more than the original because all original instructions now require executing the instruction dispatcher and their instruction handlers. However, the size of the protected program is not guaranteed to be larger if the size of compiled nodes is smaller for the custom architecture than for the native platform. In those cases, as the overhead of the interpreter is fixed, large programs might be smaller in the protected format than their unprotected native counterpart.
Another feature introduced by access to virtualization obfuscation is that by providing a portable implementation for the custom architecture interpreter (e.g. written in C), it is possible to use programs without a native compiler. As it is hard to make predictable formulas out of these portable implementations, the resulting executed code may no longer be predictable in size and runtime beyond the behavior of the custom architecture code.

Chapter 6

Implementation

Based on the design goals, the actual obfuscation framework Obfuscat [1] was developed. It takes limited Java programs as input and obfuscates them using the previously introduced techniques. The main architecture it targets is ARM Thumb2 machine code, which can be directly called from C/C++ code on that platform. It is implemented in Java and has no external dependencies except a Java Runtime. The correctness and precision of the obfuscation techniques are internally tested for the IR and the native targets. For verifying the ARM Thumb2 target, the Unicorn Framework [2] was used, which is built on top of QEMU [3]. For demonstration purposes, a web port using Google's GWT [4] was made as well, which uses MxGraph [5] to render control flow graphs.

6.1 Intermediate Representation Generation

The design itself explicitly did not specify the methods to generate the Intermediate Representation of programs, but to practically use the framework, hand-writing the IR or generating it from templates became impractical outside of very simple programs. To support more complex programs, a method to generate the IR from an expressive programming language is preferable. Designing custom tooling to parse a new programming language introduces a development overhead, though. Additionally, it would require users to learn yet another domain-specific language and the tooling around it before being able to use it. Instead, the Obfuscat framework makes use of the Java Class Format and its Bytecode. The Java Bytecode is what the Java Virtual Machine uses as the compiled format of programs that run on it [20]. Many languages, most prominently Java itself, compile to Java Bytecode. This makes it possible to use the already existing tooling to develop and compile code. As the Obfuscat Intermediate Representation is noticeably simpler and provides fewer features than the full Java Virtual Machine specifies, only a subset of the format is supported. This subset in the context of Java means no Synchronization, Floating Points, Longs, Objects, or Exception Handling. The supported instructions of the Java Bytecode get parsed and translated to the Obfuscat IR by the framework. For efficiency reasons, the format the Java Bytecode uses to store static data arrays is problematic. When the Java Runtime loads a class for the first time, the static data array is created and filled with data one by one by the code. As this causes a high overhead within the Obfuscat framework, the class initialization function is

[1]https://github.com/Pusty/Obfuscat
[2]https://www.unicorn-engine.org/
[3]https://www.qemu.org/
[4]http://www.gwtproject.org/
[5]https://jgraph.github.io/mxgraph/

instead parsed and emulated. Through this, the resulting static data after the class initialization is read and appended as read-only data to the Obfuscat IR.

6.2 Intermediate Representation Implementation

Following the design specifications of the Intermediate Representation, the following types of Nodes are implemented:

Constant Constant Nodes return a constant value as their result and take no input. The value they represent may also be a pointer to static data.

Variable Load Load Nodes load a value from a variable and return it.

Variable Store Store Nodes store a value to a variable. They may not be the input of any other Node.

Array Load Array Load Nodes load a value from an array at a variable index and return the loaded value.

Array Store Array Store Nodes store a value into an array at a variable index. They may not be the input of any other Node.

Mathematical Operation Mathematical Operation Nodes represent a single mathematical operation. They take the operants as inputs and return the result of it.

Allocation Memory Allocation Nodes represent a dynamic memory allocation and are used for arrays that are created at runtime. They return a reference to the created array.

Custom Nodes Custom Nodes are made for platform-specific behavior but are also used to implement calling functions.

Whenever a store operation directly or indirectly has a load operation as an input node, the load operation needs to be executed within the Basic Block as soon as the desired value becomes valid. This is required because expressions like $i++$ translate to $var\,r = i; i = i + 1; return\,r$. As the store operation cannot be the input of another node, anything that takes the return result of $i++$ as input would connect to the load operation before the store operation. As nodes are executed sequentially in a basic block, the store operation would be executed before the load operation, which causes the load operation to return the wrong value.

6.3 Native Code Generation

The actual native backend implemented in Obfuscat generates ARM Thumb2 code. Traditionally the main target of obfuscation tooling would be x86 for wide usage in desktop and server computers. Instead, ARM was chosen as the main target for Obfuscat, because of the very prominent and uprising usage of the architecture in mobile and embedded devices and more recently tablet and laptop processors. This allows for a wide range of supported applications for fields where not as much tooling exists. In contrast to ARM mode code, Thumb2 is a more compressed alternate instruction set supported on ARM processors. Compared to the normal mode in which instructions are always 4 bytes in size, Thumb2 instructions may be compressed 2 bytes or 4 byte long instructions. This allows for higher code density in general. Additionally, it enables using it on more limited ARM processors, designed for microcontroller use, which only support the compressed format. Generally, the generated code only uses the base extension instruction set, which exists on every process since ARMv7. For division, the SDIV instruction is required, which especially for older ARMv7-A processors might not be present. This implementation decision was made compared to using software helper functions for integer division because all microcontroller-oriented processors are guaranteed to support this instruction as well most others.

According to the design, every node compiles to the same amount of bytes and instructions independent of context. The implementation simplifies this even more and compiles every node or other program property (like jumps) to exactly 16 bytes made out of 6 instructions. In most cases, this means that compiled nodes contain code that acts as padding and serves no purpose except aligning them in size and instruction count. Besides the obvious downside of this increasing the overhead, the padding instructions may make the manual analysis of the native code more complex. The resulting code synthesis process is very simple and modular.

With these constraints calculating the actual compiled size and instruction overhead, becomes were manageable as well:

$$Size_G(p) = (FunctionPrologue + Nodes(p) + BasicBlocks(p) + ConditionalJumps(p)) \cdot 16 + StaticData(p)$$

Calculating the amount of executed instructions is done by multiplying the amount of executed nodes and other program properties by 6.

The code generation itself is separated into the following steps:

Sequentializing the Nodes As the basic blocks consist out of abstract semantic graphs of nodes and the native code is inherently sequential, the nodes need to be ordered in the sequence they are supposed to be executed.

Conncecting Dependent Nodes The implemented compilation model assumes that each compiled node reads the values it requires from the stack, processes them within registers, and writes the result at another stack position.

Optimizing Stack Slots If each node has an individual position on the stack for the output value, the number of stack spaces required scales directly with the number of nodes. This is a problem for the more complex obfuscation techniques because the memory on the stack that is required to be addressed grows higher than what is easily referenced by individual native instructions. To reduce the complexity of the node-to-native-code translation, spaces on the stack are reused after the return value of a node is no longer required.

Node to Native Code Translation The now ordered nodes are then translated to native code one by one. As the context of the program is irrelevant for the translation of individual nodes, the translation works over simple templates where only parameters of instructions are adjusted.

Linking Together As the last step, the basic blocks are linked together through unconditional and conditional jumps. Additionally, a prologue to process function arguments and the optional static data are added.

6.4 Configurable Obfuscation

A feature that arises by making the prediction formulas for the obfuscation techniques so simplistic, is that they become solvable through automated means. This can be used to automatically choose optimal obfuscation techniques or adjust settings based on restrictions and criteria.

As a proof of this concept, small scripts have been implemented that demonstrate this using the Z3 theorem prover from Microsoft [6]. At first, the obfuscation framework is used to determine the properties of the original unobfuscated program. Based on these properties, formulas representing the individual obfuscation technique overheads are given to the Z3, in addition to the formula to calculate the native code size and runtime overhead from them. These formulas are then chained together in the following way:

For the Obfuscation Techniques $A, B \in \mathit{Passes}$ and with $\mathit{Props} = \{X, Y, Z\}$:

For a Code Generator G the size of the whole program can be calculated through $\mathit{Size}_G(p) = X(p) + Y(p) + Z(p)$. Now by introducing $c_O, c_A, c_B \in \{0, 1\}$ where $\sum(c_O, c_A, c_B) = 1$ must hold, the following function $\mathit{Obf}_{O|A|B}(p)\colon \mathit{Progs} \to \mathit{Progs}$ can be build: $\mathit{Obf}_{O|A|B}(p) = \{X \to c_O \cdot X(p) + c_A \cdot AF_X(p) + c_B \cdot BF_X(p), Y \to c_O \cdot Y(p) + c_A \cdot AF_Y(p) + c_B \cdot BF_Y(p), Z \to c_O \cdot Z(p) + c_A \cdot AF_Z(p) + c_B \cdot BF_Z(p)\}$. Depending on c_0, c_A and c_B this function does apply either A, B or no obfuscation on the input Program p. Now it is possible to use Z3 to solve and optimize for the techniques to apply to the program with given constraints. This is done by solving for c_O, c_A, c_B, and as only one of these can hold the value 1 at the same time, the obfuscation technique represented by the variable can be applied. As the solver does a lot of optimization internally, it is feasible to stack these formulas recursively, so $\mathit{Obf}_{O|A|B_{O|A|B}}$ with $c_{O_0}, c_{A_0}, c_{B_0}, c_{O_1}, c_{A_1}, c_{B_1}$ etc. can be calculated as well. In practice a constraint like $\mathit{Size}_G(\mathit{Obf}_{O|A|B}(p)) < 1000$ could be used to calculate all obfuscation techniques that do not increase the overall size of the program to over 999 bytes. Similarly, constraints, to limit under which conditions techniques can be applied, can be added. This could be used to limit certain methods to only be used last, first, or following another technique.

The end result is a solver program that takes a target program and the maximum output size as input and outputs an obfuscated program that fulfills the size restriction.

6.5 Browser Port and Control Flow Graph Renderer

The implementation of Obfuscat allowed it to be compiled to JavaScript using Google's GWT. In combination with MxGraph, this makes it possible to use the framework to generate control flow graphs of programs before and after obfuscation to demonstrate the effect of different techniques (6.1).

[6] https://github.com/Z3Prover/z3

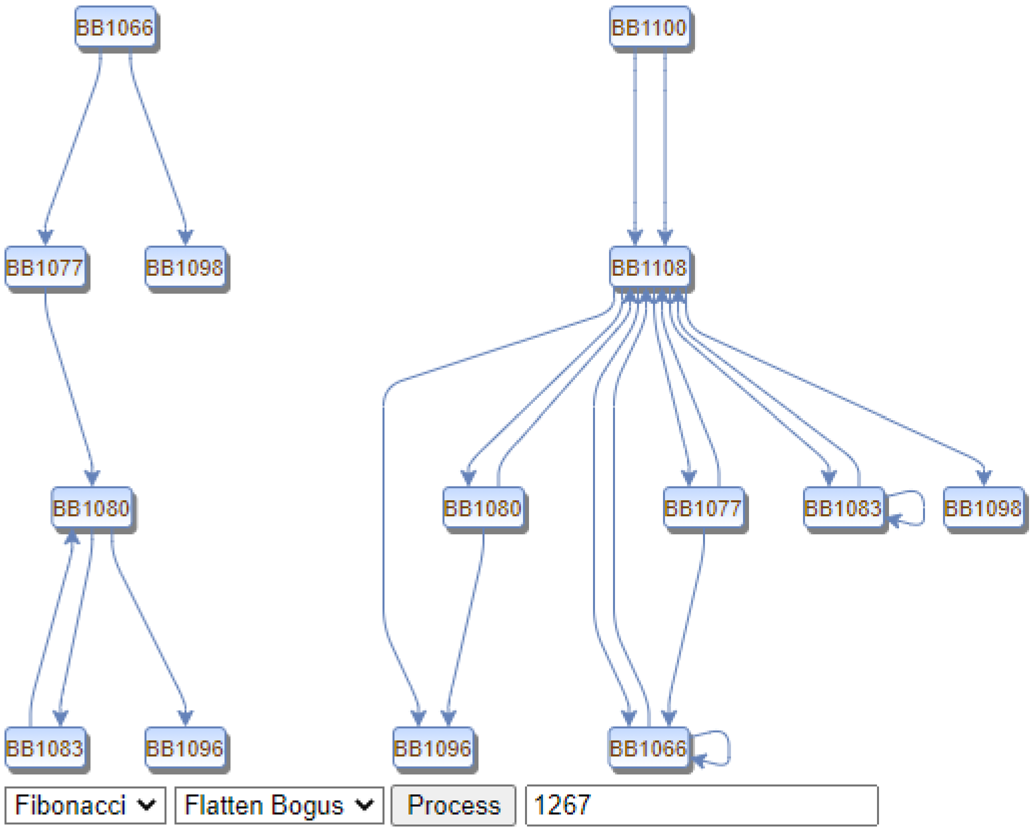

Figure 6.1: Visualization using Obfuscat: Control Flow Graph of Fibonacci Number Generation before and after applying Control Flow Flattening and Bogus Control Flow

Chapter 7
Results

To see how the obfuscation techniques implemented for Obfuscat perform, they are compared against each other for different test programs as input. One of the base assumptions for this work was that the other tools are not predictable and vary in size and runtime overhead between compilations. Tigress and Obfuscator LLVM are used as comparisons to see how Obfuscat performs in size overhead, runtime overhead, and precision. At last, the techniques compared before are roughly individually tested in their strength against symbolic execution.

7.1 Technique Overhead Evaluation

The first evaluation topic was to compare the techniques implemented in Obfuscat with each other. Implementations of the algorithms CRC32 with a small code footprint, RC4 with a medium amount of code, and SHA1 with a large code size were used for testing. All measurements were done on the native code blobs directly generated by the framework. To measure the number of executed instructions, the Unicorn Framework was used to emulate an ARM environment. The baseline to which all measurements are relative is the default output of Obfuscat without any obfuscations applied. To measure how the runtime overhead scales with input length, the tests were run with randomized input of lengths 128, 256,512, and 1024. All implementations of the previously mentioned techniques were tested. Because the overheads are deterministic the measurements represent exact values that are consistent.

Table 7.1: Obfuscat CRC32 Measurements: Comparison of Obfuscat Techniques on different programs and input sizes

		Input Length in Bytes			
Method	Size	128	256	512	1024
Default	**1 072**	**141 774**	**284 484**	**569 004**	**1 136 190**
OperationEncode	264%	279%	277%	278%	279%
LiteralEncode	279%	314%	312%	312%	312%
VariableEncode	243%	233%	232%	232%	232%
FakeDependency	212%	232%	233%	232%	233%
Flatten	234%	351%	350%	350%	350%
Bogus	154%	159%	158%	158%	158%
Virtualize	1548%	9456%	9422%	9425%	9432%

For CRC32 7.1 the first observation is that except Virtualization, all results are within 154% (Bogus Control Flow) and 279% (Literal Encoding) size relative to the output with no obfuscations. After applying Virtualization Obfuscation, the size goes up to 1548% of the default. The runtime overhead relative to the default varies very little relative to each other and does not seem to scale with input size. For most techniques, the relative size changes seem to be very close to the relative runtime changes. For Literal Encode (+10.58%) and Control Flow Flattening (+33.14%), it deviates noticeably though. Virtualization is a special case where the runtime overhead does not match the size overhead at all, and the runtime overhead ends up at 9432% of the default output for an input of 1024 bytes.

Table 7.2: Obfuscat RC4 Measurements: Comparison of Obfuscat Techniques on different programs and input sizes

		Input Length in Bytes			
Method	Size	128	256	512	1024
Default	**3 808**	**175 824**	**246 480**	**387 792**	**670 416**
OperationEncode	213%	262%	262%	262%	263%
LiteralEncode	278%	213%	210%	208%	207%
VariableEncode	253%	280%	282%	284%	285%
FakeDependency	211%	170%	169%	168%	167%
Flatten	152%	175%	171%	168%	165%
Bogus	120%	116%	115%	114%	114%
Virtualize	466%	9323%	9334%	9344%	9352%

For RC4 7.2 the relative size changes of Bogus Control Flow are even lower than as for CRC32 (120%). Control Flow Flattening (152% compared to 234%), Operation Encoding (213% compared to 264%), and especially Virtualization (466% compared to 1548%) have a noticeable decrease in size as well. Only Variable Encoding went up to 253% from the 243% it had for CRC32. The observation that the changes in runtime overhead are small still holds, but the variety increased. Again, the relative instructions executed do not seem to scale with input size, except for Virtualization, where the overhead increases a little bit each time. In contrast to CRC32, the relative runtime measurements are quite different from the relative size values. Only the values for Bogus Control Flow seem similar. The runtime overhead falls between 1.14 and 2.85 times the default measurements values, except for Virtualization where the highest value is 9352% of the default value.

Table 7.3: Obfuscat SHA1 Measurements: Comparison of Obfuscat Techniques on different programs and input sizes

		Input Length in Bytes			
Method	Size	128	256	512	1024
Default	**63 024**	**165 702**	**278 274**	**503 418**	**953 706**
OperationEncode	300%	333%	332%	331%	331%
LiteralEncode	306%	251%	249%	248%	247%
VariableEncode	223%	237%	239%	240%	240%
FakeDependency	229%	194%	193%	192%	192%
Flatten	104%	147%	148%	148%	148%
Bogus	101%	104%	104%	105%	105%
Virtualize	63%	9118%	9122%	9125%	9126%

For SHA1 (7.3), the relative size change of Bogus Control Flow again decreased to 101%, so only a 1% size overhead was added. The changes of Control Flow Flattening goes down to 104%, Variable Encode to 223%, and for Virtualization to 63% compared to the non-obfuscated output. This means that applying the Virtualization decreased the file size by 37%. For the other methods, the overhead increased, with Literal Encoding being the highest with a relative change of 306%. Again, the relative runtime measurements are close together independent of the input length, only Virtualization increased by small margins relative to the input length. Bogus Control Flow with 105% and Operation Encoding with 333% were the lowest and highest changes respectively, excluding Virtualization which increased the amount of executed instructions by 9126% for an input length of 1024 bytes.

Virtualization overall had the most interesting overhead results. For small programs like CRC32, the size grew to 1548% of the original, whereas for the medium-sized RC4, it was already only 466% , and for SHA1 it even decreases the size to 63%. Because Virtualization replaces the original program with a Virtual Machine, the static size of this interpreter is added to the size of the previous program. But because the code generated for the Simple Virtual Machine is denser than the native Thumb2 code generation, the actual program size is smaller than the default. So when $Nodes_{program} * 16 < Nodes_{program} * 6 + Nodes_{interpreter} * 16$ holds, the program will be smaller after using Virtualization obfuscation. The runtime overhead will be over 9000% , though which is way more than for all other techniques. The size changes for all other techniques fell between 101% (SHA1 Bogus) and 306% (SHA1 Literal Encode), and the runtime changes between 104% (SHA1 Bogus) and 351% (CRC32 Flatten).

7.2 Comparison against Tigress and Obfuscator-LLVM

To see how Obfuscat compares to other open-source obfuscation tools, Tigress and Obfuscator-LLVM (OLLVM) were run on the same input programs using comparable

obfuscation settings. For Obfuscat the input programs were written in Java and compiled to ARM Thumb2 native code blobs which were linked together with C code and compiled with GCC for Linux. For Tigress and Obfuscator-LLVM, the input programs were translated to C and compiled for ARMv8-A. As a baseline, the translated C programs were also directly compiled with GCC and tested. The tested algorithms were a CRC32 implementation which does not use a lookup table so it has to iterate over each input byte individually, default RC4, and a SHA1 implementation. The runtime measurements were taken on randomized inputs with 8, 64, 128, 256, 512, and 1024 byte lengths. The size measurements represent the size of the compiled runnable programs. For the runtime measurements, the amount of executed instructions were recorded using QEMUs userland emulation. The Cortex-A15 was the targeted CPU configuration. The runtime measurements do not just contain the program itself, but also the userland part of the initial loading process, as well as the LibC startup code. As Tigress and Obfuscator-LLVM do not compile predictable in size and execution overhead, and to show that Obfuscat fulfills these properties, for each technique of each tool 100 binaries were compiled and each run 10 times. The tested techniques were Bogus Control Flow (BCF), Control Flow Flattening (FLA), and Operation Encoding (SUB). For Obfuscat, the output without any techniques applied was also evaluated.
Starting with the results on the output file size:

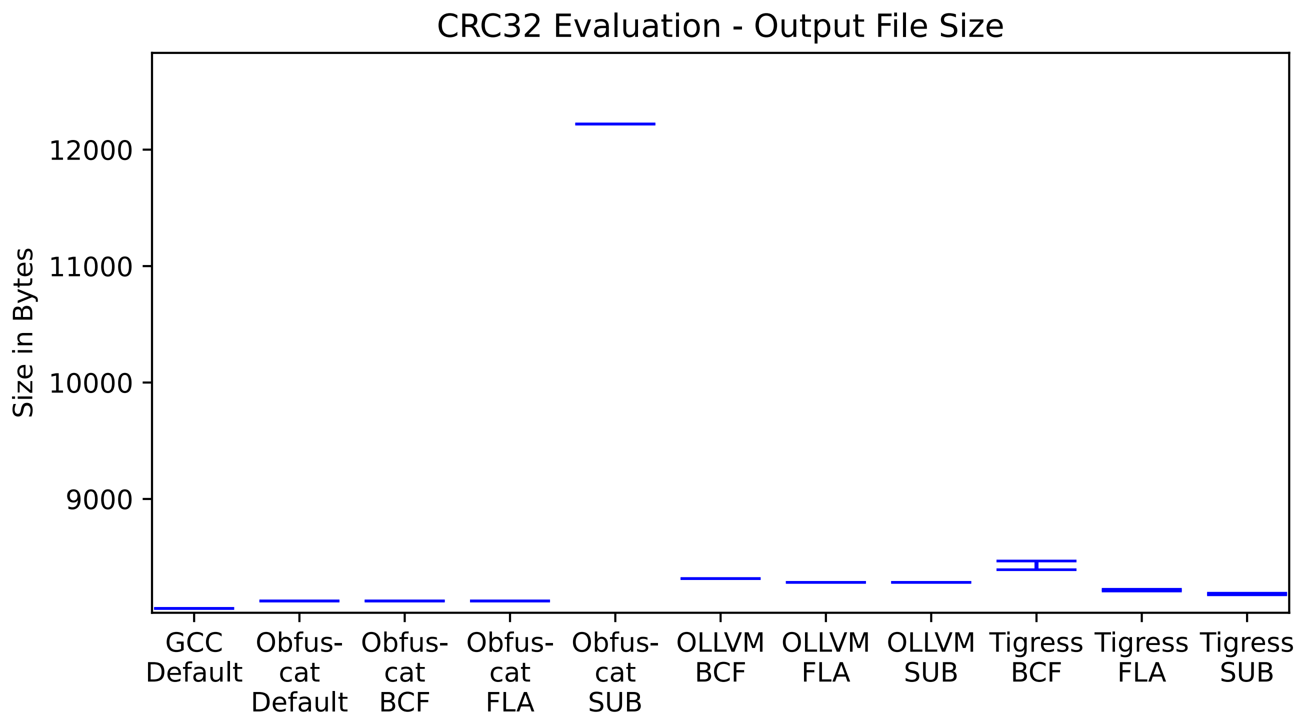

Figure 7.1: CRC32 File Size Comparison

For CRC32, the GCC Baseline was constant at 8 060 bytes. Almost all results were between 8 000 and 8 500 bytes. This is probably caused by the implementation code being very small and the binaries mostly being padding and file format overhead.

Only the Obfuscat Operation Encoding made a noticeable change, with the output files being always exactly 12 220 bytes in size. It is also noticeable in 7.1 that for all Tigress outputs, the file sizes varied by a few bytes between compilations.

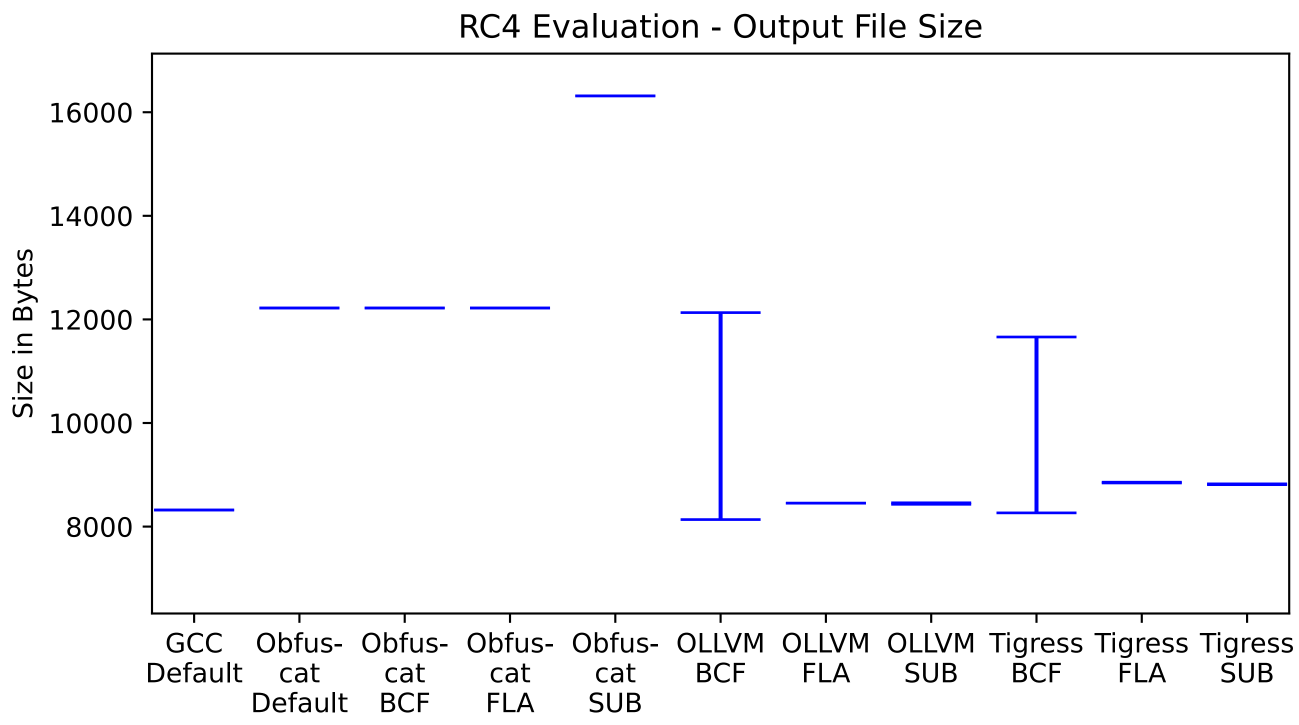

Figure 7.2: RC4 File Size Comparison

For RC4, as seen in 7.2, the GCC Baseline was constant at 8 320 bytes. Obfuscat's techniques yielded 12 220 bytes for everything except Operation Encoding, which produced 16 316 bytes. For OLLVM and Tigress, large variations were noticeable at their Bogus Control Flow techniques. OLLVM's BCF averaged at 10 133.44 bytes with a standard deviation of 1 996.824 bytes. Tigress BCF averaged at 9 962.6 bytes with a standard deviation of 1 698.0 bytes. The other techniques of the tools scored close to the GCC Baseline between 8 400 and 8 900 bytes. OLLVM's and Tigress SUB and Tigress FLA had a few bytes variation between compilations.

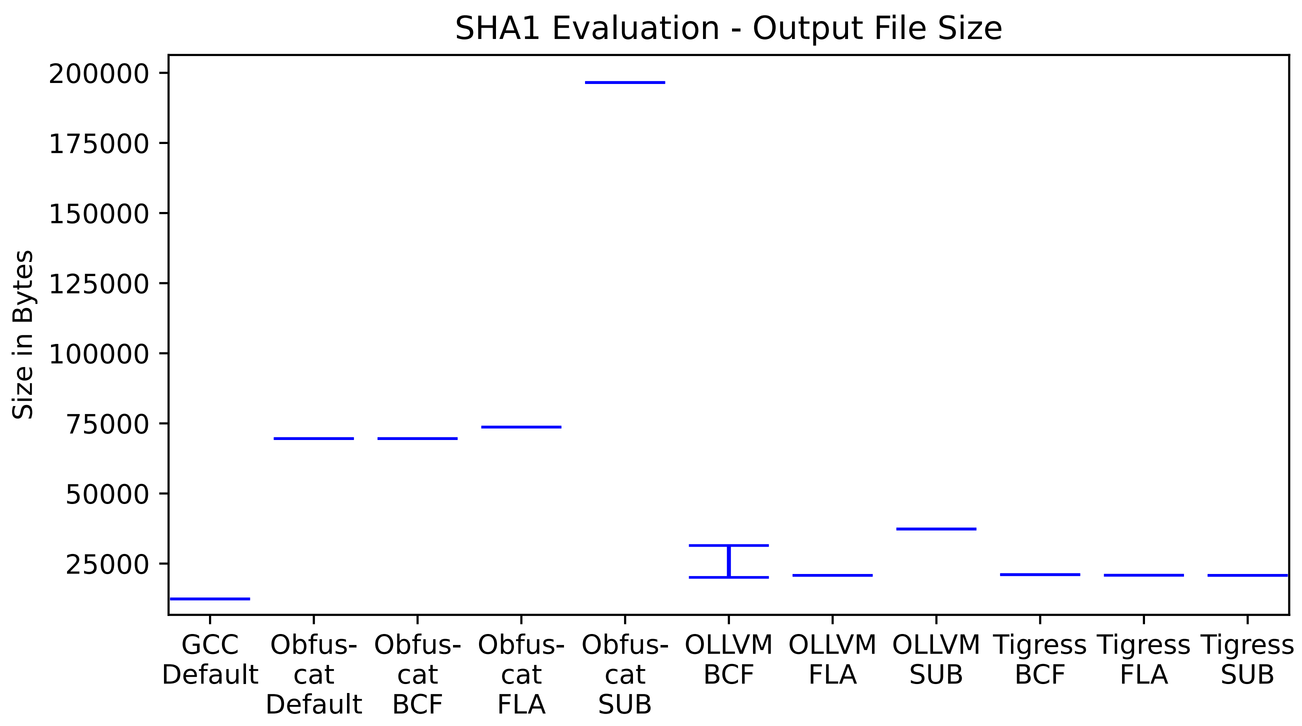

Figure 7.3: SHA1 File Size Comparison

For SHA1 in 7.3, the GCC Baseline was constant at 12 404 bytes. Because of the size of the SHA1 code, all techniques applied a noticeable overhead. The lowest was Tigress SUB with an average of 20 814.4 bytes, and OLLVM's FLA with a constant 20 816 bytes. OLLVM's SUB generated the largest non-Obfuscat files with an average of 37 324.08 bytes size. All Obfuscat files were especially large, with the baseline and BCF being at 69564 bytes, FLA at 73 660 bytes, and SUB at 196 540 bytes. OLLVM's BCF obfuscation had a noticeable standard deviation between compilations of 5 685.086 bytes. For OLLVM's SUB and Tigress BCF, FLA, and SUB the deviation was a few bytes at most.

Overall the files generated by Obfuscat were generally larger than what the other tools generated, with SHA1 being almost 4x worse. Precision-wise, Obfuscat output always had a constant file size. Only the GCC Baseline and Tigress Flattening had no deviations between compilations otherwise, especially Bogus Control Flow resulted in large deviations, most likely because of different predicates being picked each time.

The runtime overhead for the different input sizes was similar, though it is visible that not all techniques scale the same depending on the input. For randomized input with a length of 1 024 bytes, the following can be observed:

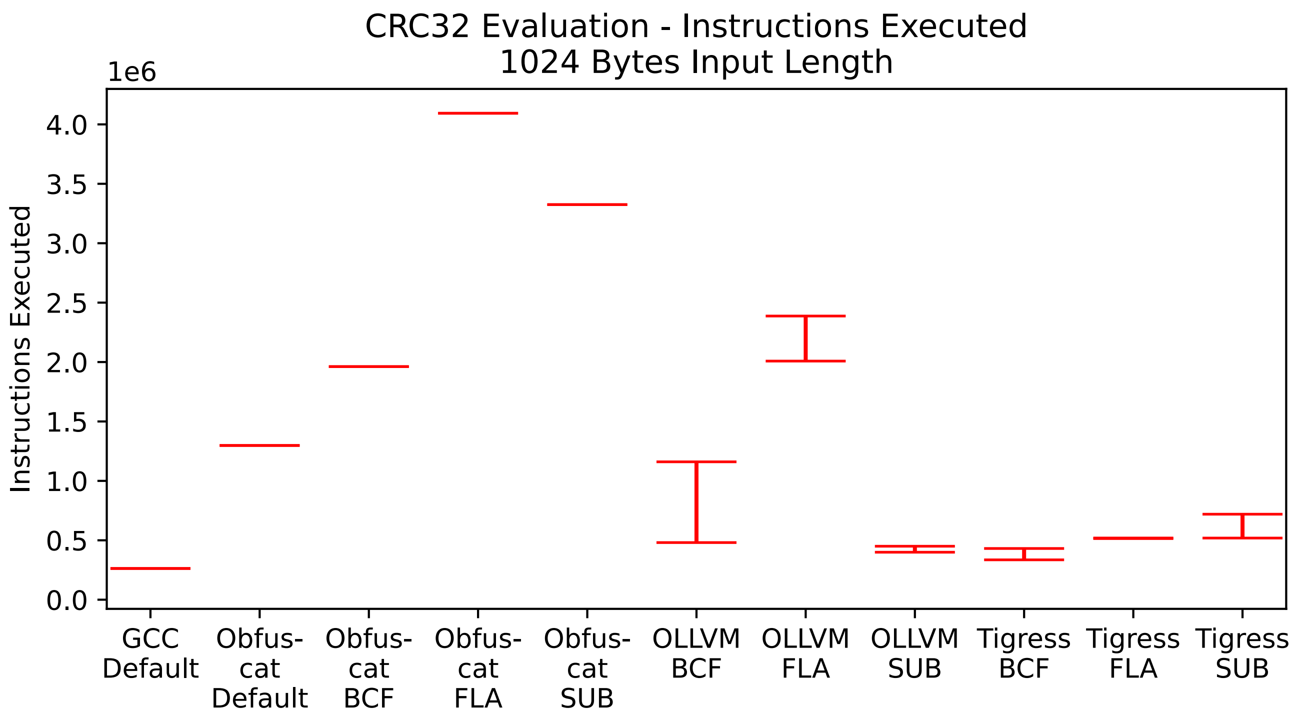

Figure 7.4: CRC32 Instructions Executed Comparison for 1024 Byte Inputs

For CRC32 in 7.4, the GCC base program executed 262 765 instructions. The results for Obfuscat were constant for all runs, with the default at 1 297 910 instructions, and the maximum at FLA with 4 093 706 instructions. Obfuscat FLA executed almost two times the instructions of OLLVM FLA, which averaged 2 197 670.66 instructions, and four times of Tigress FLA which averaged 516 865.18 instructions. While Obfuscat's results were the same across all binaries and inputs, large deviations were observed with all techniques of OLLVM and Tigress. OLLVM's BCF averaged 820 656.62 instructions but had a standard deviation of 339 775.315 instructions. Similarly, OLLVM FLA and Tigress SUB deviated by 189 541.466, and 100 388.142 instructions respectively. Tigress FLA deviated the least with 2 298.084 instructions.

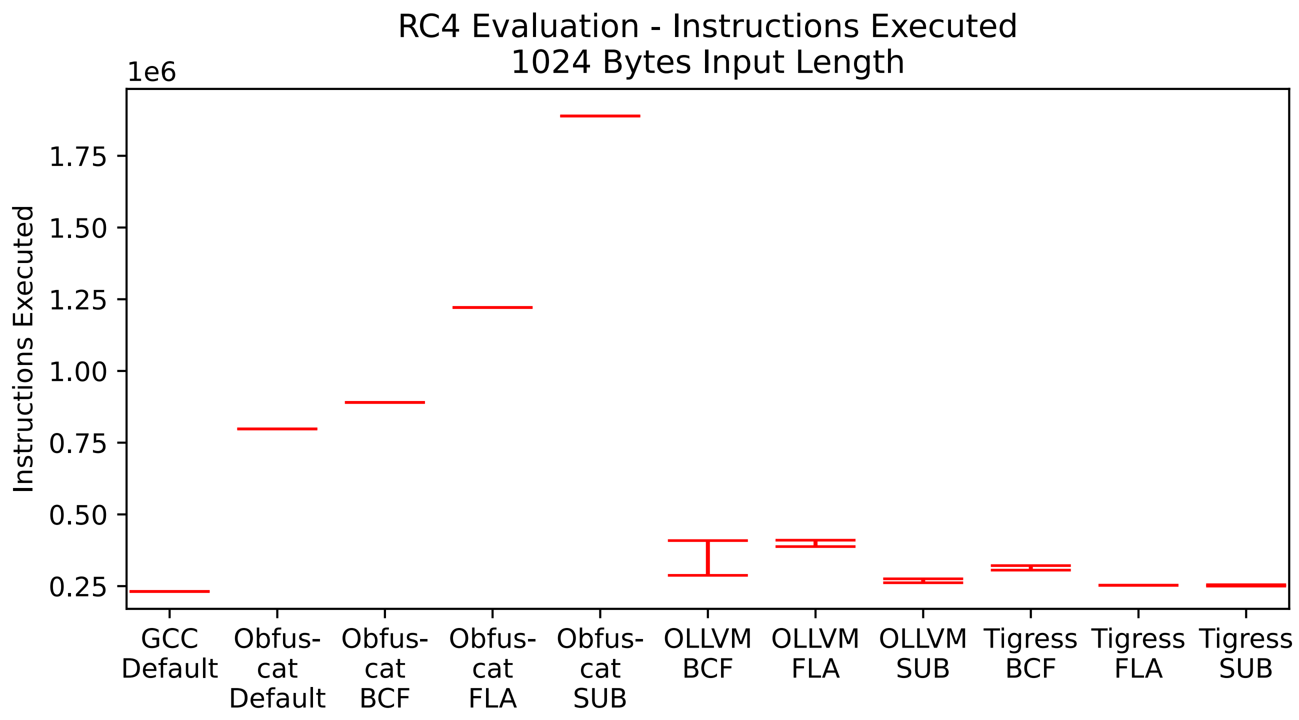

Figure 7.5: RC4 Instructions Executed Comparison for 1024 Byte Inputs

For RC4 in 7.5, the baseline was 231 341 instructions. Again Obfuscat's results were all constant. However, even without any techniques significantly more instructions had to be executed than for the other tools. Obfuscat's baseline was 798 068 instructions with no techniques as the lowest, and 1 888 688 instructions with SUB applied. In comparison, all techniques of Tigress and OLLVM were between 252 490.2 and 398 972.06 average instructions executed. Again, the deviation between runs was large for these tools. For Tigress BCF, the standard deviation was 7 913.161 instructions, 437.333 for FLA, and 2 457.6 for SUB. For OLLVM even 60 603.55 instructions for BCF.

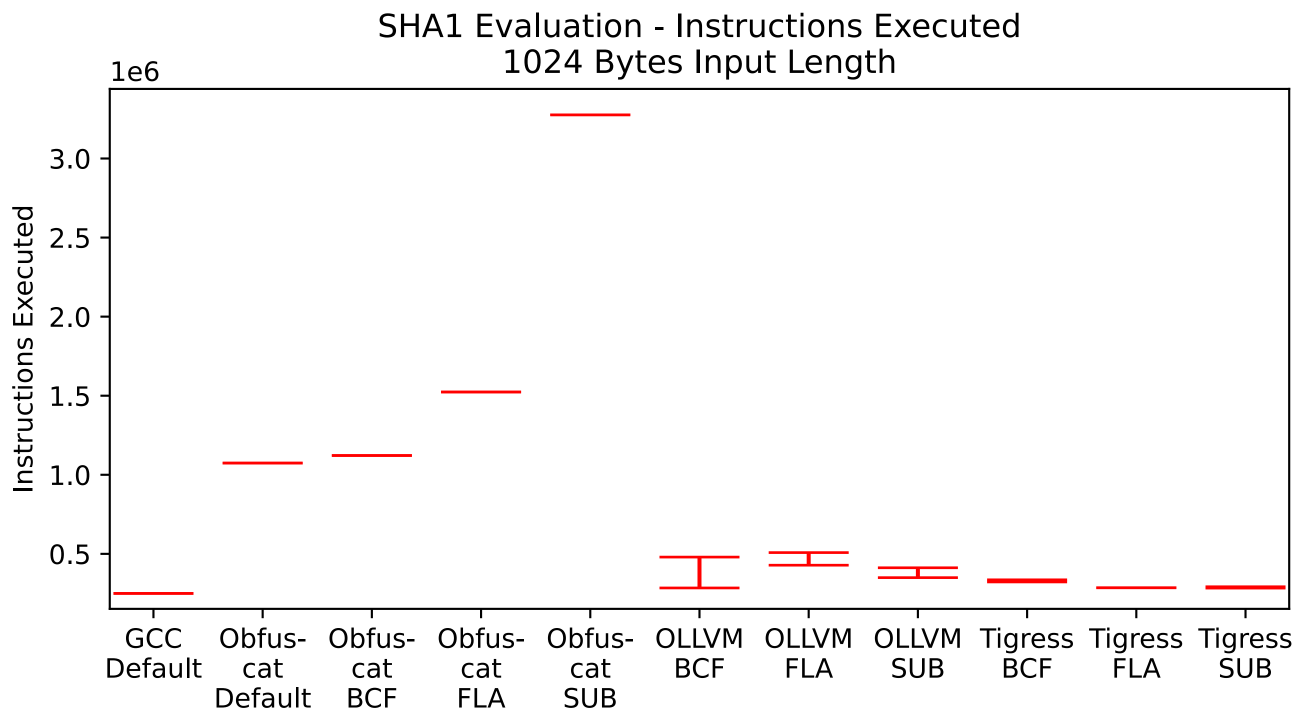

Figure 7.6: SHA1 Instructions Executed Comparison for 1024 Byte Inputs

For SHA1 in 7.6, the GCC baseline was 249 897 instructions. The Obfuscat results again were all constant, but even for the default version twice as high as for the worst technique of the other tools.The default was 1 073 996 instructions and SUB was 3 276 116 instructions. The other tool's average was between 287 732.02 instructions, and 468 142.34 instructions. Tigress' standard deviation was relatively low, with FLA at 143.346 instructions, SUB at 4 288.382 instructions, and BCF at 6778.603 instructions. OLLVM's deviations were a lot higher, with SUB at 31 226.187 instructions, FLA at 39 835.731 instructions, and BCF at 97 605 instructions.

The overall observation is that the overhead by Obfuscat for everything except CRC32 was always at least twice as large compared to the other tools. For CRC32, the overhead of Obfuscator LLVM's BCF and FLA was similar. Precision-wise, all tools except Obfuscat and the GCC Baseline had deviations between runs. For some, they were very significant. Only Tigress Control Flow Flattening had minor deviations for all tested programs. Obfuscator LLVM's Bogus Control Flow results especially had large deviations between the runs, similar to how it behaved for size deviations.

Another interesting observation from the runtime tests is that not all of them scale the same. The trends of how they behave relative to input size can be seen in the resulting data.

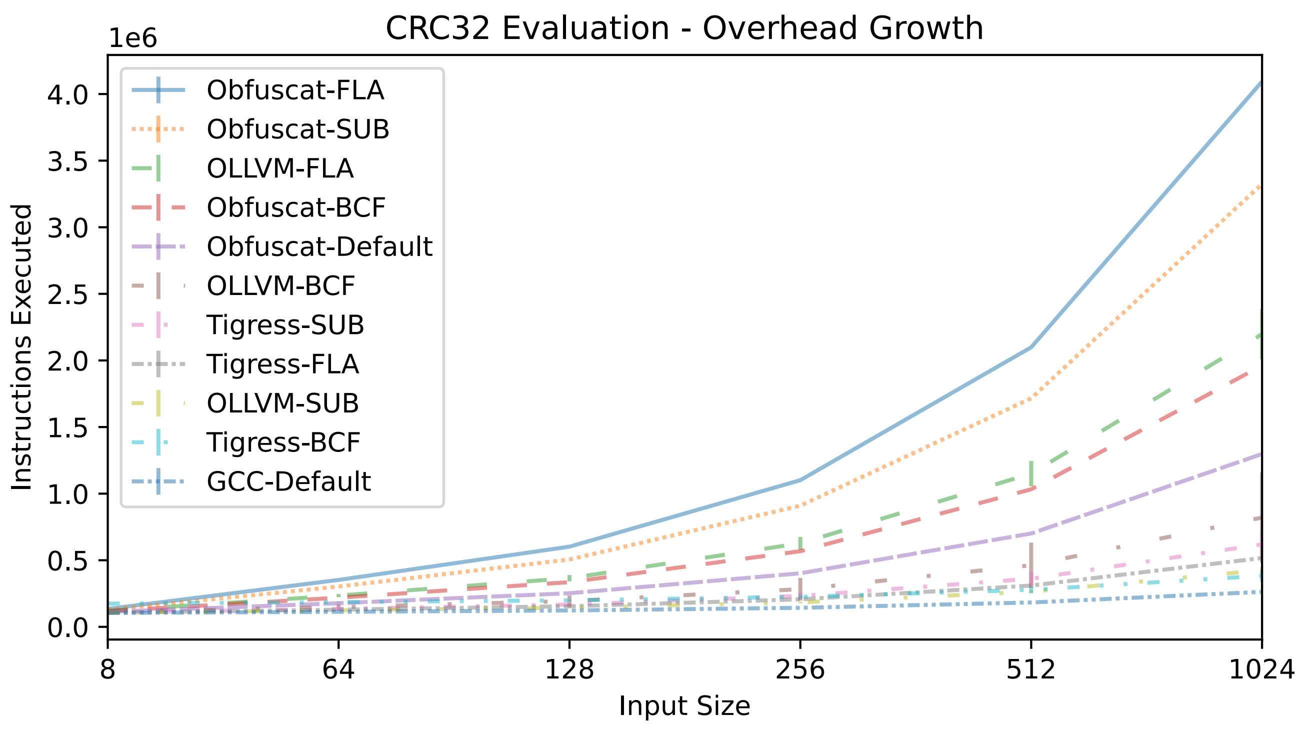

Figure 7.7: CRC32 Instructions Executed Comparison

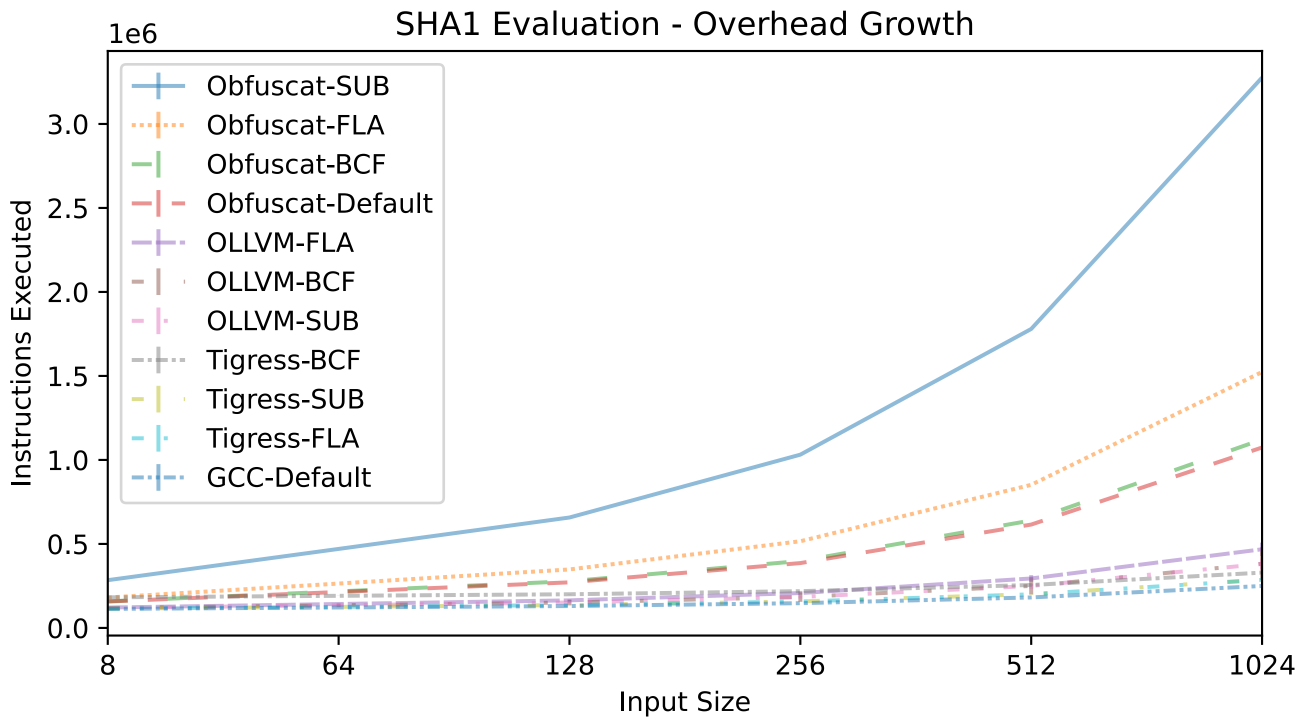

Figure 7.8: SHA1 Instructions Executed Comparison

For CRC32 in 7.7 the results of the 1 024 input are similar for all tested input sizes.

For SHA1 in 7.8 the scaling of Obfuscat Operation Encoding seems to increase more rapidly than the other techniques.
Overall the scaling of all tested techniques seems to be polynomial. The standard deviation also scales with the input size.

7.3 Strength against Symbolic Execution

While it is hard to test the strength of the techniques against manual attacks, it is possible to get a little bit of insight into how they match up against generic automated scripts. For this, a generic script for angr [1], a python symbolic analysis tool, has been prepared to measure how much time each technique adds to the analysis. The script takes a program as input and analyses which program argument input will make the program return 1. The only assumptions programmed into the script are the length of the correct input, the fact that the correct input has to be the second argument, and the method to read the return value. The to-be-obfuscated programs first check if the supplied input length is correct, and then sequentially check each input byte against the correct password. Again, Obfuscat was compared against Obfuscator LLVM and Tigress with the same techniques as in 7.2. The correct password had a length of 64 bytes for the tests, and for each technique of each tool 100 binaries were compiled and run 5 times. The testing was run on a system with two dedicated Intel Xeon Gold 6 140 cores.

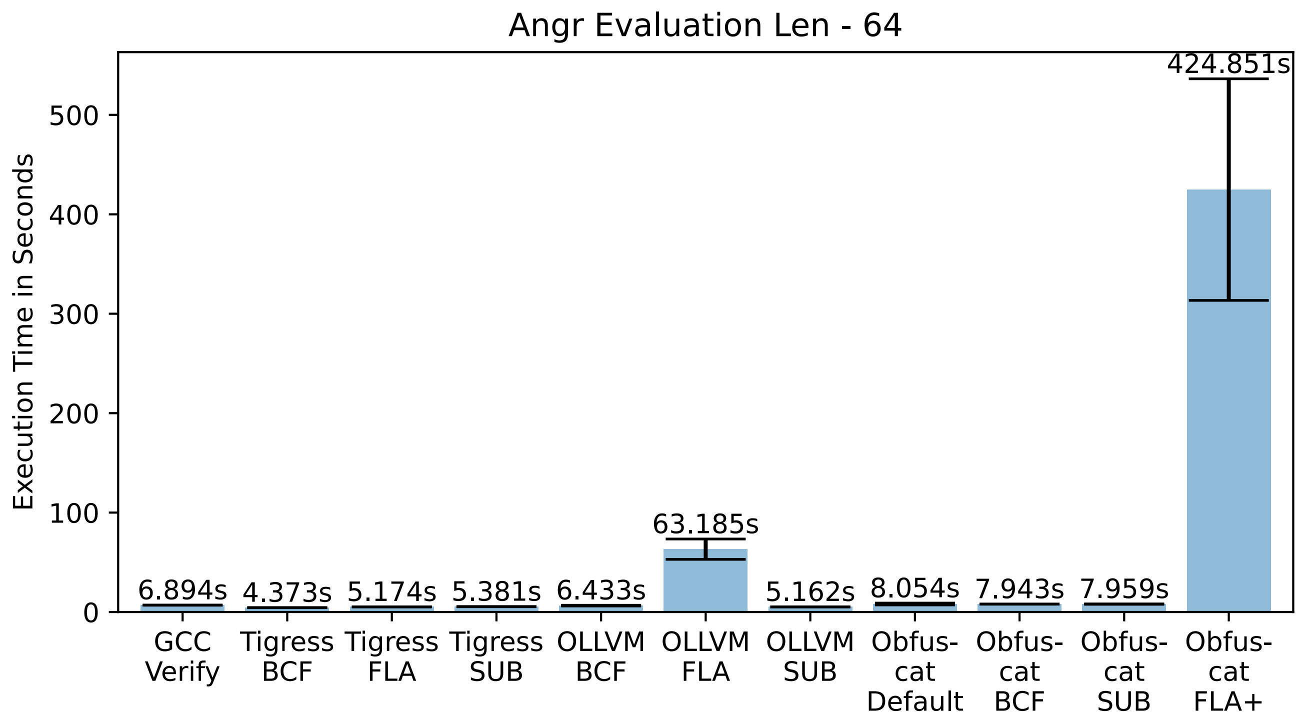

Figure 7.9: Comparison against Angr: Testing the strength of the techniques against Symbolic Execution

[1]https://angr.io/

The results 7.9 show by quite a margin that only Obfuscator LLVM's Control Flow Flattening and Obfuscat's Control Flow Flattening had a meaningful impact on the time the analysis took. The baseline GCC analysis took an average of 6.894s, Obfuscator LLVM's Control Flow Flattening took an average of 63.185s and Obfuscat's Control Flow Flattening took an average of 424.851s.

This simple test shows that individually, Obfuscat behaves comparably to other tools, but it has to be put into perspective to be meaningful. First, most of these techniques are meant to be used combined and are individually very weak. Second, while Obfuscat's Control Flow Flattening got the best results by a large margin, this result is mostly expected. This is because Obfuscat's methodology is not a simple implementation like the settings used for Tigress or Obfuscator LLVM yield. Instead, Obfuscat's implementation contains the strengthening methods of Cappaert and Preneel [17] , and the results mostly prove that their methods of strengthening are effective. Third, all of these results are only valid for the specific password length and internal program structure. Because of the complexity of the tool and analysis methods it applies, different input programs may yield completely different results.

Chapter 8

Discussion

The biggest problem shown by the comparison against other tools is that for programs, which are not small, the approach of this work causes a significantly larger overhead than others. While twice the size and executed instructions are reasonable in some applications, it is not always the case. Most of the unnecessary overhead is caused by padding, which is used on the native code level, and for most obfuscation techniques. As discussed in this work, to get perfectly precise results, this is required to some degree. By using a better compilation model or by improving the node to assembly translations, the efficiency could be generally improved. Additionally, the costly obfuscations could be more optimized, this could influence the strength of the techniques though. Instead, it could also be interesting to not aim for precise results and only calculate the worst-case overhead. This would allow for proper optimization on the native code and not require any padding in the obfuscation techniques. The obvious disadvantage would be that the minimum amount of overhead could not effectively be calculated. Also, for the precision on the upper bound to hold, the formulas for the techniques would not change. The calculations of the size and instructions by the Code Generators would also require calculating the worst-case output. While the generated native code would be smaller and more efficient, the estimation formulas would yield the same results as before. To make sure that all possible outputs still fit the constraints, the same resources as right now would need to be reserved. A more simple alternative would be to make use of all the padding. In the current implementation, it serves no purpose except to take up space. The space could be used for further obfuscation on the assembly level, which the framework currently does not handle at all. Larger paddings could even be replaced with anti-debugging techniques or anti-tampering checks. This approach is interesting because it would allow using anti-reverse-engineering tricks in a protected program without causing any additional overhead.

Another question that came up during the development was how to calculate runtime overhead. This work settled on using the most predictable unit of executed instructions. In actual usage, the amount of instructions executed is not interesting though. Usually, it is about the time a program takes. The amount of executed instructions and the time a program takes to execute are related, but on most architectures calculating the time from the instruction count is not trivial. It would be interesting to see if it is feasible to build a model for a specific embedded device to precisely calculate the execution time. Calculating the execution time from the instruction count should be relatively simple for single-cycle processors (some MIPS implementations), so a compiler backend for it might make sense. Alternatively, implementing the Virtual Machine of the Virtualization Obfuscation in a hardware description language and running it on an FPGA might make precise time calculations possible.

Chapter 9
Conclusion

Per the initial assumption, this evaluation demonstrated that the actual size and runtime overhead for obfuscations by existing tools varies between compilation runs. For some techniques, this variance is quite significant and results in noticeable differences. With the design of a general framework and rules for obfuscation techniques, Obfuscat [1] was implemented as a tool to produce predictable overhead of the output. This goal has been successfully met, but compared to the existing tools, the overhead caused is often $> 2x$ more. Through formulas for each obfuscation technique, it is possible to calculate the resulting binary size and amount of instructions that will be executed for a given input. This makes it possible to mathematically calculate the overhead of these techniques, even when combined. With the help of automatic solvers, it is possible to calculate which obfuscations to use for any additional user-given constraints.
As the extra overhead is the biggest problem of the design and implementation, future work would focus on reducing it and making use of the mandatory padding as previously mentioned. Additionally, strengthing the implementations of the obfuscation techniques similar to the Control Flow Flattening seems generally desireable.

[1]https://github.com/Pusty/Obfuscat

List of Figures

List of Tables

Bibliography

[1] C. Collberg, J. Davidson, R. Giacobazzi, Y. Gu, and A. Herzberg, “Toward digital asset protection”, *IEEE Intelligent Systems*, vol. 26, pp. 8–13, Nov. 2011. DOI: 10.1109/MIS.2011.106.

[2] S. Banescu and A. Pretschner, “A tutorial on software obfuscation”, in Jan. 2017. DOI: 10.1016/bs.adcom.2017.09.004.

[3] H. Xu, Y. Zhou, and M. R. Lyu, *N-version obfuscation: Impeding software tampering replication with program diversity*, 2015. arXiv: 1506.03032 [cs.CR].

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, “Protecting software through obfuscation: Can it keep pace with progress in code analysis?”, *ACM Comput. Surv.*, vol. 49, no. 1, Apr. 2016, ISSN: 0360-0300. DOI: 10.1145/2886012. [Online]. Available: https://doi.org/10.1145/2886012.

[5] T. Proebsting and S. Watterson, “Krakatoa: Decompilation in java (does bytecode reveal source?)”, Oct. 1998.

[6] J. C. King, “Symbolic execution and program testing”, *Commun. ACM*, vol. 19, no. 7, 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[7] B. Yadegari and S. Debray, “Symbolic execution of obfuscated code”, Oct. 2015, pp. 732–744. DOI: 10.1145/2810103.2813663.

[8] B. Barak, O. Goldreich, R. Impagliazzo, *et al.*, “On the (im)possibility of obfuscating programs”, *J. ACM*, vol. 59, no. 2, May 2012, ISSN: 0004-5411. DOI: 10.1145/2160158.2160159. [Online]. Available: https://doi.org/10.1145/2160158.2160159.

[9] X. de Carné de Carnavalet and M. Mannan, “Challenges and implications of verifiable builds for security-critical open-source software”, *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[10] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations”, *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, Jan. 1997.

[11] C. Collberg and J. Nagra, “Surreptitious software - obfuscation, watermarking, and tamperproofing for software protection”, in *Addison-Wesley Software Security Series*, 2009.

[12] H. S. Warren, *Hacker’s Delight*, 2nd. Addison-Wesley Professional, 2012, ISBN: 0321842685.

[13] B. Batteux, 2020. [Online]. Available: https://eshard.com/posts/d810_blog_post_1/.

[14] F. Gröbert, C. Willems, and T. Holz, “Automated identification of cryptographic primitives in binary programs”, in *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–60, ISBN: 978-3-642-23644-0.

[15] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, “Information hiding in software with mixed boolean-arithmetic transforms”, in *Information Security Applications*, S. Kim, M. Yung, and H.-W. Lee, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75, ISBN: 978-3-540-77535-5.

[16] C. Wang, J. Davidson, J. Hill, and J. Knight, “Protection of software-based survivability mechanisms”, in *2001 International Conference on Dependable Systems and Networks*, 2001, pp. 193–202. DOI: `10.1109/DSN.2001.941405`.

[17] J. Cappaert and B. Preneel, “A general model for hiding control flow”, in *DRM ’10*, 2010.

[18] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – software protection for the masses”, in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed., IEEE, 2015, pp. 3–9. DOI: `10.1109/SPRO.2015.10`.

[19] K. Coogan, G. Lu, and S. Debray, “Deobfuscation of virtualization-obfuscated software a semantics-based approach”, Oct. 2011, pp. 275–284. DOI: `10.1145/2046707.2046739`.

[20] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The java™ virtual machine specification. java se 7 edition*, Oracle, 2012.