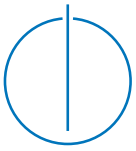
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Software Integrity Protection Versus Machine Learning Attacks

Nika Dogonadze



DEPARTMENT OF INFORMATICS

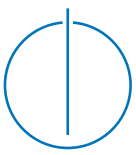
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Software Integrity Protection Versus Machine Learning Attacks

Software Integrity Schutz gegen Maschinelles Lernen basierter Angriffe

Author:	Nika Dogonadze
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor:	Dr. Sebastian Banescu
Submission Date:	15 Sep 2021



I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 15 Sep 2021 Nika Dogonadze

Acknowledgments

I would like to thank Prof. Dr. Alexander Pretschner for taking on the role of my supervisor. I would also like to express enormous gratitude to Dr. Sebastian Banescu for the diligent dedication of his time and resources in advising me on this work. Our weekly meetings were immensely helpful. Finally, I would like to thank Dr. Mohsen Ahmadvand for his knowledgeable insights and availability to help.

Abstract

It is frequently critically important that some software applications run exactly as intended by the developer. Licensed programs need to verify the validity of the copy to avoid financial damage caused by illegal software distribution. However, many such applications have to run on users' machines. The user has absolute control over all system parts, including network, operating system, memory, CPU, storage, and other hardware. Thus, the license checking part of the code can be easily circumvented. More precisely, this attack model is defined as Man At The End (MATE). There has been a vast amount of research into creating tools for Software Integrity Protection (SIP) against such a strong adversary. On the other hand, researchers are continuously developing sophisticated attacks against these protections. However, robust machine learning models, specifically deep neural networks against SIP protection schemes, have not been comprehensively examined. In this work, we implement a 5-stage machine learning pipeline that is a playground for SIP experiments. We generate a large dataset of protected and obfuscated programs and evaluate both the resilience of existing SIP schemes as well as the best machine learning-based attacks against them. From the integrity protection (defender) side, we verified a previously known observation that obfuscation is vital for functional SIP. Furthermore, we discovered some protection & obfuscation combinations that are particularly effective relative to their impact on program size. We also give general guideline for choosing specific protection & obfuscation combination (Figure 5.11) based on program size increase. From the attacker's side, we found deep learning models to be the most effective provided the attacker has a sufficiently large dataset and compute for training. Furthermore, we offer a transfer learning (Pan & Yang 2010) approach when resources are limited.

Contents

1. Introduction

Context. It is critically important for some applications to run exactly as intended by the developer. Any unwarranted modification could be a security issue in addition to the risk of monetary loss and reputation damage. For instance, commercial software often comes with built-in license verification ensured by the software itself. There is an obvious incentive for an end-user to modify the software to bypass this verification step, however, this comes at a cost to the creator. Current semi-autonomous vehicles need to check for driver presence in the seat in order to engage the driving assistant. There have been cases where people try to circumvent this check, causing reputation damage and even the death of a person described by Barry 2021.

1.1. Software Integrity Protection

One approach to prevent software misuse after distributing it includes software integrity protection (SIP) mechanisms. The idea is to take the source code of a working application and package it together with an additional logic, which detects and reacts to tampering. Ahmadvand, Pretschner & Kelbert 2018 described complete taxonomy and effectiveness of software integrity protection techniques. Another term for SIP used in literature is *tamper-proofing* defined by Collberg & Thomborson 2002 as "[a way] to ensure that [a program] executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution". We further discussed the attacker model of an adversary in the following section 1.1.

Types of SIP. Software integrity protection can be done with the help of trusted hardware, such as Intel SGX (Costan & Devadas 2016). However, purely software approaches are more convenient since they do not require having specialized hardware. Our work will only focus on 3 instances of latter approach - *Self-Checksumming*, *Oblivious Hashing* and *Control Flow Integrity*. Each of those can be applied multiple times and even combined as described by Ahmadvand, Fischer & Banescu 2019. Chapter 2 describes different protection methods in detail. These methods can be effective, but they are not invulnerable to attacks themselves.

Attacking SIP. In addition to having complete control of the host machine and the network, the attacker can inspect and modify any part of the program. This attack model is called Man-At-The-End (MATE), defined in the work of Akhunzada, Sookhak,

Anuar, *et al.* 2015. The powerful adversary has authorized and unlimited access to the program, including any self-protecting component it contains. Simple pattern matching has shown to be very effective in discovering the protection scheme used for SIP by Wessel 2019. However, primitive attacks immediately lose fruitfulness once obfuscation is applied to the code.

Obfuscation. Obfuscation is a technique to mask application source code to make it less readable for humans and software analysis tools. Any transformation can be applied to both the code and the data of a program as long as it remains functionally identical (Collberg, Thomborson & Low 1997). Obfuscation can be done at a source code level, usually for interpreted languages like Javascript. Assembly or bitcode obfuscation is preferred in compiled languages. In our work, we consider three obfuscation techniques - Instruction Substitution (IS), Bogus Control Flow (BCF), and Control Flow Flattening (CFF). Each of these approaches can be applied multiple times and in combination to achieve the desired level of effectiveness. The main trade-off here is between the effectiveness of obfuscation and the resource overhead it introduces. We describe these techniques in detail in chapter 2. The main objective of obfuscation in this work is not to hide the implementation details of original programs but to mask the particular integrity protection scheme used.

Machine Learning Attacks. Machine Learning (ML) models have been successfully used to detect SIP schemes in the presence of heavy obfuscation (Wessel 2019). The problem of detecting SIP scheme is naturally posed as a supervised learning classification task (Bishop 2006). In order to represent binary programs as an input to statistical ML models, Wessel 2019 uses well-known TF-IDF feature extraction (Section 2.5.4) on disassembled binaries. We propose that this particular representation for a binary could be a limiting factor for overall model performance. We evaluate this claim by defining a Deep Learning model (Wehle 2017) instead. There has been a growing trend of deep learning models generally outperforming traditional machine learning models in the past decade. One possible reason for this effect, as postulated by Bengio, Courville & Vincent 2013 is that DL networks learn good feature representations from raw data. In addition to this, learned features could be beneficial for transfer learning (Zhuang, Qi, Duan, *et al.* 2019) to other program-analysis-related tasks. Furthermore, we generalize *SIP classification* to *SIP localization* task. That is, classifying the SIP technique used on each of the small components of a program called *basic blocks*.

1.2. Attacks against SIP

According to the MATE attack model, the adversary has the ability to run the protected program in a fully controllable environment. Moreover, the attacker can change or

augment any of the program instructions. The ultimate goal is to bypass the SIP schemes built into the binary. Different attack modes we propose in the following chapter aim to gradually build up automated tools that would aid the attacker in disabling SIP. On the other hand, it is an excellent opportunity to evaluate the resilience of used SIPs.

1.3. Task Definition

We choose to work with subject programs in binary LLVM IR [1] format. This allows us to remain independent of programming languages. Also, to use the existing rich LLVM compiler infrastructure and tooling. The task of attacking SIP protection technique is defined as follows:

- *Software Integrity Protection Classification* - given the entire bitcode file of some program, predict if any of the 3 SIP methods (SC, OH, CFI) were used to protect any part of this bitcode.
- *Software Integrity Protection Localization* - given the entire bitcode file of some program, predict for each basic block if it contains tamper-checking condition from one of 3 SIP techniques (SC, OH, CFI).

We use task definitions to answer the research questions defined below.

Research Questions. This paper attempts to evaluate the effectiveness of machine learning attacks against various software integrity protection techniques (self checksumming, oblivious hashing, control flow integrity) in combination with obfuscation techniques (instruction substitution, bogus control flow, control flow flattening). In particular, we are interested in deep learning models due to better generalization and performance expectation. More specifically, our work attempts to answer the following research questions:

RQ1: **How well do SIP schemes hold up against machine learning attacks when combined with various obfuscations?** One of the most important values of a protection scheme is the difficulty of bypassing it. As machine learning tools get ubiquitous, protection schemes that are resilient to such automated attacks are needed. We take protected & obfuscated binary programs and train a deep neural network to attack and localize SIP schemes used in each. Then, we evaluate the performance of the model on previously unseen data.

[1] https://llvm.org/docs/LangRef.html

RQ2: **How do more expensive neural program embeddings - Code2Vec (Alon, Zilberstein, Levy & Yahav 2018) or IR2Vec (VenkataKeerthy, Aggarwal, Jain, *et al.* 2020) compare against traditional TF-IDF representation?** Deep neural networks are continuously replacing relatively old machine learning models with hand-built features. However, they come at the cost of increased compute requirements. Thus, the use of the latter needs to be justified by achieving better performance. We implement and evaluate both approaches and compare the performance.

RQ3: **How does obfuscation of protected programs impact SIP localization performance?**. Code obfuscation is known to hinder the attacker's ability to localize protection blocks (Wessel 2019). However, precisely what kind of obfuscations are the most effective against deep learning attacks for each SIP scheme is unclear. We define several different obfuscations and their combinations. Then, we run identical training and testing processes of our deep neural network to identify the obfuscation combinations that reduce the *SIP localization* model performance the most.

RQ4: **Which obfuscation techniques are the most efficient relative to increase in program size?**. Since the obfuscation techniques and their combinations can be infinitely stacked, it is always possible to make attacks more complicated by introducing an additional layer of obfuscation. This is not very effective since, at some point, the target program will become unusable. We aim to evaluate the effectiveness of obfuscations with respect to the cost of program size increase, finding the most cost-effective obfuscations.

RQ5: **How reliable are Machine Learning attacks against SIP with different obfuscations?** All the research questions presented above are concerned with the best possible machine learning model performance on average. We are also interested in understanding how reliable these attacks are. Do they always reliably produce the same results? Or do they highly depend on a particular choice of the training procedure? To answer this question, we run the experiments multiple times and evaluate the variability of the results with respect to input parameters.

RQ6: **How well does the SIP localizer model perform on a new program outside of its training data distribution?** In our model, the attacker has access to all the obfuscations & protections, but not to the source code of the protected program. So, the attacker could potentially generate a large dataset on some programs and train a powerful deep learning model. We want to gain insight into how well this model would perform on a novel protected program outside of its training distribution.

1.4. Organization

In the following **chapter 2** we provide some technical background and notations necessary to understand the motivation and approaches in this paper. It also contains information about closely related work published in recent years.

Chapter 3 discusses high-level approaches and methodology of how we attempt to answer the research questions defined above. It does not contain specific details but rather design decisions and justifications behind them. Then, **chapter 4** describes all the concrete implementation details of our experiments.

Chapter 5 presents actual results of our experiments. Since our results have many data points, we mostly resort to visual representations and include full result tables in the Appendix chapter A.

The **discussion chapter 6** looks at the results from the previous chapter closely and discusses tangential findings that are not limited to the research questions.

Finally, **chapter 7** draws conclusions about the entire work done in this paper. Here we also discuss possible limitations and prospects for future work.

Additionally, the **appendix chapter A** contains some of the very detailed information that is not central to the topic of this thesis. Nevertheless, they might be attractive to some of the readers.

2. Background & Related Work

This chapter introduces some of the background knowledge our work builds upon. We also include a brief review of related work, motivating our research questions.

2.1. LLVM

LLVM is a compiler framework originally introduced by (Lattner & Adve 2004). It consists of 3 main components:

- **Front-End** - Takes the source code written in the supported languages and produces assembly-like intermediary representation (LLVM IR).
- **Middle-End** - Takes a program in LLVM IR format, optimizes, and produces optimized LLVM IR code.
- **Back-End** - Compiles LLVM IR into target architecture, e.g., x86_64.

We rely on the LLVM framework to compile our source program into LLVM IR. This allows our dataset to remain programming language-agnostic. Also, to leverage the existing tools around this LLVM framework for working with the dataset. Figure 2.1 Shows a high-level diagram of the compilation process from source code to binary executable.

Name	Function	Reads	Writes
clang	C Compiler	.c	.bc
opt	Optimizer	.bc/.ll	.bc
llvm-dis	Disassembler	.bc	.ll

Table 2.1.: A list of tools from the LLVM framework used in our work.

2.1.1. LLVM IR

LLVM Intermediary Representation (IR) is an assembly-like representation of a program. It consists of a sequentially executed list of instructions. Some of these instructions

change the control flow, making jump to another, out-of-order instruction. Uninterrupted sequence of instructions form a *basic block*. By definition a *basic block* has to end in control-flow-altering instruction. Functions can contain one or more basic blocks, and finally, a *module* is an entire program containing all the function definitions.

LLVM IR is not directly executable machine code. However, it combines similar low-level concepts, such as branching and basic blocks with high-level abstractions, like variables, functions, and strong typing. It has a static single assignment (SSA)(proposed by Leung & George 1999) property which means that each variable is assigned exactly once, and every variable needs to be defined before it is used. This property makes automatic program optimizations significantly easier.

Intermediary representations of programs are usually stored in bitcode files with `.bc` extension or text files with `.ll` extension. Both formats are equivalent and can be used interchangeably with various LLVM tools. Full LLVM IR language reference is given on official LLVM website[1].

Listing 2.1.1 shows an example of a simple C function and the following listing 2.1.1 is the corresponding LLVM IR representation of this function.

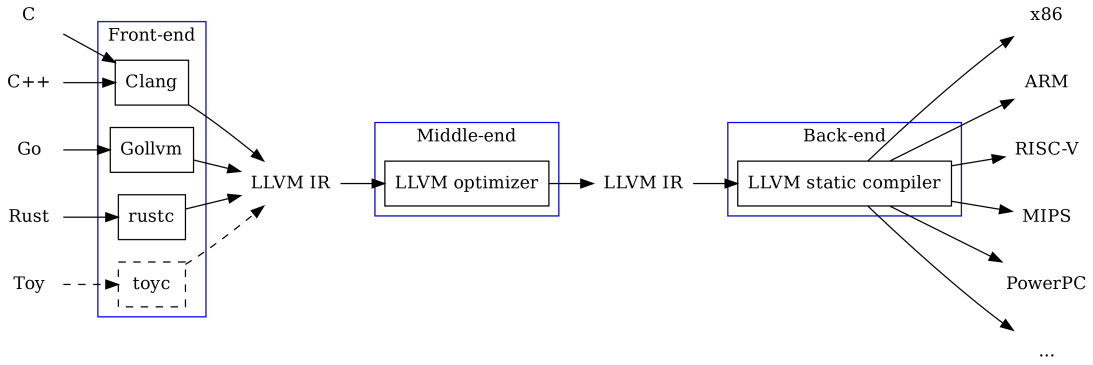


Figure 2.1.: Path from a source code to executable using with LLVM compiler framework. Visualization by (Eklind 2018).

```
1 int square(int n) {
2     return n * n;
3 }
```

Listing 2.1: Integer squaring function in C

[1] https://llvm.org/docs/LangRef.html

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @square(i32 %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul nsw i32 %3, %4
  ret i32 %5
}
```

Listing 2.2: Integer Squaring function in LLVM IR

2.1.2. LLVM Pass

LLVM pass operates on a unit of IR (module or function). There are two types of LLVM passes:

- Transformation pass will read and modify the unit of IR.
- Analysis pass will read the unit of IR and produce some high-level information about it.

The LLVM framework provides a number of optimization and analytical passes described on their website[2]). For instance, `-mergefunc` pass looks for equivalent function definitions and folds them, `-dot-cfg` prints control flow graph of a program.

LLVM framework also provides an interface to implement a custom LLVM pass using C/C++ language. The framework covers all the standard functionality between the passes, such as reading, parsing bitcode files, and writing back transformed units.

2.2. Man At The End (MATE) attack model

In this work, we operate under the (MATE) attack model assumption. The attacker is given a binary executable and can inspect, modify and run the program on a fully controlled host machine. There are no limitations on the attacker's access to the host machine's software and hardware, including the network, operating system, memory, CPU, and storage. However, the assumed attacker does not have access to the program source code. This limitation requires the attacker to invest some reverse engineering efforts in order to modify the program.

[2]https://llvm.org/docs/Passes.html

2.3. Software Integrity Protection (SIP)

Integrity protection of software refers to detecting unauthorized modifications of executable binaries. Protections against tampering can cover the entire executable code or only some sensitive portions of it. SIP techniques have been reviewed in the work by (Ahmadvand, Pretschner & Kelbert 2018). Generally, all approaches to the SIP can are split into two groups:

- Trusted Computing (TC) - Requires trusted hardware module on the attacker's host machine. It can be assumed out of reach of the attacker and provide some security guarantees. Examples of such approaches are Trusted Platform Module (TMP) (Garfinkel, Pfaff, Chow, *et al.* 2003), Intel SGX (Costan & Devadas 2016) and ARM TrustZone (Ngabonziza, Martin, Bailey, *et al.* 2016).

- Software Protection (SP) - The approach relies purely on software. The general idea is to precompute some invariant properties of a program that would inevitably be changed by tampering and then include the invariant checking into a program itself. Such invariants could be derived from instruction checksums (Chang & Atallah 2002), memory access patterns (Y. Chen, Venkatesan, Cary, *et al.* 2002) and control flow stacks (Abadi, Budiu, Erlingsson & Ligatti 2009). Software protection mechanisms can be attacked too, but if they sufficiently increase the effort required for the attack, it might become not worthwhile. SP can include obfuscation and hardening (Abrath, Coppens, Broeck, *et al.* 2020).

Previous works by (Dedić, Jakubowski & Venkatesan 2007) and (Barak, Goldreich, Impagliazzo, *et al.* 2001) have shown that it is generally impossible to create absolutely secure software defence against polynomial-time adversary. However, practically, the attacker's resources are finite. It could still be possible to have purely software-based protection that provides a sufficient level of protection so that it is not worthwhile for the adversary to conduct the attack.

TC can be more robust provided the availability and security of the hardware module on the host machine. Still, it has some drawbacks. There are several effective attack vectors on hardware modules. A work by (Nilsson, Bideh & Brorsson 2020) surveys dozens of published attacks against Intel SGX split into seven categories. They suggest that these attacks can successfully thwart the hardware defense provided by SGX. There exist number of similar against TPM as well (Van Bulck, Piessens & Strackx 2018, Moghimi, Sunar, Eisenbarth & Heninger 2019).

The following subsections describe particular SIP protection schemes implemented and evaluated in this work.

2.3.1. Self Checksumming (SC)

This protection scheme is based on the program inspecting its own instruction sequences. SC takes a sensitive portion of the original program, calculates some invariant, and injects additional code that verifies this invariant. Usually, this invariant is some hash function of sensitive instructions. Using a good hash function is important as the attacker can not find a collision. Otherwise, sensitive instructions can be modified while keeping the same hash value. The injected CFG node that checks for the invariant is called a `guard node`. There can be multiple guard nodes (Chang & Atallah 2002). Also, some guard nodes can be checking other guard nodes. The listing 2.3.1 shows an example of SC protection guard code. The Values in brackets <> need to be computed after the compilation step and patched into a binary as a post-processing step.

```
1  void guardCheckHash(
2      void *address,
3      const unsigned int length,
4      const unsigned int expectedHash
5  ) {
6      auto beginAddress = (const unsigned char *) address;
7
8      unsigned char hash = 0;
9      for (unsigned int i = 0; i < length; i++) {
10         hash ^= *(beginAddress + i);
11     }
12
13     if (hash != (unsigned char) expectedHash) {
14         // Unexpected hash code
15         exit(-1);
16     }
17 }
18
19 int sensitiveFunction(int n) {
20     return n * n;
21 }
22
23 int main() {
24     guardCheckHash(sensitiveFunction, <function length>, <hash_value>)
25 }
```

Listing 2.3: Simple SC guard example code

The program with SC protection exhibits an unusual behavior of reading its own segment code. This property was successfully used by (Qiu, Yadegari, Johannesmeyer, *et al.* 2015) to locate and defeat SC guard nodes. However, the complexity of their approach makes it unusable on large programs.

Another possible attack on self-checksumming integrity protection is to have two versions of program memories: untainted and tampered. If every segment code read gets redirected to the untainted memory, then SC can be circumvented. This particular attack is negated by the improved, self-modifying SC (Giffin, Christodorescu & Kruger 2005).

2.3.2. Oblivious Hashing (OH)

The shortcomings of unusual self-reading behavior of the SC approach are addressed by Oblivious Hashing approach (Y. Chen, Venkatesan, Cary, *et al.* 2002). Instead of reading the code segment, OH approach verifies memory values at the time of execution. (Ahmadvand, Hayrapetyan, Banescu & Pretschner 2018). The protected program keeps some hash values that get periodically updated and verified based on current memory values. Figure 2.2 shows an example of OH protected program execution. This protection method adds hashing instructions after memory access operations and later verifies the hashes.

This approach has a significant shortcoming of not protecting operations working on non-deterministic data as their hash values depend on the memory values and thus can not be pre-computed. This issue was partially addressed by (Ahmadvand, Hayrapetyan, Banescu & Pretschner 2018) with a modification over OH called *short range oblivious hashing*. They conduct data-dependency analysis of a program and identify data-independent instructions that depend on the control flow. The SROH method bases hash values on the sequence of control flow blocks preceding the target instruction.

2.3.3. Control Flow Integrity (CFI)

Another SIP approach proposed by (Abadi, Budiu, Erlingsson & Ligatti 2009) doesn't rely on reading the instruction or memory values. It keeps track of the program execution path in the control flow graph and compares it with predefined paths. When some function is called, it gets registered to *shadow call stack*, which is a special memory segment that has to be inaccessible to the attacker. Function returns cause de-registration from this stack. When a sensitive segment starts, the call stack has to be verified against predefined proper control flow. The Figure 2.3 shows an example of valid and invalid control flows.

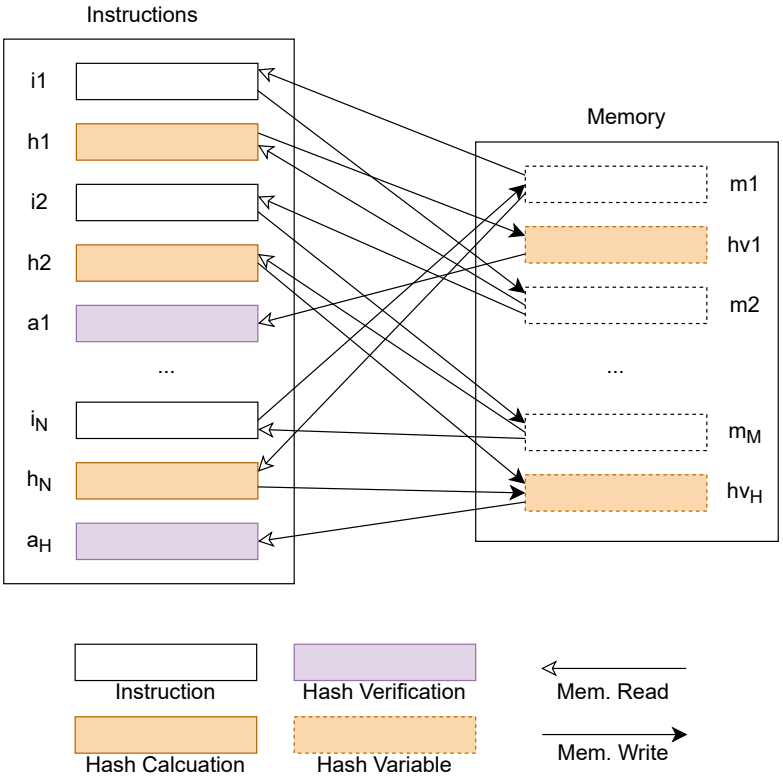


Figure 2.2.: Sequential execution of a program with OH protection. Memory access instructions are followed by hashing calculations. The hash values are later checked with hash verification instructions.

SIP compositions can be stacked and combined (Ahmadvand, Fischer & Banescu 2019), but it has been shown that combining SIP schemes does not improve resilience against pattern-matching attacks.

2.4. Obfuscation

Obfuscation is aimed at a similar goal as software protection. However, it does not functionally modify the code but merely rewrites it to make it difficult to understand. It can be applied both at the source code or machine code level. The process of taking a complex system and examining the principles of how it works is called *reverse engineering*. Obfuscation is one of the leading generic tools against reverse engineering, but it has limited effectiveness and some weaknesses, such as deobfuscation attacks (Salem & Banescu 2016a).

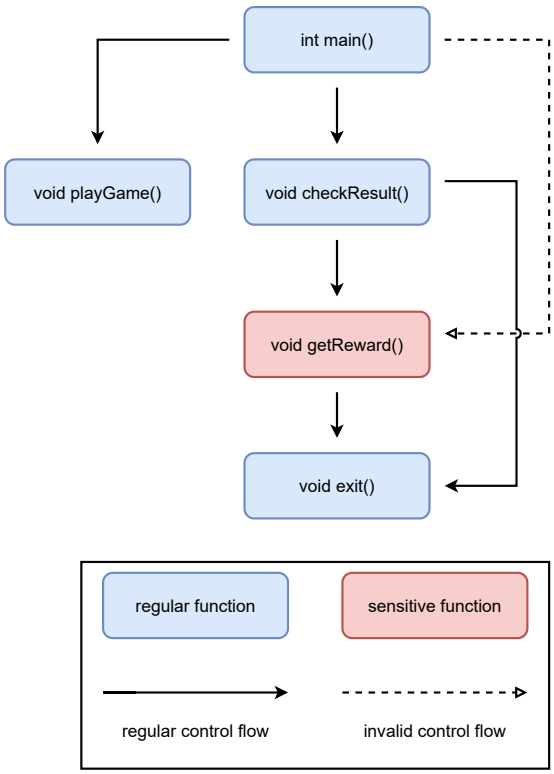


Figure 2.3.: An example of valid program control flow execution (solid line) and an invalid flow from `main()` to `getReward()` (dashed line).

In our program integrity scenario, the attacker tries to localize guards in the protected program. SIP attack requires some examination and analysis of the program instructions. Obfuscation in conjunction with the SIP scheme makes it more costly to break the protection. We discuss three different types of obfuscations Instruction Substitution (IS), Bogus Control Flow (BCF), and Control-Flow Integrity (CFI). Since this work focuses on assembly-like intermediary language LLVM IR, all the obfuscations are assumed to be done on intermediary representations.

2.4.1. Instruction Substitution (IS)

The instruction substitution works at the individual instruction level of a program. Each instruction can be replaced with functionally equivalent but more convoluted, potentially more than one instruction. The example taken from (Y. Chen, Venkatesan,

Cary, *et al.* 2002) shows that the assignment operator $a = b + c$ can be rewritten as:

$$a = (b \oplus c) - ((-1 - 2b) \vee (-1 - 2c)) - 1$$

Any number of such equivalencies can be combined to generate a complex expression. However, this approach alone does not provide much protection since it can be effectively attacked with compiler optimizations (Junod, Rinaldini, Wehrli & Michielin 2015).

2.4.2. Bogus Control Flow (BCF)

This obfuscations approach operates on the control flow graph of a program. It adds bogus nodes to the graph with conditional jumps that always evaluate to false. Thus, these bogus jumps are never taken during the normal run of the program. Still, the presence of bogus nodes makes static analysis of the program more difficult. Jump conditions can be made *opaque* (Collberg, Thomborson & Low 1998). Stealthy opaque predicates are expressions that can be simplified to a fixed value but are difficult to evaluate statically. For example, the expression $(1 + a(a + 1)|2) mod 2 = 1$ is always 0, but it seems to depend on a particular value of a variable. An example control flow graph of BCF obfuscated program is shown in Figure 2.4.

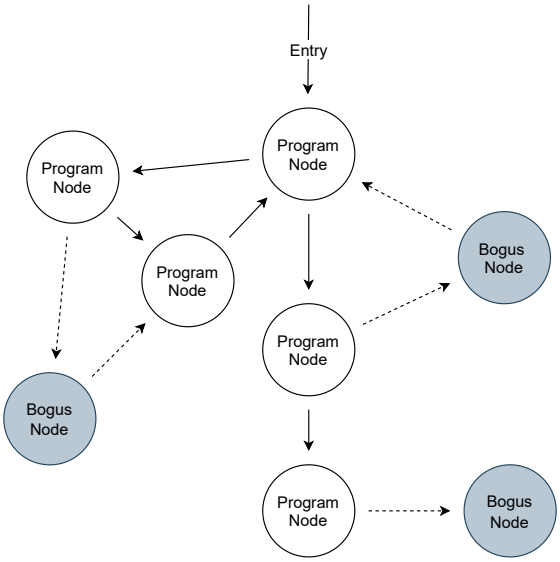


Figure 2.4.: An example of program control flow graph with injected bogus nodes. Dashed lines represent jumps with opaque predicates that are never taken. Regular lines represent original jumps between nodes.

2.4.3. Control Flow Flattening (FLA)

Another common CFG-graph-based obfuscation is called Control Flow Flattening (FLA). Here a set of graph node relations are replaced by a central controller node, where each existing connection must go through. An example is described in Figure 2.5. The graph on the left shows original nodes with their relations. The right side shows the same graph with FLA obfuscation. The new controller node decides which block to jump to next. Every original block (except the exit node) jumps back to the controller node.

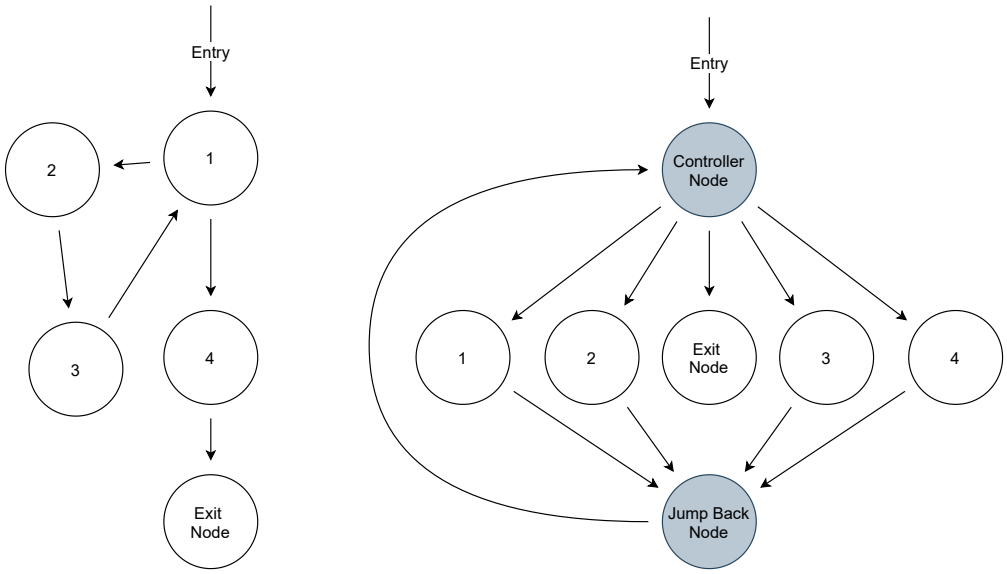


Figure 2.5.: An example of a program control flow graph with flattened control flow obfuscation. Original control flow graph (left) and flattened obfuscation graph (right).

2.5. Machine Learning

Machine learning has been increasingly replacing traditional algorithms over the last decade (Shinde & Shah 2018). It shifts the programming paradigm from hand-built programs to optimizations based on data. A number of scientific fields as well as the industry follow a trend that shows advantages of such approach. Some examples include self-driving cars and robotics (Voulodimos, N. Doulamis, A. Doulamis, *et al.* 2018), game playing (Silver, Huang, Maddison, *et al.* 2016), natural language processing (Goldberg 2015), ads and recommendations (Gomez-Uribe & Hunt 2016), Physics,

biology and medicine (Larrañaga, Calvo, Santana, *et al.* 2006). The machine learning field branches into several sub-fields. They can be broadly categorized into three categories.

Supervised Learning - A process of building a predictive model using a supervisory signal. The model is some function that maps input data $\{x_1, x_2, ..., x_N\}$ into a target variable y. The supervisory signal is a training dataset containing examples of correct mappings from $\{x_1, x_2, ..., x_N\}$ to $\{y_1, y_2, ..., y_N\}$. The model is optimized for approximating the supervision signal. When the target variable is categorical, the supervised learning task is called **classification**. For continuous target variable, it is called **regression**. Examples of classification tasks are handwritten digit recognition, transaction verification (valid or fraudulent), and SIP localization. Similarly, regression examples are weather temperature prediction and test score prediction.

Unsupervised Learning - Finding data point structure, grouping or patters without explicit supervision signal. The common examples are clustering Rokach & Maimon 2005 and probability density estimation (Ram & Gray 2011).

Reinforcement Learning - Approximating optimal playing strategy that maximizes some reward function in a dynamic game environment. An agent is put into this environment without explicit supervision signal of target actions, but it is rewarded when positive goals are achieved (defined by the reward function). The agent has to come up with a policy that maximizes the reward. This approach was successfully used to create the best go player in existence Silver, Huang, Maddison, *et al.* 2016.

In this work, we mainly focus on supervised learning since our task of SIP localization and the available data are well-suited for it.

2.5.1. Model Training Procedure

A supervised machine learning model can be represented as some function $f_\theta(x)$, where θ denotes model parameters and x contains input values. These parameters have a domain of all possible values, $\theta \in \Theta$. Different parameters will change the function behavior and the process of learning is to find the `optimal` values for θ, so that the model predictions $\hat{y} = f_\theta(x)$ best approximates y values. I.e., find the optimal model parameters θ^* with respect to some criteria, defined by the *loss* (L) function.

$$\theta^* = \underset{\theta \in \Theta}{\operatorname{argmin}} L(\hat{y}, y) = \underset{\theta \in \Theta}{\operatorname{argmin}} L(f_\theta(x), y)$$

Usually, the L (loss function) gives a score of how well the f_θ function (model) explains the training data, i.e how different y and $\hat{y}$ values are. However, the loss function does not estimate the model's generalization ability. It only shows how well the model performs on the data it has been optimized for. To solve this common problem,

the dataset is split into train and test sets. Model parameters are only optimized on the training set, and the test set is reserved to evaluate how well the model performs on unseen data points. This approach with some variations has become the de facto standard of machine learning projects.

For many complex machine learning models, given a sufficiently high number of model parameters θ, it is possible to find optimal θ^* values, so that f_θ is an ideal approximation and $\hat{y} = y$. However, this often leads to models that just *remember* the dataset. Similar to a lookup table and do not generalize at all. This problem, called overfitting (Dietterich 1995), is common in the machine learning field. Many countermeasures have been proposed against overfitting. They are usually specific to a particular model type that is trained. (Srivastava, Hinton, Krizhevsky, *et al.* 2014, Dietterich 1995).

2.5.2. Classification Performance Metrics

When evaluating the model on a test set, the most natural performance metric seems to be what fraction of class predictions did it get correct, i.e., *accuracy*:

$$\texttt{accuracy} = \frac{\texttt{correct predictions}}{\texttt{total predictions}}$$

Though this metric can be very informative in some cases (binary classification, balanced classes), it can sometimes be misleading. In particular, when true class labels in the dataset are very unbalanced (Caruana & Niculescu-Mizil 2004). The model that always predicts the most frequent class would have very high accuracy but is not very useful. To look at the full picture, we need to consider full *confusion matrix* (Caruana & Niculescu-Mizil 2004), consisting of 4 prediction cases:

- True Positives (TP) - number of positive test samples the model predicted correctly.
- True Negatvies (TN) - number of negative test samples the model predicted correctly.
- False Positives (FP) - number of positive test samples the model predicted incorrectly.
- False Negatives (FN) - number of incorrectly test samples the model predicted incorrectly.

The first letter (T, F) denotes whether the model prediction was correct, and the second letter (P, N) denotes the actual class of the sample. Using these notations

$$\texttt{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

We also use two other metrics to get a good insight into how well the model classifies each class.

- *Precision* - Also called positive predictive value, specificity, is what fraction of positively classified points were correct.
- *Recall* - Also called sensitivity, is the fraction of positive samples were classified correctly.

$$\texttt{precision} = \frac{TP}{TP + FP}$$

$$\texttt{recall} = \frac{TP}{TP + FN}$$

Usually, it is easy to tune the model to increase one of the *precision* or *recall* metrics at the cost of another. There is another metric that attempts to reconcile both aspects into a single value, keeping them somewhat balanced. It is derived from the generalized harmonic mean, plugging in precision and recall.

$$F_1 = \frac{\texttt{precision} + \texttt{recall}}{2 \times \texttt{precision} \times \texttt{recall}} = \frac{TP}{2 \times TP + FN + FP}$$

2.5.3. K Fold Cross Validation

In some cases, training data is scarce, and splitting out test the set has a big impact on overall model performance. Furthermore, it is important to evaluate if different train/test splits would lead to significantly different results. In such cases, a convenient replacement for a fixed split is K-fold cross-validation. The training process is done in k iterations. Each iteration splits the data differently and stores evaluation results. The Figure 2.6 describes the steps of such training process with $k = 5$. Cross-validation requires more computation but has the advantage of using all the data for training. Also, it provides variability information of model evaluation results.

In extreme cases, *leave one out* cross-validation can also be done. That means $k = N - 1$, where N is number of samples in the entire dataset.

2.5.4. Feature Engineering

Machine learning models usually work with the *vector* representations of data points. However, raw datasets are rarely in vector form. Because of this, a manual step of transforming the dataset into vector space is needed. This process is called feature

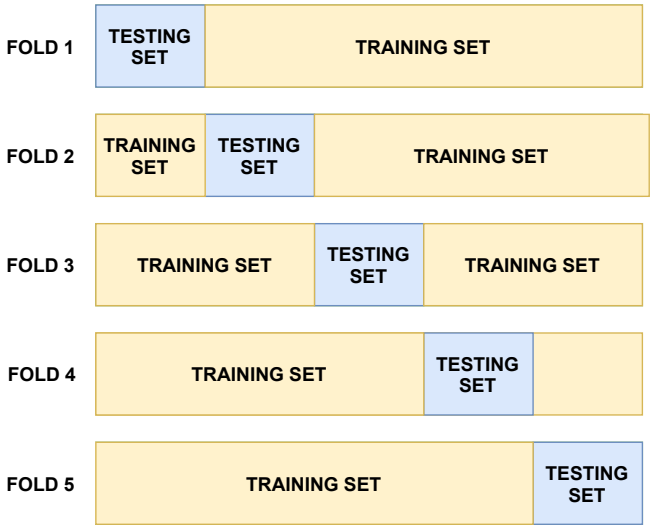


Figure 2.6.: K-Fold cross-validation process with $k = 5$. Each test split contains about %20 of the total data, leaving the other %80 for the training.

engineering, and it can have a big impact on model performance (Zheng & Casari 2018). One common feature engineering approach that we use in this work is called TF-IDF. It was initially used for information retrieval (Salton & McGill 1986), but was later found to work in other fields too (Aizawa 2003).

TF-IDF

This feature representation is from the information retrieval field. The problem is defined as follows: Given a search query `q = {t_1, t_2, ...}` that contains a list of terms t_i and a large set of documents `D = {d_1, d_2, ... d_N}`, rank documents by relevance to the query.

The first part of this notation is Term Frequency (TF). TF is just an occurrence count of the relevant term in the given document. Some terms, however, could be widespread in all the other documents as well. This phenomenon is especially apparent in natural languages. In any English document, the word "the" is very likely to appear. Frequent terms need to be discounted when measuring relevance. The discounting fraction is called Inverse Document Frequency (IDF). The final relevance score of term t, in document d is:

$$\texttt{TF}(t,d) = \frac{\texttt{No. terms in } d}{\texttt{Total no. terms in } d}$$

$$\texttt{IDF}(t) = \log\left(\frac{\texttt{Total no. documents} + 1}{\texttt{TF}(t,d) + 1}\right) + 1$$

$$\texttt{TF-IDF}(t,d) = \texttt{TF}(t,d) \times \texttt{IDF}(t)$$

The additional +1 values have smoothing functions to prevent zero division, except for the last $+1$ in `IDF(t)`, which is used not to discount terms appearing in all documents entirely.

To obtain vector representations of all the terms or documents, one would calculate a matrix of all (term, document) `TF-IDF` values and use columns or rows as term or document vector embeddings. Matrix dimensionality reduction algorithms are commonly applied here to get more compact representations.

2.5.5. Deep Learning

Deep learning is a sub-field that is only concerned with particular subsets of machine learning models - *Neural Networks*. They have a brain-like structure (very loosely) of neurons that are organized in layers. Networks with more than a single layer are called deep neural networks.

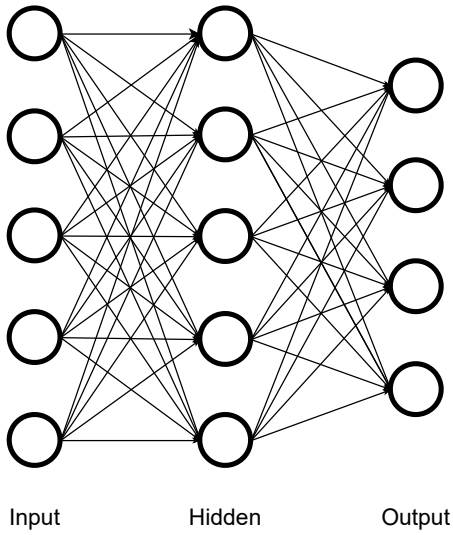


Figure 2.7.: Deep Neural Network with a single layer and four output neurons.

Figure 2.7 shows a structure of a deep network. Each neuron has its weights $\mathbf{w}$, represented by the incoming arrows. It takes in all the input numbers and calculates their linear combination with optional bias term b. The result of the this calculation is

then passed through a non-linear activation function, such as sigmoid $\sigma(x) = \frac{1}{1+e^{-1}}$ or Rectified Linear Unit $\texttt{ReLU}(x) = max(0, x)$

$$\texttt{neuron output } (x) = \texttt{Activation}(\sum_i w_i x_i + b)$$

Each neuron has its own weight vector w and bias scalar b. The stack of neuron outputs at layer n becomes the input to the next layer $n + 1$.

A deep neural network with a single hidden layer and non-linear activations like sigmoid is proven to be a universal approximator (Sonoda & Murata 2017). This means that given a sufficient number of neurons, there exist optimal neuron weights that approximate the neural network to any function over $\mathbb{R}^D$ with arbitrarily small ϵ.

Model Definition

One of the crucial parts of the recent success in the deep learning field is that the operation of the entire layer of neurons can be expressed as matrix multiplication and element-wise activation. This allows for easy and efficient implementation of the entire neural network operation called *forward pass*.

$$x \texttt{ is an input vector, } x \in R^D$$

$$h = ReLU(W_h x + b_h)$$

$W_h \in R^{K \times D}$ and $b_h \in R^K$. Consequently, $h \in R^K$

Final classification layer of neurons usually has *softmax* activation function. This non-linear activation constrains the output values of all neurons to sum up to one, resembling the probability distribution of prediction classes.

$$\texttt{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{j=n} e^{z_j}}$$

Where z_j denotes j-th neuron value before activation. In practice, neural network weights initially are randomized to small values (Thimm & Fiesler 1995). Afterward, weights should be modified during the training to improve model performance.

Loss Function

Common loss function measure for multi-class neural network classifier is *categorical cross-entropy loss*. It is a metric that originated from information theory and measures how different predicted probability distribution and actual distribution of classes are.

$$L(\theta, X) = \sum_{n=1}^{N} \sum_{c=1}^{C} [Y_{nc} \log \mathtt{NN}_\theta(X_n)_c]$$

In this definition, θ refers to model weights, N is total number of samples, C is number of classes, $Y_{nc} = 1$ if data point with index n is of class c and 0 otherwise. $\mathtt{NN}_\theta(X_n)_c$ is prediction output of class c on data point X_n.

This loss value is always positive and only becomes 0 when predictions exactly match the data.

Gradient Descent

Gradient descent is the most common version of neural weight optimization algorithms. It is an iterative process and iteration consists of three steps:

1. **Forward Pass** - The data is passed through neurons and the loss function $L(\theta, X)$ value is calculated.
2. **Backpropagation** - Starting from the last layer, goes in the backward direction and calculates the *gradient* of the loss function with respect to each neuron weight. It relies on differentiable loss function L and well-known *chain rule* to propagate loss values from the layer $n + 1$ to layer n.

$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)] \frac{du}{dx}$$

3. **Weight Updates** - Previous step saves the gradient (∇) values of each neuron weight. The update changes the weights slightly in the opposite direction of its gradient. α is a scalar called *learning rate* and is a training hyper-parameter.

$$\theta_{\mathtt{new}} = \theta_{\mathtt{old}} - \alpha \nabla L(\theta, X)$$

The significant strength of neural networks and gradient-based optimizations is that all three steps of the training process can be implemented efficiently, vectorized, and scaled (Scanzio, Cumani, Gemello, *et al.* 2010).

It is widespread to run training iterations on small subsets of the data called *batches*. Small batches are very useful under memory constraints and have been postulated even to improve trained model performance (Amir, Koren & Livni 2021).

Adam Optimizer

Adaptive Moment Estimation (Adam) optimizer (Kingma & Ba 2017) is an extension of gradient descent that adds adaptive learning rates. Instead of having one fixed, scalar, and hand-picked value α, it adapts the learning rate to each model weight. In addition, it uses an exponentially decaying average of past gradients (g_t) and gradient squares (g_t^2).

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The values of m_0 and v_0 are initialized to 0. This introduces bias which authors suggest to correct via:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2}$$

Then update rule becomes:

$$\theta_{\texttt{new}} = \theta_{\texttt{old}} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

The proposed default values for training parameters by authors are: $\beta_1 = 0.9$, $\beta_2 = 0.999$ and 10^{-8} for ϵ. Even though there is not much behind these numbers, they empirically show that Adam is favorable to other algorithms in practice.

Dropout

Dropout is a neural network regularization technique that is frequently used against overfitting (Srivastava, Hinton, Krizhevsky, *et al.* 2014) problem. This simple yet powerful approach has been shown to be very effective. During the training process, each neuron has its output set to 0 with probability p. This effectively turns off a neuron and sending input 0 from this particular neuron to the next layer. Intuitively, this reduces the reliance of the next layer on this particular neuron and "forces" it to find alternative signals.

2.6. Other Related Work

Software protection for tamper-proofing, obfuscation, and hardening (Abrath, Coppens, Broeck, *et al.* 2020) is an active area of research. The work by (Ahmadvand, Pretschner

& Kelbert 2018) reviews the taxonomy of various SIP techniques. They point out a general lack of research in understanding the resilience of existing integrity protection and obfuscation methods. (Wessel 2019) implemented pattern-matching and machine-learning-based attacks against protected & obfuscated programs. He discovered that SIP schemes without obfuscation are very vulnerable to such attacks. Also, increasing hardening complexity generally results in more resilience. However, their attack model is only based on pattern-matching and machine learning models with TF-IDF features. This feature representation could be the limiting factor in the analysis of (Wessel 2019). Traditional TF-IDF approach has long been outperformed by deep learning-based feature representations (Mikolov, K. Chen, Corrado & Dean 2013) (Le & Mikolov 2014) in natural language processing. This work suggests that TF-IDF, though effective, could be non-optimal for programs as well.

There are several program representation approaches for machine learning tasks. VenkataKeerthy, Aggarwal, Jain, *et al.* 2020 compile programs to LLVM IR and then model entities of IR as `(subject, predicate, object)` relationships. They demonstrate the effectiveness of such representation on heterogeneous device mapping and thread coarsening tasks. (Ben-Nun, Jakobovits & Hoefler 2018) define individual LLVM IR instruction space *instr2vec*. Program embedding is the aggregate of all the instruction vectors and *contextual flow graph* using recurrent neural networks (Jozefowicz, Zaremba & Sutskever 2015). This approach was demonstrated to be superior in computing device mapping, optimal thread coarsening performance estimation tasks. On a higher, source code level (Alon, Zilberstein, Levy & Yahav 2018) have demonstrated the ability to predict method name based on method body correctly. They first parse the body into an abstract syntax tree (AST). Then, represent it through the list of leaf pairs and tree paths between them. They call the embeddings of (leaf, path, leaf) triplets *path context vectors*. *Path attention model* is trained on these embeddings to generate the output method name. In contrast to previous approaches, *Code2Vec* operates on the high-level source code (initially Java), and support of other languages is limited.

To the best of our knowledge, using deep learning embeddings specifically for localizing software integrity protection is novel.

3. Design

We take a set of generic programs, compile them using various protection schemes and obfuscations. Then, we try to localize the blocks containing SIP checking instructions using machine learning. The general purpose of these experiments is to get an idea about the most effective approaches for defending and attacking SIP.

We implement a machine learning pipeline with common functionality between different research questions and extend it for specific experiments. Figure 3.1 shows a sequence of stages in this pipeline. It starts with the protected & obfuscated program data generation. Then data is preprocessed to prepare for the feature extractor stage. Then we train the model on some combination of features and finally evaluate the results. The evaluation stage yields results based on the choice made in previous stages of this pipeline. So, we can also make conclusions about how the choice of different preprocessing techniques, features, or model architecture impacts the SIP localization performance

Another advantage of using such modular architecture is extensibility for future work. This structure is meant to be a flexible playground for various experiments. Our code can easily be extended to other SIP research, out of scope for this work.

The following sections define the components of the pipeline and their role.

3.1. Protection & Obfuscation

3.1.1. Source Programs

In order to evaluate different approaches to attacking SIP schemes, we take a set of generic Command Line Interface (CLI) programs in their source code form. We do not make any assumptions about what tasks these programs accomplish, how they are implemented, or even their programming languages. We try to approximate the distribution of all possible programs that might be subject to SIP. To remain programming language independent, we use intermediate representation (LLVM IR) for our programs.

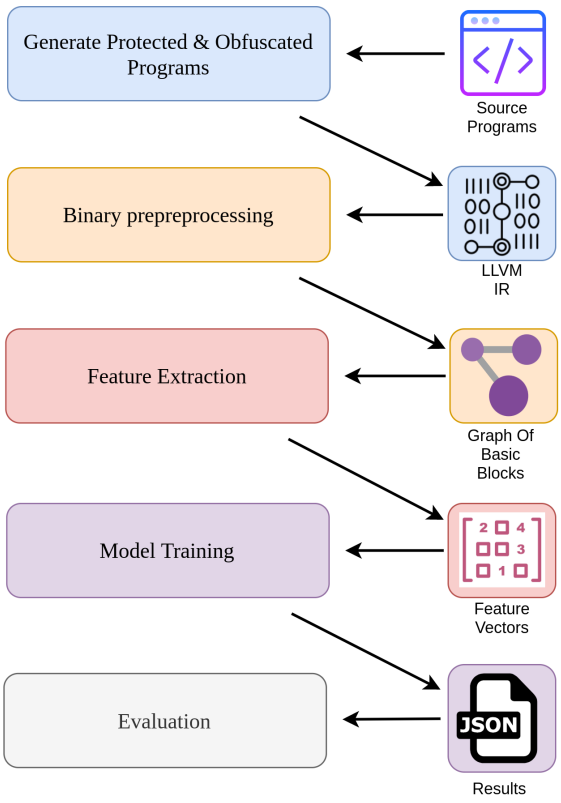


Figure 3.1.: Five stages and of our machine learning pipeline. Each stage is color-coded with its output.

3.1.2. Data Generation

This stage generates protected & obfuscated programs compiled to LLVM IR representation.

Integrity Protection

To generate protected binaries data for the ML model, we apply either no protection or one of the SIP schemes:

- Self-checksumming (SC) - section 2.3.1
- Oblivious Hashing (OH) - section 2.3.2
- Control Flow Integrity (CFI) - section 2.3.3

It is also possible to combine these SIP schemes, however (Ahmadvand, Fischer & Banescu 2019) found that this does not affect the performance of *SIP localizer* model. So, we also do not stack different protection techniques and focus on one at a time. Programs also get compiled in the process of protecting them. The exact compilation format and details are defined in the chapter 4.

Obfuscation

In addition to SIP, various obfuscation techniques have also been applied to our dataset. Wessel 2019 found that without such techniques, detecting SIP in a binary is achievable even with basic pattern-matching, concluding that obfuscation can be an essential part of SIP. The obfuscation transformations in our dataset are:

- Instruction Substitution (IS) - section 2.4.1
- Bogus Control Flow (BC) - section 2.4.2
- Control Flow Flattening (FLA) - section 2.4.3

To further increase the strength of obfuscation, the same technique is applied multiple times. We also use various compositions of different obfuscations.

3.1.3. Dataset

The process from the source programs to a dataset of protected binaries is shown in Figure 3.2. First, the program gets compiled with an embedded SIP scheme which yields a binary protected program. Then some combination of obfuscation is used to make SIP detection more difficult. We keep the labels of exactly which SIP and obfuscations were used for final analysis.

Repeating this process for all the source programs, SIP schemes, and obfuscation levels generate the training dataset for further processing in the pipeline 3.1.

3.2. Preprocessing

At the preprocessing stage of the pipeline, the input is an intermediate representation of a program that has been protected & obfuscated. Here, the goal is to do a cleanup of the raw data. The subsequent feature extraction stage requires cleanup. Preprocessing examples include disassembly, train/test split, instruction generalization, etc. All the preprocessing steps are detailed in chapter 4.

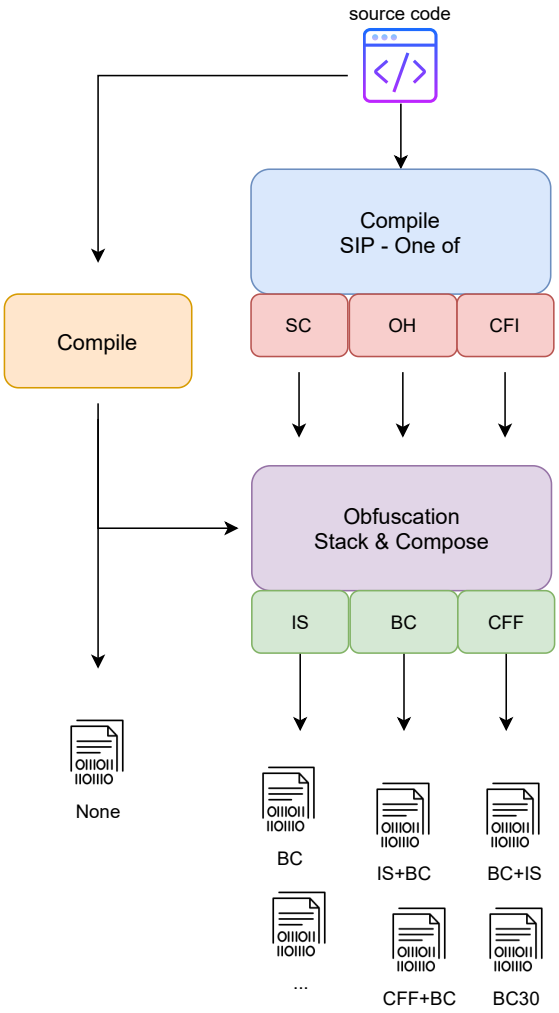


Figure 3.2.: Data generation procedure for a single program, showing the path from the source code to protected & obfuscated binaries. Only one of the SIP schemes is applied at most. Same obfuscation can be applied multiple times, as well as any combination.

3.2.1. Program representation

We chose to have a hierarchical representation of a program. At the highest level, the executable is a directed control flow graph of basic blocks. The basic block is a linear sequence of simple assembly instructions.

The major design decisions at this stage are:

- How to represent a single instruction?
- How to aggregate linear list of instructions into a block representation?
- How to aggregate graph of block representations into a program representation?

Concrete answers to these questions are given in the chapter 4. We define various feature extractors, which might answer these questions differently.

3.2.2. Intermediary language processing

Any assembly-like language instructions would contain memory addresses and constant values specific to the program or particular compiler options used to create it. Training a model on such a dataset could easily lead to a prevalent problem in machine learning called overfitting (Dietterich 1995). One of the ways to interpret this phenomenon is to say that overfitting happens when the training samples are specific and the model is capable enough to "remember" the data and not generalize well. While there are some heuristic approaches to prevent this problem, such as dropout in neural networks (Srivastava, Hinton, Krizhevsky, *et al.* 2014), they are usually model-specific. Thus, we still chose to solve this problem at preprocessing by making data look generic.

3.3. Feature Extraction

This section refers to taking the data and transforming it into a more suitable format for the Machine Learning model. Usually, data points are represented as multidimensional vectors in some abstract space. For classification tasks, the alignment of data points in this abstract space is more important than what precisely the axis represents. The main objective for a suitable feature extractor is to represent the data points differently to make it easier for the model to differentiate between them.

Each program in our dataset after the preprocessing step has two main components:

- Basic Blocks - sequence of generalized instructions
- Directed connections between basic blocks.

The feature representation for each basic block should capture the information about the kind of instructions it contains and its position to other blocks within the graph.

There are multiple approaches to feature extraction we would like to evaluate for our research questions. Thus, we implement different feature extraction components independently and combine them by concatenating vectors, as shown in Figure 3.3.

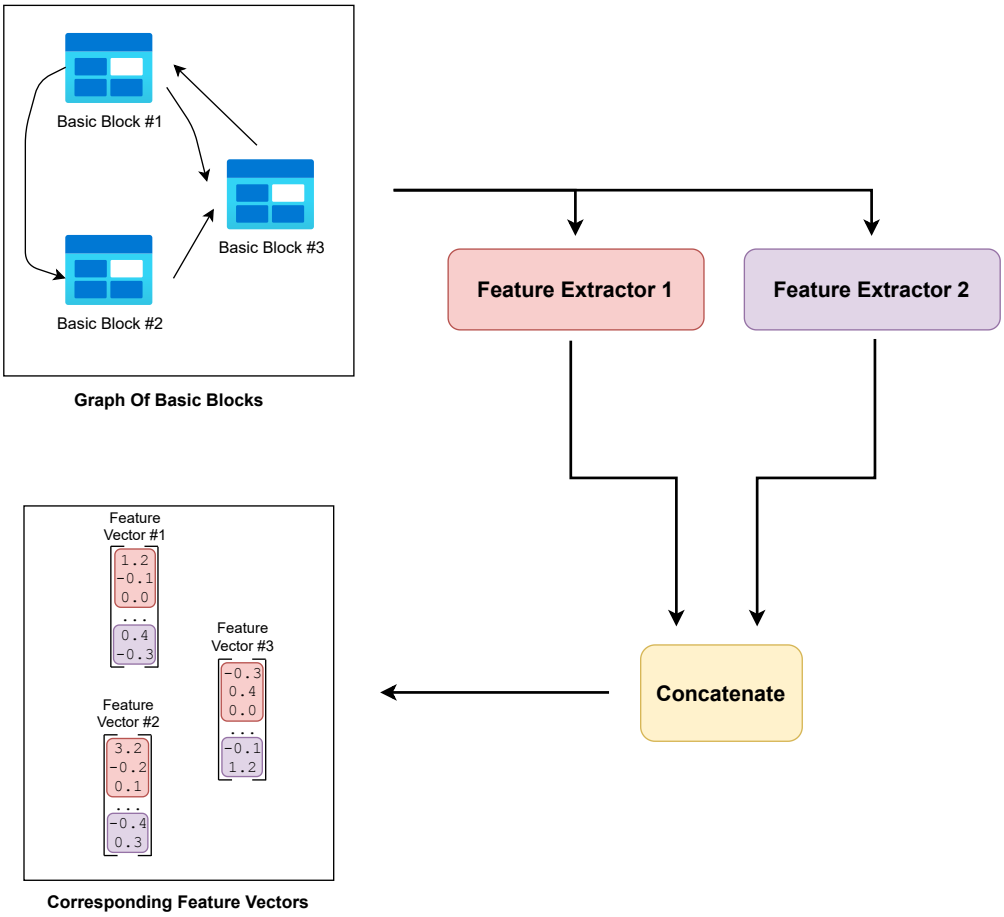


Figure 3.3.: The feature extraction from the graph of basic blocks using multiple independent feature extractors. Each basic block contains generalized instructions, together with control jumps to other blocks. Feature Extractors 1 & 2 capture different information about each. The final vector representation for each block is comprised of their concatenation.

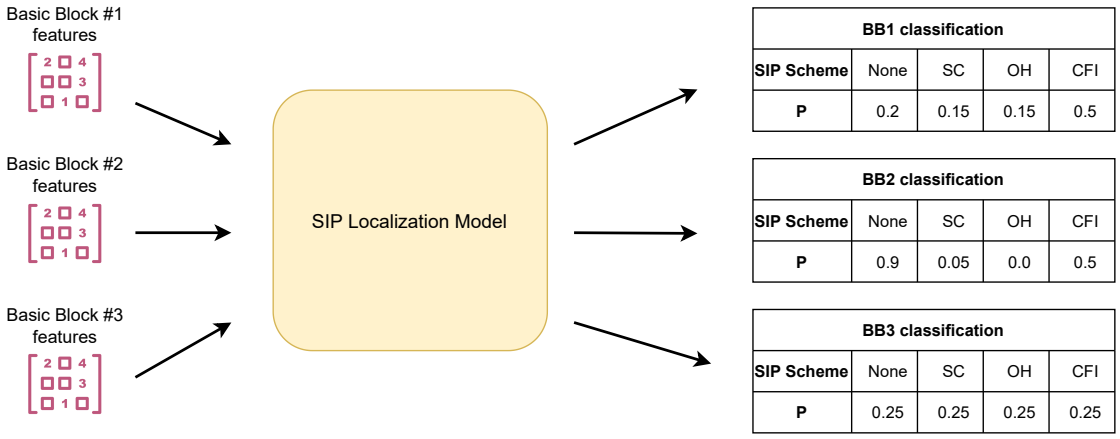
3.4. Machine Learning Model

Our model has a task of *Software Integrity Localization*. It takes in graph of basic block embeddings and classifies each with 1 out of 4 classes: *None*, *Self Checksumming* (SC), *Oblivious Hashing* (OH) and *Control Flow Integrity (CFI)*. As mentioned in the previous chapters, each program is either untouched or protected using only one out of these classes. The ideal model should predict the correct SIP scheme with a probability of

1.0, leaving 0 mass for other classes.

3.4.1. Basic Block Classification

SIP localizer model takes in the feature vector of each basic block and classifies them into protection categories. This process is depicted in Figure3.4. It is important to note that the classifier works with the graph of blocks. It can potentially consider features of all the neighboring blocks and not just one block at hand.



BB1 classification				
SIP Scheme	None	SC	OH	CFI
P	0.2	0.15	0.15	0.5

BB2 classification				
SIP Scheme	None	SC	OH	CFI
P	0.9	0.05	0.0	0.5

BB3 classification				
SIP Scheme	None	SC	OH	CFI
P	0.25	0.25	0.25	0.25

Figure 3.4.: An example SIP localizer model prediction. The input to the model is a feature vector for each basic block in the program. Prediction values per block sum up to 1.0

3.4.2. Single Model per Obfuscation

The original machine learning SIP attack by (Wessel 2019) was to train a new instance of the same model for each obfuscation class in the dataset. For example, the programs obfuscated with BC30 and BC100 instruction substitution would be trained and evaluated independently. This approach is valid for exploring how obfuscation techniques affect SIP analysis. A hypothetical attacker would only have a single program as a target, obfuscated with one particular obfuscation combination. Also, the experiments on differently obfuscated programs can be run in parallel. However, there are some potential disadvantages too:

- It is inefficient to train a new model from scratch on every obfuscation method.
- Bigger generic model can learn more complex rules for classification.

3.4.3. Model Training

For the task of graph node classification, we use GraphSAGE (Hamilton, Ying & Leskovec 2017) inductive model. We also considered graph convolutional neural networks (Kipf & Welling 2016), but we chose the inductive model since it is much more efficient and our computing resources are limited.

It is important to note that the dataset split happens at the program level, so the model is evaluated on the programs it has not seen in the training dataset.

To get the information about the variability of our results, we decided to run each training process multiple times and use k-fold cross-validation for evaluation.

3.5. Results

The final part of our machine learning pipeline contains a result generation script. The model training process produces artifacts describing the intermediate performance values and final test results. We aim for complete and easy reproducibility of our results. Thus, all the experiment results, tables, and charts given in the chapter 5 are generated using automated scripts and interactive notebook (Kluyver, Ragan-Kelley, Pérez, *et al.* 2016) files.

4. Implementation

The following chapter describes all the implementation details of our pipeline and experiments. It repeats the same sub-section structure as in the chapter 3 - Design. The supplemental code, together with the entire dataset and pre-trained model weights, is publicly available on our Github repository[1].

Project Directory Structure The project directory contains all the scripts used in this work and is publicly available as a git repository on Github. It includes an entire machine learning pipeline and interactive notebook-style scripts for data examination and plot generation. For the code outside of the ML pipeline, we generally use web-based interactive Jupiter notebook IDE (Kluyver, Ragan-Kelley, Pérez, *et al.* 2016). The aim is to enhance the reproducibility and transparency of our experiments and pave the way for future work.

- **diagrams/** - contains xml formatted definitions for all the diagrams used in this work. The format is not human-readable, but can be examined and modified through free online platform - (draw.io).
- **sip_vs_pipeline/** - directory containing ml pipeline with 5 stages
  - `data_generation`
  - `preprocessing`
  - `feature_extraction`
  - `model_training`
  - `evaluation`
- **notebooks/** - stores all the interactive jupyter notebook files

4.1. Data Generation

This section defines the data generation procedure, including the script usage and output directory structure.

[1] https://github.com/tum-i4/sipvsml

4.1.1. Data Generation Script

The data generation script definition is given in listing 4.1.1. The script uses docker container[2] to take care of all the OS and library dependencies. It only has a few CLI parameters:

- `<output_dir>` - mandatory argument of the path to the output directory. For safety, it has to either be empty or non-existent.
- `--force` - An option to force write if the output directory is non-empty.

```
$ python3 ./dataget/generate_dataset.py [-h] [--force] <output_dir>
```

Listing 4.1: Data generation script synopsis. The square brackets denote optional arguments. Otherwise, all arguments are mandatory.

4.1.2. Data Directory Structure

Training Data Structure Data generation script creates a nested directory structure, with all the combinations of source program files, obfuscations and protections. The visual representaiton of this structure is given in Figure 4.1. The root of the dataset directory is named **LABELED-BCs**. It contains has 3 sub-directories

- **simple-cov** - contains source programs from the simple dataset and simpler obfuscations
- **simple-cov2** - the same simple set of programs as in simple-cov, but with heavy obfuscations
- **mibench-cov** - heavy obfuscations identical to simple-cov2, but the source programs are more from mibench dataset.

Each obfuscation sub-directory contains protected, obfuscation, and LLVM-IR compiled programs, with *.bc* extensions. Generally, all pipeline components operate on the obfuscation directories independently and in parallel.

4.2. Preprocessing

The data generation step from the data generation section creates a set of LLVM IR program representations protected and obfuscated with various combinations as given

[2]https://github.com/Megatvini/smwyg-artifact/blob/master/Dockerfile

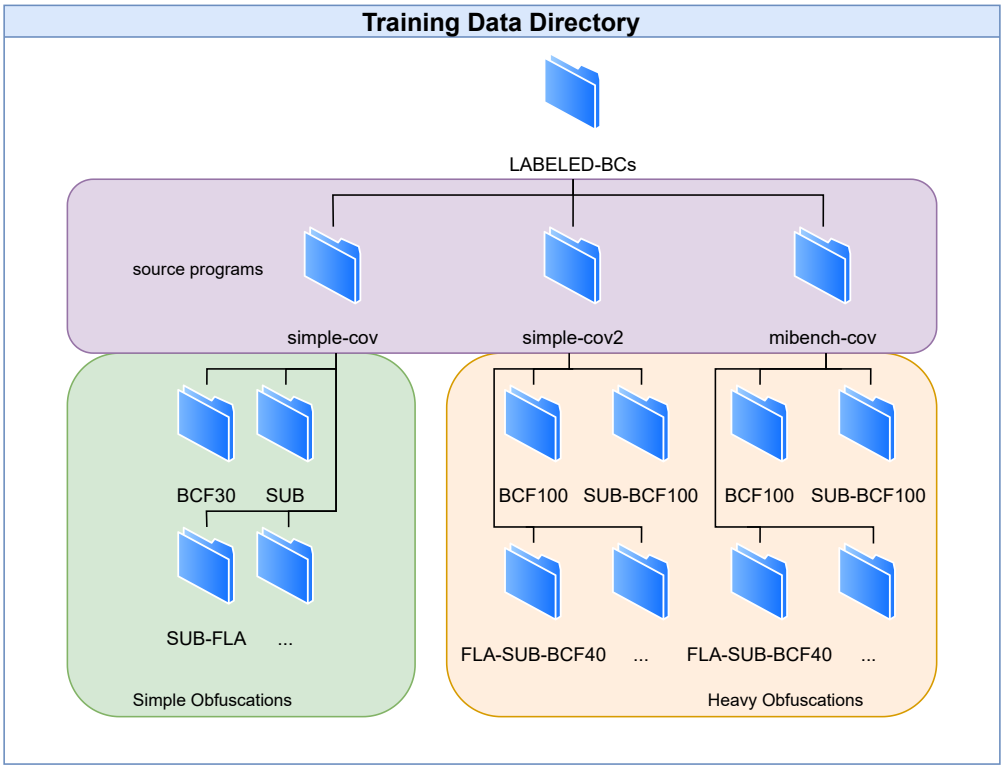


Figure 4.1.: The directory tree of protected and obfuscated programs. Each obfuscation sub-directory contains all the related source programs with the particular obfuscation combination and one of the protections applied. The simple-cov2 directory has the same source programs as the simple-cov dataset but heavier obfuscations like mibench-cov.

in Figure 4.1. The following step in the pipeline is to do preprocessing of raw *.bc* files. In the name of extensibility through modularity, we separate all the preprocessing steps into independent processes. The main preprocessing script *preprocessing/process_binaries.py* takes in a list of preprocessing steps as an argument and executes them. The preprocessing is done independently and in parallel for each obfuscation directory.

The listing 4.2 describes the synopsis for the preprocessing script. It takes in several arguments:

- `--preprocessors` - Allows choosing between any combinations of preprocessing steps to run. At least one preprocessor must be chosen.
- `--run_sequentially` - The optional flag to run the code sequentially. It uses less memory and helps to debug.

- `--dataset` - Optional argument to only run the script on a particular dataset. E.g., simple-cov.
- `<labeled_bc_dir>` - Mandatory argument pointing to the LABELED-BCs directory.

```
$ python3 ./preprocessing/process_binaries.py [-h] \
   [--preprocessors {\
      disassemble_bc llvm_sip_labels remove_ll_metadata pdg \
      code2vec general_ir k_fold_split \
   }] \
   [--run_sequentially] \
   [--dataset DATASET] \
    <labeled_bc_dir>
```

Listing 4.2: Pre-processing script synopsis. The square brackets denote optional arguments. Curly brackets denote options to choose from. Otherwise, all arguments are mandatory.

The list of all available preprocessing steps are:

- **pdg** - extracts program dependence graph data for each program.
- **general_ir** - generalizes instructions for on IR2Vec feature extractor. (Venkata-Keerthy, Aggarwal, Jain, *et al.* 2020)
- **code2vec** - extracts training files for code2vec model feature extractor.
- **disassemble_bc** - disassembles binary LLVM IR files into text representation *.ll* files.
- **k_fold_split** - generates static dataset splits for k-fold cross validation and writes the splits into json files.
- **llvm_sip_labels** - reads *.bc* file metadata and extracts basic block protection sip labels, which can be one of the (NONE, SC, OH, CFI).
- **remove_ll_metadata** - removes the LLVM metadata content from the *.ll* files, such as sip labels or other sip composition framework side-effects.

Program Dependence Graph

This step looks at all the raw *.bc* files in a particular obfuscation sub-directory and generates two large files:

- **blocks.csv.gz** - comma-separated and compressed file, which assigns an unique id to each basic block in protected and obfuscated source programs. This unique id is used for all the subsequent references to the basic blocks. For instance, when mapping basic block features.
- **relations.csv.gz** - this file defines the directed multi-graph for basic blocks identified in the **blocks.csv.gz** file. It contains three columns: source, destination, and the relation type. The types of relations between blocks that we capture can be both explicit and implicit (Burke & Cytron 2004). They extracted from Control-Flow-Graphs (CFG) and strongly connected components (SCC). (Nuutila & Soisalon-soininen 1995).

In addition to basic block identifiers, **blocks.csv.gz** file contains Static Value Flow (SFV)[3] analysis features as well (Sui & Xue 2016). It is an open-source LLVM code analysis tool that would be more suitable to place at the feature extraction stage considering our pipeline. Nevertheless, since our implementation nuances make it easier to include the SVF together with the data generation process, we implement it as a preprocessing step.

Generalized Ir2Vec Instructions

In order to represent our basic blocks with feature vectors from the IR2Vec model (VenkataKeerthy, Aggarwal, Jain, *et al.* 2020), we need to repeat their preprocessing steps exactly. All the instructions must be generalized into tuples of operation code, type, and a list of arguments. An example of concrete instruction generalization is given in table 4.2. The operation code stays unchanged, but the type and arguments are generalized as shown in the table 4.1. A generic representation categorizes all the specific identifiers like memory addresses or function names.

Identifier	**Generic representation**
Variables	VAR
Pointers	PTR
Constants	CONST
Function names	FUNCTION
Address of a basic block	LABEL

Table 4.1.: Ir2Vec - Mapping identifiers to generic representation
Generic representations of LLVM IR instruction components used by Ir2Vec model

[3]https://svf-tools.github.io/SVF/

LLVM IR Instruction	Generalized Instruction
%3 = alloca i64*, align 8	alloca pointerTy constant
br i1 %43, label %44, label %assert.1	br voidTy variable label label
%130 = icmp ne i64 %128, %129	icmp integerTy variable variable
call void @exit(i32 1) #5	call voidTy constant function
%151 = add nsw i64 %149, %150	add integerTy variable variable

Table 4.2.: IR2Vec - Instruction Generalization
LLVM IR instructions taken directly from the dataset and corresponding generalized representations after Ir2Vec preprocessing pass. Each generalized instruction contains an operation code, type, and a list of arguments.

The work by (VenkataKeerthy, Aggarwal, Jain, *et al.* 2020) does feature a repository with all the necessary code for getting program-level embeddings of LLVM IR code. However, we did not find a direct way to get basic block-level embeddings that would preserve our identifiers in their codebase. Thus, we resorted to writing our own implementation for LLVM IR parsing and generalizing according to their specifications and IR language definition from https://llvm.org/docs/LangRef.html. To verify the correctness of our implementation, we matched the program-level output of the Ir2Vec repository with a small subset of our dataset. The test code, together with our parser implementation, is given in our public repository - https://github.com/tum-i4/sipvsml/blob/master/data_scripts/ir_line_parser.py.

The Program Dependence Graph preprocessing step does not generate new files. It adds a new column to **blocks.csv.gz** file containing a list of generalized instructions for each basic block.

Code2Vec Preprocessing

This stage prepares training data for the Code2Vec model that will be later used to extract block-level code2vec features. We extend the Code2Vec repository[4] to add a new source language - LLVM IR. Originally, Code2Vec was meant for high-level programming languages such as Java and C#, and it was trained to predict method names from method body source code. We replace the method name with the basic block sip label and the body by its instruction list. We found that the Code2Vec model is suitable for LLVM IR as well.

The Code2Vec model (Alon, Zilberstein, Levy & Yahav 2018) operates on *Abstract Syntax Tree* (AST) representations of the source code. It is the most complex preprocessing

[4]https://github.com/Megatvini/code2vec

step in our implementation, consisting of multiple stages.

- **LLVM IR EBNF Parser** - We use an open-source extended Backus–Naur form (EBNF) grammar[5] and a Textmapper[6] tool to generate a parser library written in Go. Our implementation[7] of the LLVM parser is using this library to print out pre-order traversal of the AST. The grammar is written for LLVM IR version 10. Since we were unable to find grammar definitions for newer versions, we are always using version 10 of the suite of LLVM tools. The example format is displayed on Figure 4.2.
- **SIP Label Extraction** - This is the same as label extraction preprocessing step (section 4.2), but we implement this separately as part of the Code2Vec extension.
- **Tree Traversal** - From the AST of a program, we traverse all the basic block nodes in the same order as they appear in the source *.ll* file. For each block, we generate leaf pair paths according to Code2Vec model specifications. Each basic block is represented as a list of target labels (SIP protection scheme) and all possible leaf pair paths it contains. To keep number of paths reasonable for large blocks, the extractor script limits the maximum path *length* and *width*. We keep default values *8* and *2* respectively from original Code2Vec Java extractor implementation.

The Code2Vec creates one file per protected & obfuscated program with `.raw_c2v.gz` extension. It contains all the basic block leaf pair path representations.

Dissasemble BCs

This simple preprocessing step just iterates over all the *.bc* files and disassembles them, generating corresponding *.ll* file. It uses LLVM version 10, since at this time it is the latest version for which we found EBNF grammar available online. The example command for disassembling *anagram-BCF.bc* program from *simple-cov* dataset with *BCF30* obfsucation would be:

```
$ llvm-dis-10 LABELED-BCs/simple-cov/BCF30/anagram-BCF.bc
```

This command automatically creates a disassembled file in the same directory with the same name, but with *.ll* extension. In this case *./LABELED-BCs/simple-cov/BCF30/anagram-BCF.bc*

[5]https://github.com/llir/grammar/blob/master/ll.tm
[6]https://github.com/inspirer/textmapper
[7]https://github.com/Megatvini/ll

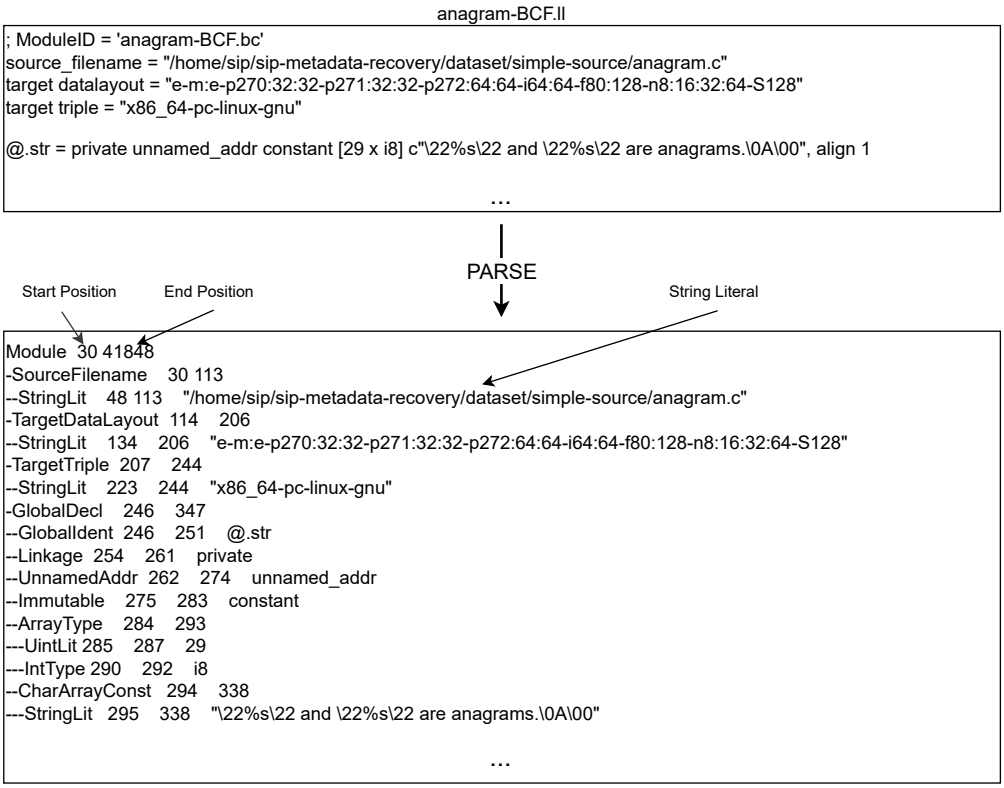


Figure 4.2.: An example of LLVM IR parsing process. The first box shows the top few lines of the original *.ll* file. The second box contains the output of our parser. It is a pre-order traversal of the parse tree. The number of beginning hyphens indicates the node depth. Each line contains a node name, followed by the corresponding starting and ending positions from the input. Leaf nodes also contain literal strings for convenience. The path extraction is detailed in Figure 4.3.

K-Fold dataset split

The `k_fold_split` statically defines k train/test splits for each obfuscated & protected sub-directory. The k-fold training process is shown in Figure 2.6.

To balance the trade-off between pipeline speed and reliable variance measurements, we chose to use k-fold cross-validation with $k = 5$. For every fold, the model will be trained on %80 of the data and evaluated on the other %20.

It is important to note that we always do train/test split at the *program* level. The more straightforward version of the split would be to do it at the basic block level.

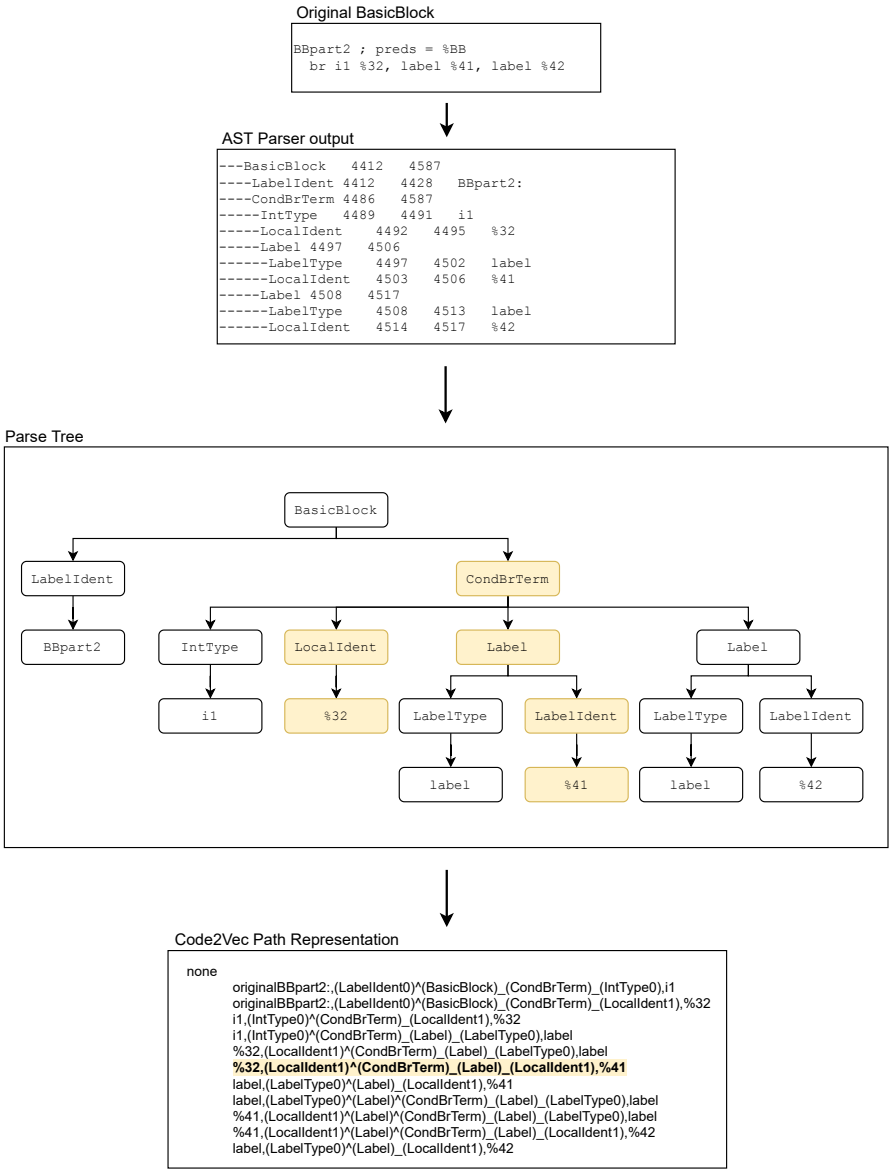


Figure 4.3.: An example of simple basic block parsing and leaf pair paths extraction. The highlighted path triplet (<source,path,dest>) corresponds to the highlighted nodes in the basic block parse tree. The label **none** indicates that this particular basic block does not contain integrity protection check.

However, we argue that this would lead to invalid results. During the test phase, the model might have already seen an unprotected version of the same basic block within the same source program during the training.

This preprocessing step creates a new `folds/k_fold_*` directory inside each obfuscation directory. All `k_fold_*` directories are further subdivided into *train* and *val* sub-directories which contain *programs.json* files, listing the program names that belong to that particular split.

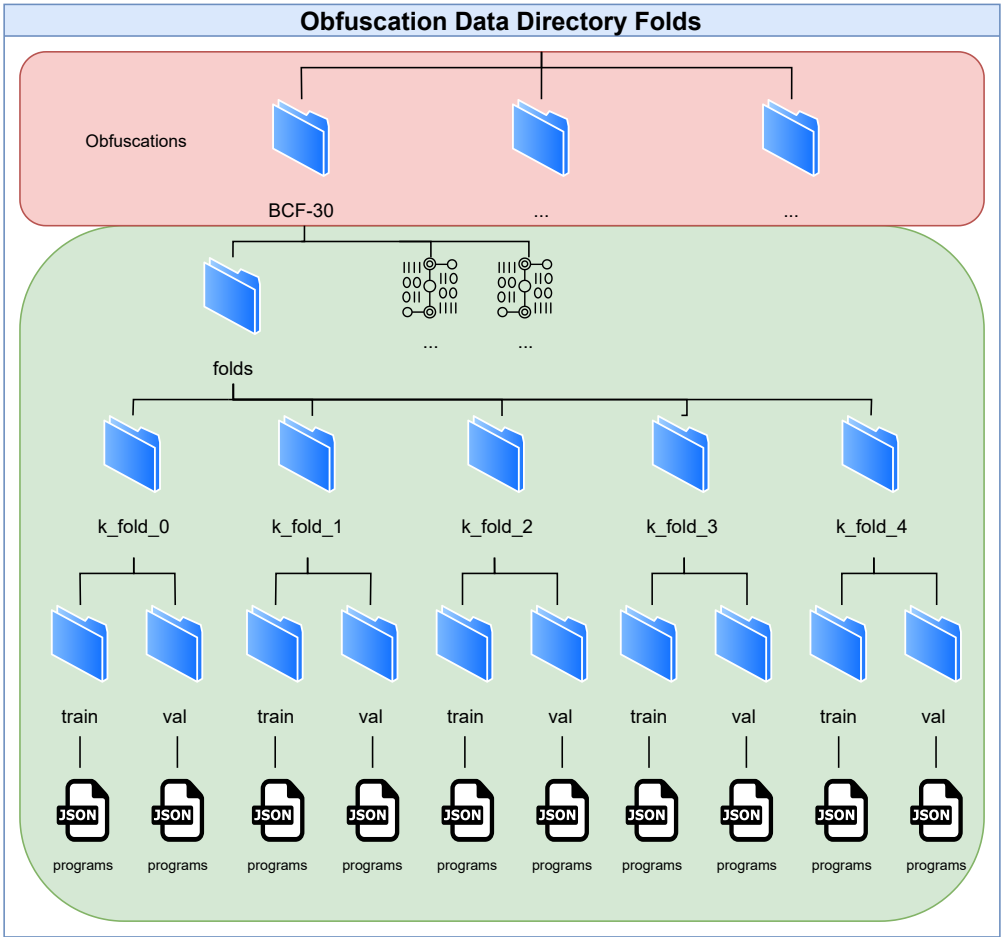


Figure 4.4.: The directory tree of a single obfuscation sub-directory after k-fold split preprocessing step has been applied

SIP labels extraction

The labels for SIP model training are contained in the metadata of *.bc* files. The label extraction preprocessing step reads these files and extracts labels mapped to each unique basic block identifier. Since these labels are accessed very frequently throughout the pipeline life-cycle, it is more efficient to store them in separate files rather than invoke LLVM reader tools all the time.

This step is implemented as an LLVM pass in C++. All it does is to visit all the basic blocks and print out a single line for each, keeping their order. The line contains tab-separated triplet:

- **Basic Block path** - path to the file containing the basic block concatenated with the function name and the block label. For example: *LABELED-BCs/simple-cov/BCF30/anagram-BCF.bcmain%2*. Using the full path is essential because the same source programs are repeated in many obfuscation sub-directories.
- **Basic Block unique ID** - a hash value of the basic block path. This unique identifier is used exclusively throughout the whole project to refer to basic blocks.
- **Basic Block Label** - label indicating whether this particular block is a part of the SIP scheme. It has to be one of (none, sc, oh, cfi).

The labeling LLVM pass can also be run independently outside of the preprocessing pipeline. For instance, the command for *anagram-BCF* would be:

```
$ opt-10 -load ./libModuleLabelling.so \
    -bc-file-path="LABELED-BCs/simple-cov/NONE/" \
    -legacy-module-labelling \
    -disable-output ../anagram-BCF.bc
```

This command outputs a file with the *.sip_labels* extension, containing basic block labels. The *-bc-file-path* argument is needed because the module itself does not contain any information about where it is located.

LLVM IR metadata Removal

The SIP implementation we use by (Ahmadvand, Fischer & Banescu 2019) has a side effect of adding metadata to generated programs. This information is specific to the implementation. The target labels for the model are included in the metadata as well. Thus, the metadata needs to be removed prior to using programs as training/testing data. Concrete examples are given in table 4.3.

Original IR Line	**Stripped IR Line**
store i32 %56, i32* %16, align 4, *!data_dep_instr !6, !argument_dep_instr !9, !control_dep_instr !11*	store i32 %56, i32* %16, align 4
%58 = load i32, i32* @y.2, align 4	%58 = load i32, i32* @y.2, align 4
br label %66, *!data_indep_instr !8, !argument_dep_instr !9, !control_dep_instr !11*	br label %66
call void check_anagram0(i32* %16), *!data_indep_instr !8, !argument_dep_instr !9, !input_dep_block !10, !control_dep_instr !11*	call void check_anagram0(i32* %16)

Table 4.3.: Examples of LLVM IR lines taken from the training data and corresponding versions of the same lines without the metadata.

4.3. Feature Extraction

The objective of this stage is to reconcile the preprocessed data into a fixed-size floating-point vector representation. Each feature vector has to be mapped to a basic block identifier from the preprocessing stage. The model will only get to see the feature vector of a block and directed block relations. So, it is important to capture as much relevant information as possible at this stage. Similar to preprocessing stage, we take a modular approach, separating the extraction of different feature types and running them independently. The vectors representing different features can be stacked together, possibly capturing more information from the preprocessed data.

This stage generates `<feature_name>.feature.csv.gz` file for each feature representation described below. There is one training and one testing feature file per k-fold split. They reside in `folds/k_fold_<i>/train` and `folds/k_fold_<i>/val` directories respectively. The feature extractor process is implemented in such a way that it will not overwrite existing files. If the output file already exists, the computation will be skipped.

During the model training, one or several features can be used. Combining feature vectors is trivial. They are just concatenated into a larger fixed-size vector. The table 4.4 shows vector sizes of different feature vectors discussed in the following sections.

Feature Name	Feature Vector Dimensions
PDG	64
TF_IDF	200
IR2VEC	300
CODE2VEC	384

Table 4.4.: Feature vector sizes of four different features that we extract from the pre-processed dataset.

```
$ python3 ./feature_extraction/extract_features.py [-h] \
    [--path_to_ir2vec_vocab PATH_TO_IR2VEC_VOCAB] \
    [--feature_extractor {code2vec,tf_idf,ir2vec,pdg}] \
    [--run_sequentially] \
    [--max_workers MAX_WORKERS] \
    [--dataset DATASET] \
    <labeled_bc_dir>
```

Listing 4.3: Feature extractor script synopsis

4.3.1. PDG

The implementation of Program Dependence Graph (PDG) features we take from the existing work by (Sui & Xue 2016). This is the most straightforward baseline feature extraction step in our work. The implementation uses the LLVM Pass interface to generate `.csv` files. Source code is published on Github[8]. It produces 63-dimensional integer vector representation for blocks. PDG contains static code analysis, which utilizes SVF (Sui & Xue 2016) library to do instruction pointer analysis.

4.3.2. TF-IDF

The feature extractor for TF-IDF algorithm produces a 200-dimensional floating-point vector for each block. We use the implementation from the open-source module module (`sklearn.feature_extraction.text.TfidfTransformer`) from the `Scikit-learn` (Buitinck, Louppe, Blondel, *et al.* 2013) python package. Dimension for the TF-IDF feature vector is a hyper-parameter that can be tuned for optimal results.

We generate the text corpus for the extraction using an aggregate of all the basic block instructions in the dataset. Before creating the corpus, we first make some instruction generalizations, similar to Ir2Vec preprocessing. Generalization removes all the specific

[8]https://github.com/mr-ma/program-dependence-graph/blob/cleanup/lib/Debug/PDGCSV.cpp

identifiers, labels, and memory addresses that are specific to a particular program. Also, the metadata may contain the SIP label. The example of IR cleaning is given in table 4.5.

Original IR Line	**Cleaned IR Line**
%3 = alloca i32, align 4	VARo = alloca i32, align4
%13 = inttoptr i64 %12 to i8*	VARo = inttoptr i64 VARo to i8*
br label %18	br label VARo
%61 = load i32, i32* @y, !sc_guard !36	VARo = load i32, i32* REFo

Table 4.5.: Examples of LLVM IR lines taken from the training data and corresponding cleaned versions for the TF-IDF corpus.

4.3.3. IR2Vec

The IR2Vec (instruction to vector) feature extractor is built on the neural LLVM IR embedding work done by (VenkataKeerthy, Aggarwal, Jain, *et al.* 2020). Their paper, together with the source code, is freely available. However, a significant modification to their source code would be needed to include the Ir2Vec model in our pipeline as a feature extractor. This is because the CLI of Ir2Vec executable only takes in a complete `.bc` file. Thus, we chose to re-implement the inference phase of their model by following their publication carefully. We also use the pre-trained seed embeddings vocabulary and do not do any training of our IR2Vec model. The following sections describe our IR2Vec extractor implementation.

Ir2Vec Basic Block Embedding To obtain a basic block embedding $\boldsymbol{B}$, we take the embeddings of individual instructions $\boldsymbol{I}_l$ and sum them up. So, if a block contains L instructions - ($\boldsymbol{I}_1$, $\boldsymbol{I}_2$, ..., $\boldsymbol{I}_L$) then the block embedding would be:

$$\boldsymbol{B} = \sum_{l=1}^{L} \boldsymbol{I}_l$$

Consequently, the block embedding dimension matches the instruction embedding dimension. The embedding of individual instruction is calculated using the generalized form (Section 4.2).

Ir2Vec Instruction Embedding Generalized instruction $\boldsymbol{I}_l$ has an operation code, type and a list of arguments. Let's denoted these with O_l for an OpCode, T_l for Type and $A_l^1, A_l^2, ..., A_l^i$ for arguments. Then, the instruction embedding vector definition is:

$$\boldsymbol{I}_l = w_0 \boldsymbol{V}[O_l] + w_t \boldsymbol{V}[T_l] + w_a \sum_{l=1}^{L} \boldsymbol{V}[A_l]$$

Where $\boldsymbol{V}[*]$ denotes a seed embedding vocabulary lookup, which maps all possible instructions and generalized labels to pre-trained and fix-sized embeddings. We take the seed embeddings lookup dictionary from the original IR2Vec repository[9]. We could train our own model and obtain vocabulary specific to SIP task. However, we found that existing embeddings work rather well too.

The parameters w_0, w_t, and w_a are just empirically defined scalar hyper-parameters, which we leave unchanged from the original Ir2Vec implementation.

$$w_0 = 1.0$$

$$w_t = 0.5$$

$$w_a = 0.25$$

The dimension of Ir2Vec block embedding is defined by and equals to seed embedding vector dimensions. The vocabulary we use has 300-dimensional embeddings. Hence, Ir2Vec features for each basic block are 300-dimensional.

4.3.4. Code2Vec

The feature extraction process from the Code2Vec (Alon, Zilberstein, Levy & Yahav 2018) model is significantly more complex than others. Code2Vec model was initially meant to embed high-level languages such as Java. We extend it to work with our protected and obfuscated LLVM IR programs. Thus, we can not use any of the training data or pre-trained models available online. We have prepared training data for our Code2Vec model, trained on our protected & obfuscated dataset to obtain LLVM IR vector embeddings in the preprocessing stage. The feature extraction process takes the training data, trains the Code2Vec model, and uses it to get embeddings for training and test data. This process is shown in great detail in Figure 4.5. For the sake of completeness, the diagram also includes preprocessing data flow.

We make heavy use of the original Code2Vec codebase and only make a few modifications to the data preprocessing script to make it work with the LLVM IR language. We add a new `LLIRExtractor` directory on top of the original Code2Vec implementation, which keeps the same interface as their `JavaExtractor` had. Also, the target label becomes the SIP scheme instead of a method name. We keep the embeddings dimension size to 384.

[9] https://github.com/tum-i4/sipvsml/blob/master/sip_vs_pipeline/feature_extraction/ir2vec_seed_embeddings.txt

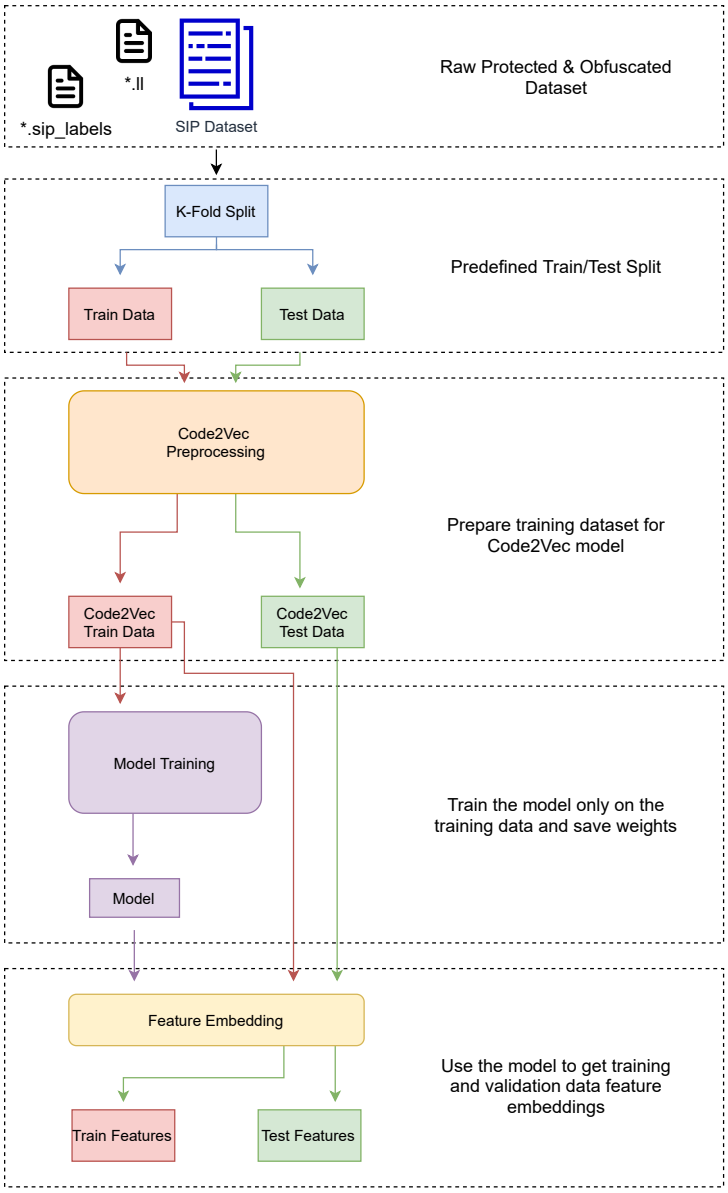


Figure 4.5.: Top to bottom: The process of extracting Code2Vec features from a protected and obfuscated dataset. First, the data is split according to the predefined train/test split. Then pre-processing script prepares training data for our own Code2Vec model. Already trained model is then used to get both train and test data embeddings.

4.4. Graph Neural Model

Once all the feature vectors are extracted from the data, the model training stage can be executed. The feature vectors have a straightforward and generic format. We have a set of interconnected basic blocks, where each basic block has a feature vector embedding and ground truth label associated with it. The feature vector can contain only a single or a concatenation of 4 options - (`pdg`, `tf_idf`, `ir2vec`, `code2vec`). The listing 4.4 describes the synopsis of the model training script. When using a combination of features, the order *does not* matter. The resulting vectors that the model uses will be a permutation containing the same information.

The objective of our model is to correctly predict the basic block label given the embeddings and connections. This module is designed to be easily extended to using other machine learning models at the training stage. However, for our research questions, a single neural model is sufficient. Thus, there is only one choice for the `--model` argument in the model training script (4.4). The model architecture is described in the following section 4.4.1

```
$ python3 ./model_training/train_all_models.py [-h] \
    --use_features {pdg,tf_idf,ir2vec,code2vec} \
    [--model {graph_sage}]
    [--num_processes NUM_PROCESSES]
   <labeled_bc_dir>
```

Listing 4.4: Model training script synopsis

It is important to note that every individual source dataset and obfuscation gets its model instance, trained on that particular obfuscation only. Hence, the script name `train_all_models.py`.

4.4.1. GraphSAGE

The graph embedding model we chose for our experiments is GraphSAGE (Sample and Aggregate) introduced by (Hamilton, Ying & Leskovec 2017). Due to the inductive approach, it is more efficient than typical existing Graph Convolutional Networks (GCN) (Kipf & Welling 2016) used for node labeling. Figure 4.6 shows a process of obtaining node embeddings based in the graph based on the sample & aggregate approach.

To complete the SIP localizer (figure 3.4), we take the node (basic block) embeddings and pass them through an additional softmax layer with four SIP classes (NONE, SC, OH, CFI).

The model has several hyper-parameters, which we choose empirically and keep constant between different obfuscations k-fold instances. Our guiding principle for choosing hyper-parameters is to keep the computational requirements for training and evaluation low while attaining good model performance.

- `Layer Sizes = [80, 80]` - We set the number of hidden layers to 2, each with the dimension size 80.
- `Dropout = 0.5` - High dropout often helps with the generalization gap problem (Keskar, Mudigere, Nocedal, *et al.* 2016), but at the cost of slower training convergence. Our training process converges relatively quickly, so we set a high dropout value of 0.5.
- `Aggregator Function = MeanAggreator` - The aggregator function as defined by (Hamilton, Ying & Leskovec 2017) takes in a list of sampled node neighbor vectors and aggregates it to a fix-sized vector. This vector is then passed through a single neural layer with some activation function. We choose the MeanAggreator simplest aggregator available for the GraphSAGE model in and according to results from (Hamilton, Ying & Leskovec 2017) it performs nearly as well as more complex alternatives.

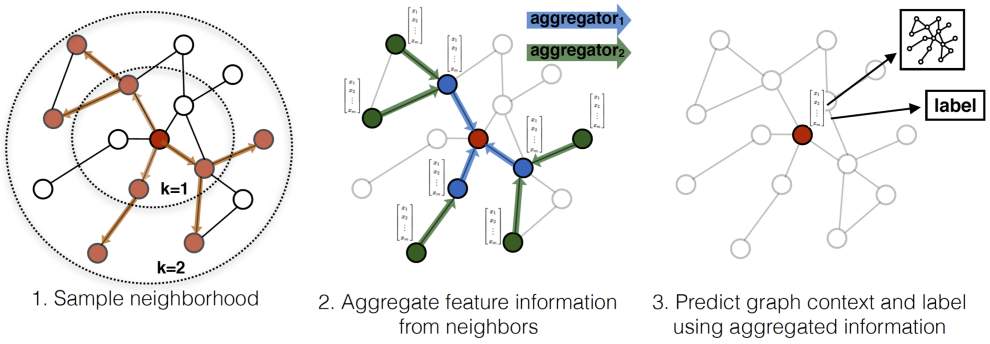


Figure 4.6.: 3 step GraphSAGE (sample and aggregate) model inference process. 1) Choose a node and sample other nodes from its neighborhood. 2) Aggregate the neighboring node embeddings into a target node embedding. 3) Predict the node label based on the aggregated embedding. (Image taken from http://snap.stanford.edu/graphsage/

4.4.2. Training

For the training process, we again on `stellargraph` python package. All the steps in the model forward pass shown in Figure 4.6 and the additional softmax layer to get the sip labels are `differentiable`. Thus, we can use backpropagation (Schmidhuber 2015) with Stochastic Gradient Descent (SGD) to train our model.

The following is a list of training hyper-parameters:

- `Batch Size = 50` - Small batch size helps with the memory usage and sometimes improves the model performance (Masters & Luschi 2018).
- `Number of Samples = [10, 5]` - Number of neighbors to sample for each GraphSAGE layer.
- `Wight Balancing` - Infrequent target labels contribute proportionally higher loss value to make final classification accuracy balanced amongst the target classes.
- `Optimizer = Adam` - The variation of (SGD) is commonly used because it is empirically known to have good results without much hyper-parameter tuning (Kingma & Ba 2017) .
- `Learning Rate = 0.005` - We found that changing the learning rate within the range of $0.005 - 0.00001$ does not impact the performance, so we choose the higher value for faster convergence.
- `Number of Epochs = 7` - We keep the number of epochs constant between different model instances. Usually, the model converges after just 3 or 4 epochs.
- `Loss Function = Categorical Cross Entropy` - Standard approach for neural classification tasks (Section 2.5.5).

When the training stage completes, it produces `<feat1_feat2_...>.results.json` file in the corresponding k-fold split directory. The results include training logs, data statistics, and, most importantly, test results. The schema for the results JSON file is given below in listing 4.4.2.

```
{
    "data_source": string,
    "data_dir": string,
    "results": {
        "train_size": int,
        "test_size": int,
        "subject_groups_train": {
```

```
            "none": int,
            "sc_guard": int,
            "cfi_verify": int,
            "oh_verify": int
        },
        "subject_groups_test": {
            "none": int,
            "cfi_verify": int,
            "oh_verify": int,
            "sc_guard": int
        },
        "graph_info": string,
        "test_metrics": [
            {
                "name": string,
                "val:": double
            }
        ],
        "classifier": {
            "sc_guard": double,
            "cfi_verify": double,
            "oh_verify": double,
            "none": double
        },
        "history": {
            "epochs": 7
            "training_log": {
                "loss": [double],
                "acc": [double],
                "categorical_accuracy": [double],
                "val_loss": [double],
                "val_acc": [double],
                "val_categorical_accuracy": [double]
            },
            "training_params": { "epochs": int, "steps": double}
        }
    },
    "features": [string]
```

```
}
```

Listing 4.5: Json schema for the single model instance results

4.5. Data Visualization

The final stage of the pipeline is `evaluation`. It contains a module for processing the training result files (`generate_plots.py`) and an interactive notebook file (`generate_plots.ipynb`) for result examination and plot generation. The format for the results file is given in listing 4.4.2. We deliberately used an interactive notebook environment for the evaluation task because this stage required the most frequent modifications during the development process. That is likely to continue in any future work based on this pipeline. For plot renderings, we rely on the `matplotlib` library.

5. Evaluation

The following chapter presents all experiment results conducted in this work. We first introduce the dataset with some of the aggregated statistics. Then, we describe the experimental setup in detail. Finally, we provide a visual representation of the results and discuss their relevance to our initial research questions.

5.1. Dataset

For our experiments, we use the binaries comprised of 40 simple programs from work by (Salem & Banescu 2016b), 20 MiBench programs (Guthaus, Ringenberg, Ernst, *et al.* 2001), and four large open-source CLI programs. We borrow the source programs from work by Ahmadvand, Fischer & Banescu 2019. MiBench programs were originally published for benchmarking CPU architectures. A good benchmark has to have a diverse set of programs. Diversity is desirable for Software Integrity Protection. We do not make any assumptions about what underlying protected programs are doing. Ideally, our dataset should approximate the global distribution of all binary programs one might want to protect with SIP.

SIP scheme	Total Blocks	Target Blocks	Avg # IR Lines	Avg % Added IR Lines
NONE	8,167		1075.40	0.00
SC	15,997	3,010	2090.50	94.39
CFI	37,373	3,130	4351.32	304.62
OH	32,476	5,589	6042.29	461.87

Table 5.1.: Protection Impact on Dataset
The table describes the impact of applying SIP on the size of the unobfuscated subject program. The first column contains a protection scheme. **Total Blocks** shows total basic blocks in unprotected programs in the dataset. **Target Blocks** lists the number of blocks that are target to the SIP localization task. The last two columns show the average LLVM IR lines per program and the average % increase SIP introduces, respectively.

Our dataset only contains intermediary program representations (LLVM IR). We

assume the attacker only has access to binaries and not to the high-level source code. Intermediary representation is appropriate because programs are usually distributed only as binaries without the source code. The format we chose is LLVM Bitcode[1] mainly because this pseudo-assembly language is open-source and independent of specific platform architecture. To analyze the binary with any model, we first disassemble it into a sequence of LLVM IR instructions and extract features from there.

5.1.1. Integrity Protections in dataset

Each program in our dataset has had either no protection or one of the SIP schemes (SC, OH, CFI) applied. It is also possible to combine these SIP schemes, however (Ahmadvand, Fischer & Banescu 2019) found that this does not affect the performance of *SIP localizer* model. So, we also do not stack different protection techniques and focus on one of them at a time. We also explore the impact of the SIP technique on the program size. Table 5.1 shows the total number of basic blocks in the dataset per SIP scheme, together with how many of these blocks belong to protection. The same information is given on the level of IR instructions. Self-checksumming has the most negligible impact on program size. It only increases the number of IR instructions by about 100%. Conversely, both OH and CFI increase the instructions by about 350%.

	Total Blocks			
Protection / **Source Dataset**	NONE	SC	OH	CFI
mibench-cov	1,821,549	3,596,715	6,997,849	6,367,185
simple-cov	58,474	115,782	212,839	417,708
simple-cov2	243,570	479,596	887,538	1,739,939

Table 5.2.: Source program protection scheme stats
Number of protected blocks in the final obfuscated dataset, grouped by protection schemes and source programs.

5.1.2. Obfuscations in dataset

In addition to SIP, various obfuscation techniques have also been applied to our dataset. In his master's thesis, Wessel 2019 found that without such techniques, detecting SIP in a binary is achievable even with basic pattern-matching, concluding that obfuscation can be an essential part of SIP. The obfuscation transformations in our dataset are:

[1] https://llvm.org/docs/BitCodeFormat.html/

- Instruction Substitution (IS) - (Section 2.4.1)
- Bogus Control Flow (BC) - (Section 2.4.2)
- Control Flow Flattening (CFF) - (Section 2.4.3)

In contrast to protection schemes, the combination of different obfuscation transformations is present in the dataset. Moreover, in some cases, the same obfuscation is applied multiple times. The combinations are denoted with the + sign, e.g. "BC+IS", "IS+BC+CFF". When the same obfuscation is used multiple times, we write "BC30" or "IS100". It is important to note that obfuscation transformations are not associative. "BC+IS" is not the same transformation as "IS+BC." Table 5.3 lists the obfuscations applied to simple-cov source programs, and then it's impact on program size. Mibench-cov and simple2-cov datasets contain more complex, identical set of obfuscations, shown in Table 5.4. Full obfuscation statistics table A.1 can be seen in the appendix A.

Obfuscation	Blocks	Avg IR Lines / Program	Avg % IR Lines Incr.
NONE	783	228.85	0.00
SUB	783	240.72	5.19
BCF30	2,019	458.98	100.56
FLA	2,415	533.38	133.07
SUB-BCF30	2,169	541.12	136.45
FLA-SUB	2,415	544.55	137.95
SUB-FLA	2,415	553.35	141.80
BCF30-SUB	2,073	641.35	180.25
BCF30-FLA	3,938	1035.38	352.43
SUB-BCF30-FLA	3,865	1055.12	361.06
BCF30-FLA-SUB	3,887	1155.60	404.96
FLA-BCF30	6,711	1281.95	460.17
SUB-FLA-BCF30	6,939	1358.47	493.61
FLA-SUB-BCF30	6,999	1382.67	504.18
BCF30-SUB-FLA	4,052	1395.05	509.59
FLA-BCF30-SUB	7,011	1793.22	683.58

Table 5.3.: Obfuscations in unprotected simple-cov programs
The first column gives the obfuscation technique. NONE means that no obfuscation has been applied. Suffix numbers denote the obfuscation strength.

Obfuscation	Blocks	Avg IR Lines / Program	Avg % IR Lines Incr.
NONE	6,601	4019.91	0.00
SUB	6,601	4924.09	22.49
FLA	17,764	8087.70	101.19
BCF30	17,365	8399.70	108.95
FLA-SUB	17,764	9123.70	126.96
SUB-FLA	17,764	9173.30	128.20
BCF40	20,605	9889.65	146.02
SUB-BCF30	17,335	11929.70	196.77
BCF30-SUB	17,239	12121.57	201.54
SUB-BCF40	19,855	13800.48	243.30
BCF40-SUB	20,797	15603.35	288.15
BCF30-FLA2	30,004	15906.39	295.69
BCF30-FLA	30,553	16653.52	314.28
BCF40-FLA	33,480	18276.61	354.65
SUB-BCF30-FLA	29,807	18599.57	362.69
BCF100	40,669	18690.57	364.95
FLA-BCF30	51,292	19673.91	389.41
BCF30-FLA-SUB	29,897	21592.65	437.14
BCF30-SUB-FLA	30,308	22068.43	448.98
FLA-SUB-BCF30	50,998	22100.91	449.79
SUB-BCF40-FLA	33,678	23208.17	477.33
FLA-BCF40	61,264	23360.26	481.11
SUB-FLA-BCF30	51,250	23638.52	488.04
BCF40-FLA-SUB	34,253	24554.52	510.82
FLA-SUB-BCF40	60,460	26541.83	560.26
SUB-BCF100	40,669	26826.96	567.35
BCF40-SUB-FLA	33,631	27028.00	572.35
SUB-FLA-BCF40	60,676	27687.65	588.76
FLA-BCF30-SUB	51,460	27710.96	589.34
BCF100-SUB	40,669	29865.91	642.95
FLA-BCF40-SUB	60,550	33334.13	729.23
BCF100-FLA	57,357	33992.91	745.61
SUB-BCF100-FLA	57,357	43395.43	979.51
BCF30-FLA2-SUB2	29,888	45101.43	1021.95
BCF100-FLA-SUB	57,357	45174.00	1023.76

Continued on next page

Obfuscation	Blocks	Avg IR Lines / Program	Avg % IR Lines Incr.
FLA-BCF100	124,222	46060.04	1045.80
BCF100-SUB-FLA	57,357	51048.87	1169.90
FLA-SUB-BCF100	124,222	54225.57	1248.92
SUB-FLA-BCF100	124,222	54976.74	1267.61
BCF30-SUB2-FLA2	30,087	55254.26	1274.51
FLA-BCF100-SUB	124,222	69479.30	1628.38

Table 5.4.: A list of all the obfuscations and their program size impact applied to mibench source programs.

5.1.3. Source Programs

The final training dataset is generated from two original program sources.

- **Simple**: A total of 40 simple programs, originally written in C. They have only a basic functionality such as sorting, I/O, factorial. These programs were taken from the work of Salem & Banescu 2016b.
- **MiBench**: contains 20 programs from the MiBench Benchmark suite (Guthaus, Ringenberg, Ernst, *et al.* 2001). These programs have more complex functionality, originally meant as a benchmark. We also include 4 large open source programs together with MiBench. (say.x.bc susan.bc tetris.bc toast.x.bc from (Ahmadvand, Fischer & Banescu 2019))

The table 5.5 captures source program statistics. To make our iterative development quicker, we generate two versions of obfuscated and protected a simple dataset. The first has a lighter set of obfuscations applied (simple-cov), keeping the dataset small and easy to run tests. The second (simple-cov2) has heavier obfuscations, identical to that of the mibench-cov. The -cov suffix in the dataset names indicates that coverage improver pass was run on the plain BC files. Cov suffix has no relevance with the current work as it is just a naming convention from the SIP implementation by (Ahmadvand, Fischer & Banescu 2019).

5.2. Experiment Setup

All the experiments mentioned in this chapter are done within our software pipeline. In this section, we list our experimental setup, including software/hardware dependencies. All the experiments and evaluation results can be easily repeated without any changes

src_dataset	No. Obfuscations	No. Programs	Total No. Basic Blocks
simple-cov	16	40	804,803
simple-cov2	41	40	3,350,643
mibench-cov	41	24	18,783,298

Table 5.5.: Dataset Statistics by source programs
The volume of data by source dataset.

to the code. We used a 10-core 2.4 GHz machine with 44GB RAM, running Ubuntu 20.04.2 LTS (Focal Fossa). The table 5.6 lists all the software dependencies, together with the exact version number and installation notes. Our machine does not have specialized hardware to accelerate deep learning training, such as a GPU or a TPU. We resort to using CPU together with manually compiled[2] Tensorflow (Martin Abadi, Ashish Agarwal, Paul Barham, *et al.* 2015) package to take advantage of the Intel SIMD instruction set. While this is still far from a performance even a moderate GPU could achieve, we still got about %30 to %40 improvement.

Program	Version	Notes
python	3.8.5	
tensorflow	2.5.0	built for SIMD CPU instructions
conda package manager	4.9.2	
jupyter notebook	1.0.0	
matplotlib	3.3.4	
networkx	2.5.1	for basic block graph analysis
stellargraph[3]	1.2.1	machine learning on graphs
numpy	1.2.0	
tqdm	4.61.2	low impact progress display
go	1.16.6	>= 1.16 is critically important
llir-grammar*	10	https://github.com/Megatvini/ll
c++	14	needed for LLVM analytic pass
LLVM	10.0.2	we use llvm-dis-10 and opt-10
docker	20.10.7	

Table 5.6.: Software dependencies and their versions in our SIP pipeline.

[2]https://www.tensorflow.org/install/source

5.3. Evaluation Metrics

We evaluate the model performance in *SIP localization* using the mean F1 score, calculated on the test portion of different k-fold splits (ref to background). Using this metric allows us to get balanced classification *precision* and *recall* despite having large imbalance of classes in the dataset (table 5.2).

Additionally, we also display the variability of the F1 score through different splits. We use this additional information to reason about how reliable the models and our training procedure are.

5.4. Experiments

In this section, we describe in detail all the experiments conducted to answer specific research questions. We also show a representation of results that is appropriate for interpretation. We also discovered some findings closely related to the research subject.

5.4.1. Research Question 1

How well do SIP schemes hold up against machine learning attacks when combined with various obfuscations?

We take the dataset and train an independent machine learning model for each obfuscation. A single obfuscation directory contains examples of all the different SIP schemes (NONE, SC, OH, CFI). The model is trained on SIP localization, so the output predicts one out of these classes per basic block in the dataset. The results are visually displayed in Figure 5.1 for simple-cov dataset and figure 5.2 for mibench-cov. There are a number of conclusions regarding RQ1 that can be drawn from the aforementioned figures:

- SIP schemes are easily defeated using machine learning without obfuscations. Our model has a near-perfect F1 score with low variance on all the source programs and SIP schemes. Without obfuscations, OH is the only option that has some (though minimal) effectiveness.
- The model recognizes unprotected basic blocks much better than protected ones. In the mibench dataset, the F1 score of the unprotected basic blocks is well above 0.9 even in the presence of the strongest obfuscations. Simple-cov dataset models are similar in that they also perform better on unprotected blocks. However it drops down to 0.8 on **BC30+IS+FLA** obfuscation.

- Control flow integrity verification seems to be slightly more effective on smaller dataset with simple obfuscations (Figure 5.1). However, on a larger mibench dataset, oblivious hashing performs better (Figure 5.2). In combination with strong obfuscation, such as **BC30+IS2+FLA2**, the localization F1 score ranges in-between about 0.35 and 0.7.

	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std
obfuscation								
NONE	0.991	0.004	0.998	0.004	0.976	0.014	0.902	0.034
IS	0.987	0.005	0.997	0.005	0.970	0.038	0.859	0.054
IS+BC30	0.978	0.009	0.866	0.064	0.745	0.105	0.704	0.111
BC30	0.974	0.009	0.845	0.116	0.649	0.075	0.700	0.100
BC30+IS	0.972	0.008	0.775	0.092	0.572	0.116	0.736	0.071
FLA	0.912	0.028	0.442	0.102	0.246	0.079	0.459	0.060
FLA+BC30	0.949	0.018	0.419	0.141	0.176	0.058	0.461	0.043
IS+FLA	0.909	0.012	0.386	0.050	0.278	0.032	0.358	0.049
FLA+IS	0.908	0.020	0.370	0.172	0.285	0.081	0.409	0.056
FLA+IS+BC30	0.947	0.005	0.365	0.159	0.162	0.027	0.455	0.066
FLA+BC30+IS	0.929	0.020	0.310	0.126	0.129	0.032	0.425	0.035
IS+FLA+BC30	0.939	0.012	0.226	0.080	0.187	0.041	0.383	0.032
BC30+FLA+IS	0.821	0.043	0.217	0.069	0.093	0.026	0.313	0.064
BC30+FLA	0.827	0.029	0.121	0.027	0.117	0.022	0.366	0.077
IS+BC30+FLA	0.826	0.016	0.175	0.066	0.102	0.021	0.296	0.035
BC30+IS+FLA	0.812	0.062	0.146	0.044	0.129	0.066	0.290	0.089

Table 5.7.: SIP localization F1 score split by protection techniques and obfuscations. Rows are sorted by the average mean score of all protections.

	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std
obfuscation								
IS	0.996	0.003	0.997	0.005	0.983	0.008	0.958	0.033
NONE	0.995	0.004	0.999	0.001	0.971	0.046	0.943	0.067
BC30	0.996	0.001	0.999	0.002	0.984	0.012	0.885	0.035

Continued on next page

	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std
obfuscation								
BC100	0.997	0.001	0.998	0.002	0.994	0.010	0.859	0.026
BC40	0.995	0.002	0.994	0.012	0.975	0.022	0.858	0.027
IS+BC100	0.996	0.002	0.999	0.001	0.986	0.010	0.829	0.043
BC40+IS	0.994	0.002	0.992	0.013	0.971	0.027	0.826	0.045
IS+BC40	0.994	0.002	0.998	0.002	0.988	0.008	0.816	0.027
BC100+IS	0.995	0.003	0.995	0.004	0.970	0.022	0.822	0.094
IS+BC30	0.993	0.002	0.999	0.001	0.983	0.010	0.810	0.027
BC30+IS	0.991	0.003	0.977	0.025	0.968	0.013	0.792	0.062
FLA	0.993	0.003	0.897	0.079	0.938	0.047	0.838	0.028
IS+FLA	0.989	0.004	0.838	0.127	0.936	0.031	0.779	0.111
BC100+FLA	0.992	0.006	0.843	0.048	0.856	0.125	0.730	0.149
FLA+IS+BC100	0.996	0.002	0.864	0.057	0.894	0.078	0.759	0.143
FLA+IS	0.987	0.005	0.883	0.107	0.907	0.068	0.714	0.076
FLA+BC100	0.996	0.003	0.846	0.093	0.866	0.094	0.765	0.144
BC30+FLA	0.988	0.009	0.853	0.099	0.770	0.089	0.745	0.132
FLA+BC40	0.994	0.003	0.852	0.095	0.765	0.110	0.780	0.107
BC100+IS+FLA	0.989	0.010	0.846	0.053	0.815	0.026	0.712	0.214
FLA+BC30	0.994	0.004	0.841	0.072	0.779	0.070	0.780	0.126
FLA+IS+BC30	0.993	0.004	0.878	0.073	0.783	0.100	0.746	0.107
BC30+FLA+IS	0.989	0.005	0.879	0.054	0.746	0.127	0.729	0.056
FLA+BC100+IS	0.996	0.002	0.827	0.063	0.831	0.104	0.785	0.098
FLA+IS+BC40	0.994	0.002	0.847	0.047	0.788	0.124	0.708	0.113
BC40+FLA	0.990	0.004	0.852	0.084	0.733	0.077	0.762	0.062
IS+BC100+FLA	0.988	0.005	0.900	0.066	0.811	0.122	0.620	0.095
IS+FLA+BC40	0.992	0.001	0.907	0.071	0.773	0.095	0.629	0.091
BC40+IS+FLA	0.985	0.011	0.847	0.086	0.711	0.072	0.694	0.192
IS+FLA+BC100	0.995	0.002	0.852	0.088	0.790	0.150	0.713	0.082
IS+FLA+BC30	0.991	0.004	0.873	0.069	0.695	0.166	0.712	0.110
BC30+FLA2	0.985	0.005	0.850	0.131	0.759	0.076	0.672	0.083
BC40+FLA+IS	0.988	0.003	0.879	0.068	0.734	0.055	0.691	0.083
FLA+BC30+IS	0.991	0.005	0.836	0.084	0.667	0.098	0.737	0.152
BC30+IS+FLA	0.987	0.004	0.863	0.073	0.711	0.074	0.715	0.097
IS+BC40+FLA	0.982	0.006	0.831	0.096	0.685	0.066	0.610	0.102

Continued on next page

	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std
obfuscation								
FLA+BC40+IS	0.991	0.005	0.842	0.073	0.583	0.120	0.757	0.156
BC100+FLA+IS	0.987	0.006	0.795	0.032	0.765	0.095	0.618	0.107
IS+BC30+FLA	0.982	0.006	0.780	0.103	0.704	0.099	0.634	0.066
BC30+IS2+FLA2	0.975	0.012	0.759	0.066	0.618	0.058	0.575	0.132
BC30+FLA2+IS2	0.977	0.006	0.800	0.105	0.561	0.057	0.568	0.078

Table 5.8.: SIP localization F1 score on mibench-cov split by protection techniques and obfuscations. Rows are sorted by the average mean score of all protections.

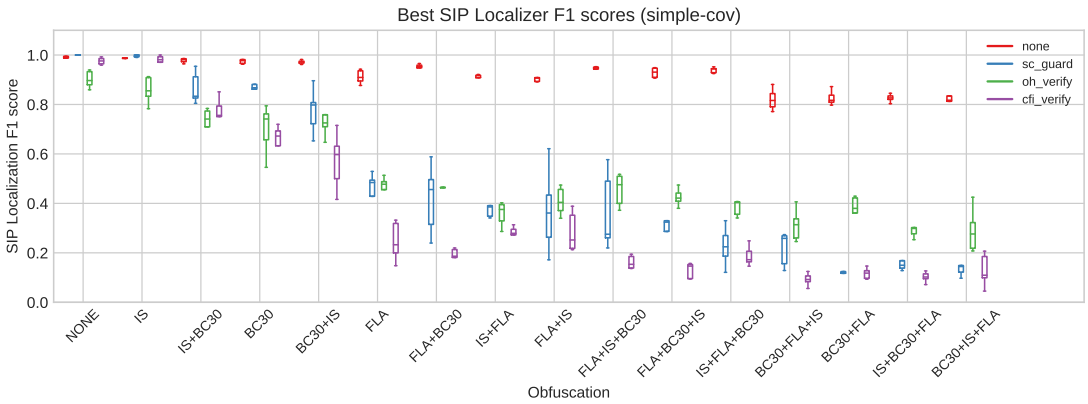


Figure 5.1.: SIP localization F1 scores of our best model in the presence of various obfuscations. Models were trained on simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for corresponding obfuscations.

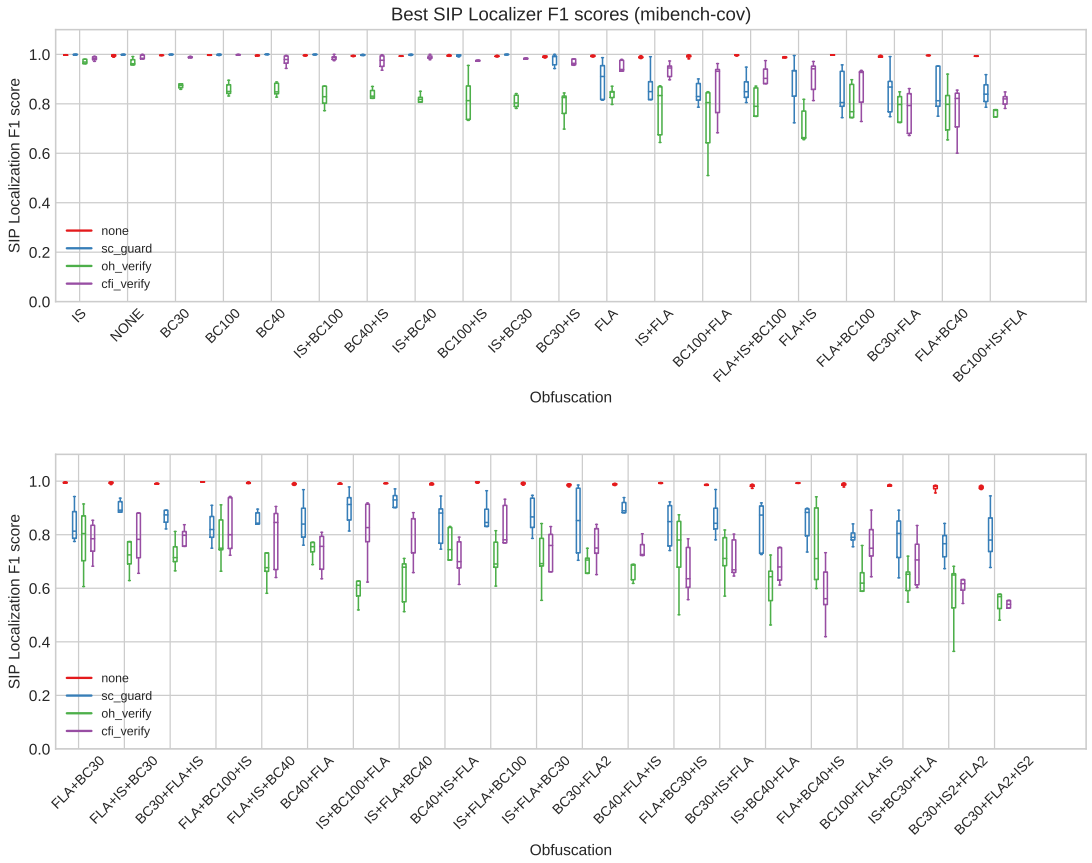


Figure 5.2.: SIP classification F1 scores of our best model in the presence of various obfuscations. Models were trained on mibench-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for corresponding obfuscations.

5.4.2. Research Question 2

How do more expensive neural program embeddings - Code2Vec (Alon, Zilberstein, Levy & Yahav 2018) or IR2Vec (VenkataKeerthy, Aggarwal, Jain, *et al.* 2020) compare against traditional TF-IDF representation? To answer this question, we break down the SIP localization score by individual features and compare them for all source datasets, protection & obfuscations.

The figures 5.3, 5.4, 5.5 and 5.6 show SIP localization results on simple-cov dataset and SIP schemes (NONE, SC, OH, CFI) respectively. Similarly, the figures 5.7, 5.8, 5.9 and 5.10 display same results for the mibench-cov dataset. We see significantly different results within the different features and even with the same features and obfuscations but different source programs.

Similar to RQ1, unprotected basic blocks are identified better than protected ones, regardless of which features are used.

Overall, we draw the following conclusions from the results regarding each of the features:

- **PDG** - Both on simple-cov and mibench-cov datasets, PDG generally performs worse than IR2Vec and Code2Vec features. It is better than TF-IDF except for correctly classifying unprotected basic blocks. The disadvantage of only using this feature representation is especially apparent on mibench-cov dataset, where features obtained from neural models perform much better.

- **TF-IDF** - Performs well on unprotected blocks, even on the smaller simple-cov dataset. However, the performance drops drastically whenever any protection technique is applied. In other words, when the model looks at the basic blocks through the TF-IDF representation, it correctly identifies unprotected blocks but fails to distinguish between different protection techniques. Thus, TF-IDF features could be advantageous if we just used binary (protected vs. unprotected) sip localization.

- **IR2Vec** - On a smaller dataset (simple-cov), IR2Vec mostly shows better results than all the other features regardless of obfuscations. Despite being comparatively better, the overall F1 score is still below 0.5 for most protections and obfuscations. This can be explained by our approach of using pre-trained seed embeddings vocabulary (section 4.3.3), whereas the Code2Vec feature extractor model gets trained from random weights. The former already gets us reasonable basic block representation (for thread coarsening task VenkataKeerthy, Aggarwal, Jain, *et al.* 2020) without any training data. On a larger dataset (mibench-cov) it is still better than non-deep learning features (PDG, TF-IDF) but gets outperformed by Code2Vec.

- **Code2Vec** - Contrary to Ir2Vec features, Code2Vec shows the best results on a larger dataset and struggles on a smaller one. This is typical for deep learning models, they easily overfit (Dietterich 1995) on the smaller dataset, but the performance gets better as the training data volume grows.

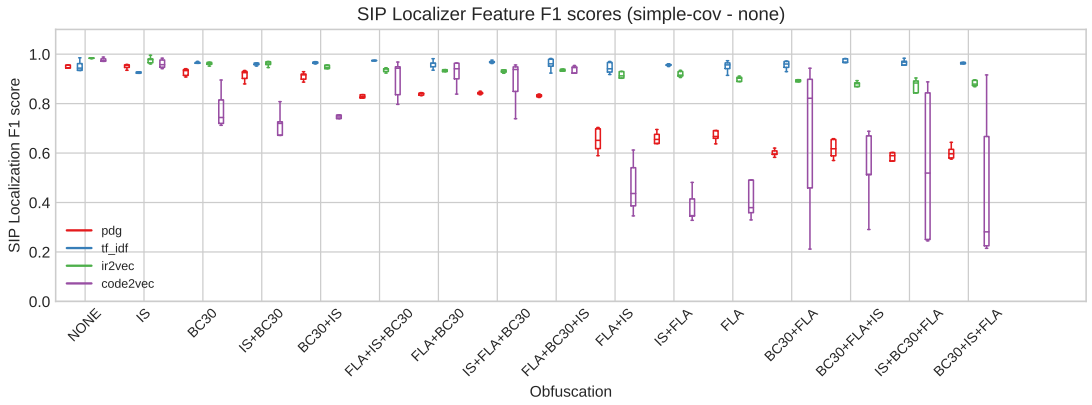


Figure 5.3.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

We also consider using the combinations of features to see if the overall performance can be improved over choosing one. The results on different datasets are given in appendix section A.2. From these diagrams, we observe that the features are complementary. The best overall results are obtained by combining all the described features, regardless of the training dataset.

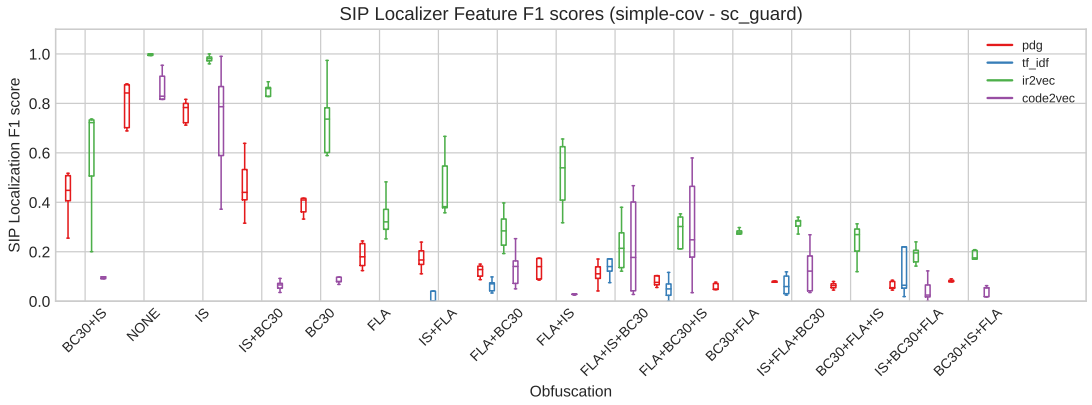


Figure 5.4.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

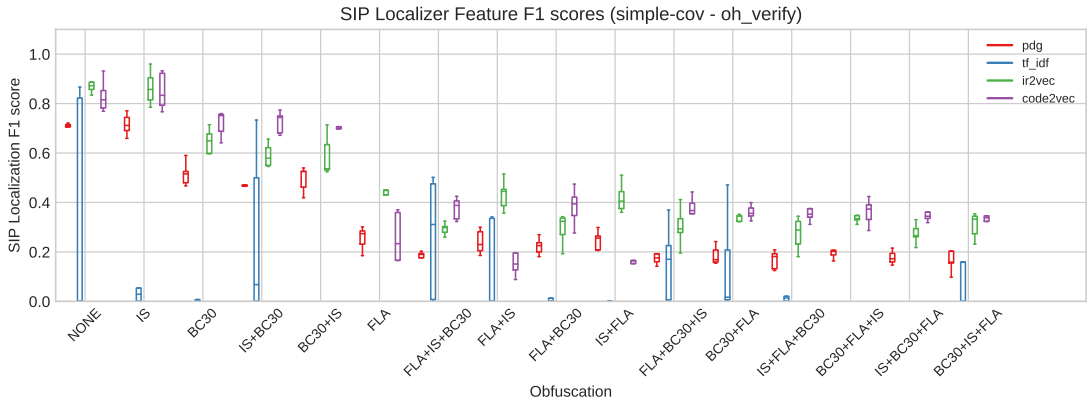


Figure 5.5.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

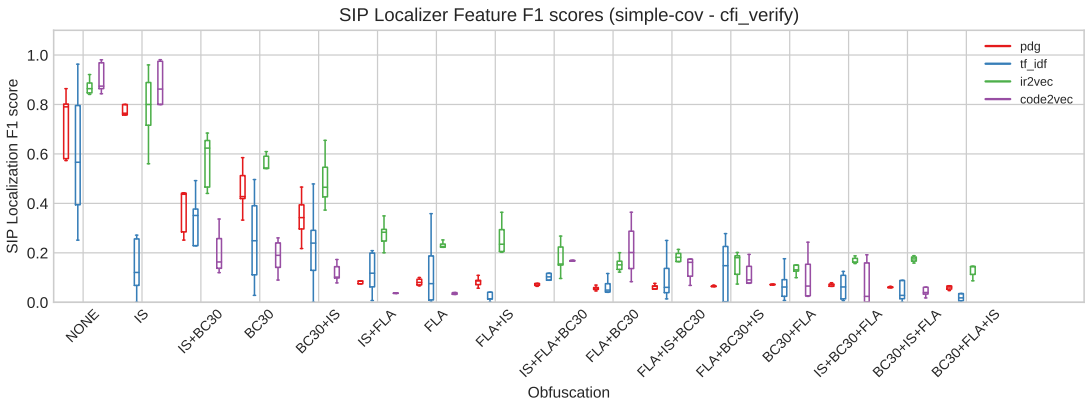


Figure 5.6.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

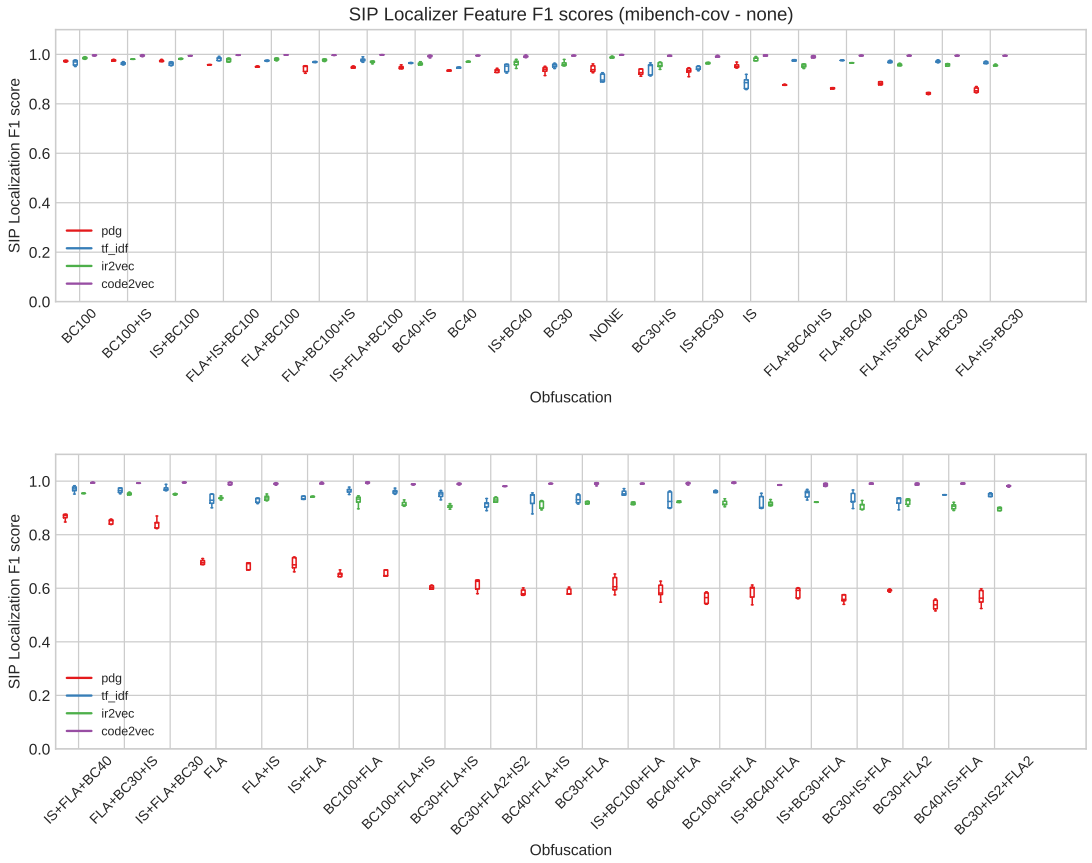


Figure 5.7.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

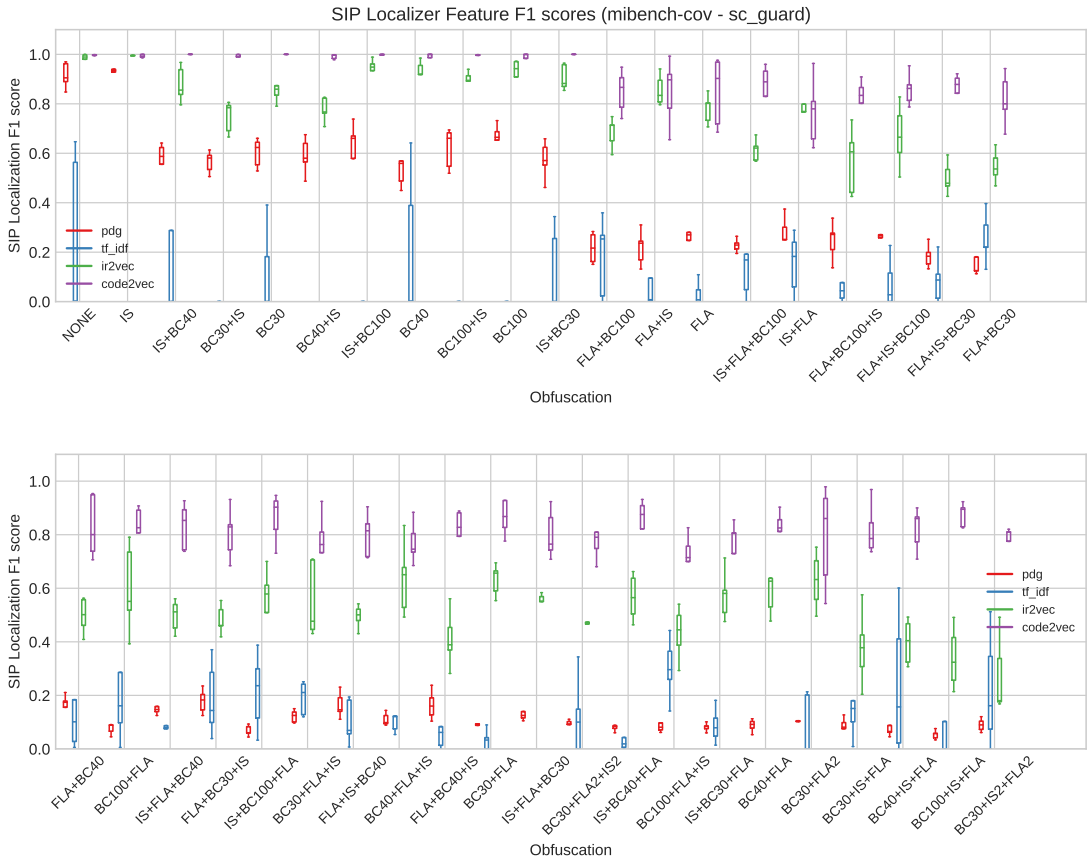


Figure 5.8.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

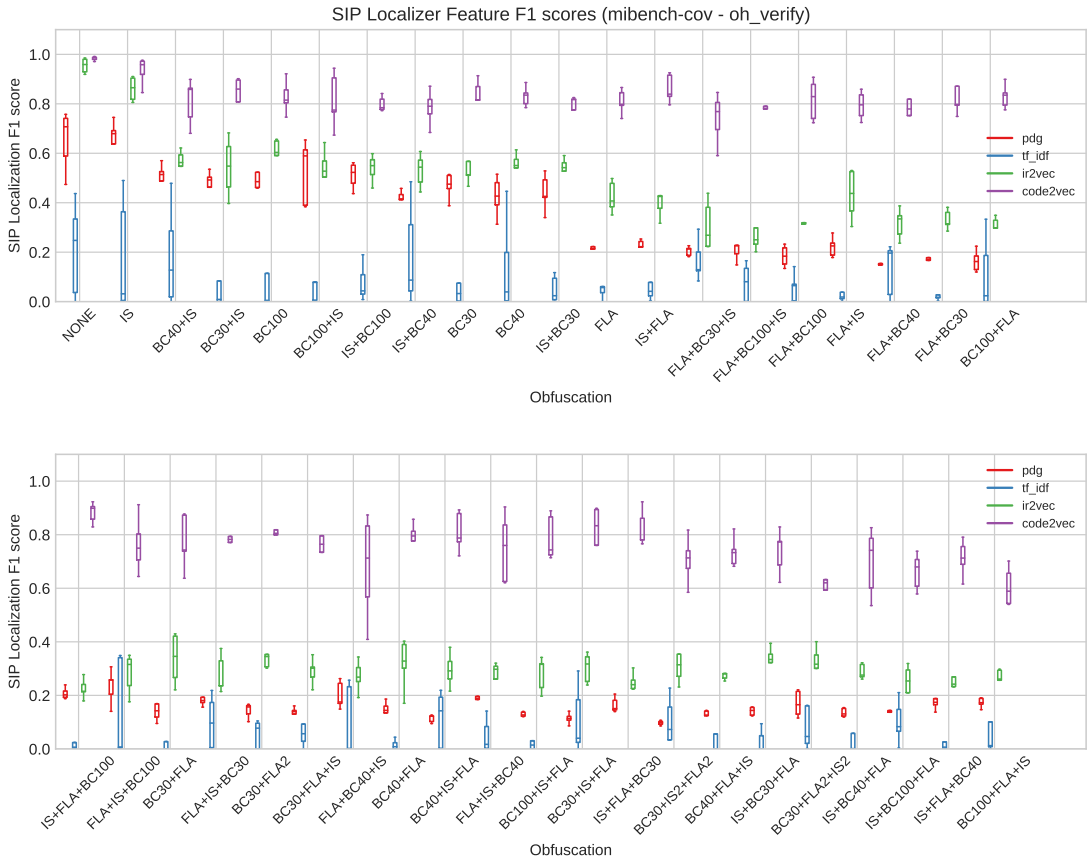


Figure 5.9.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

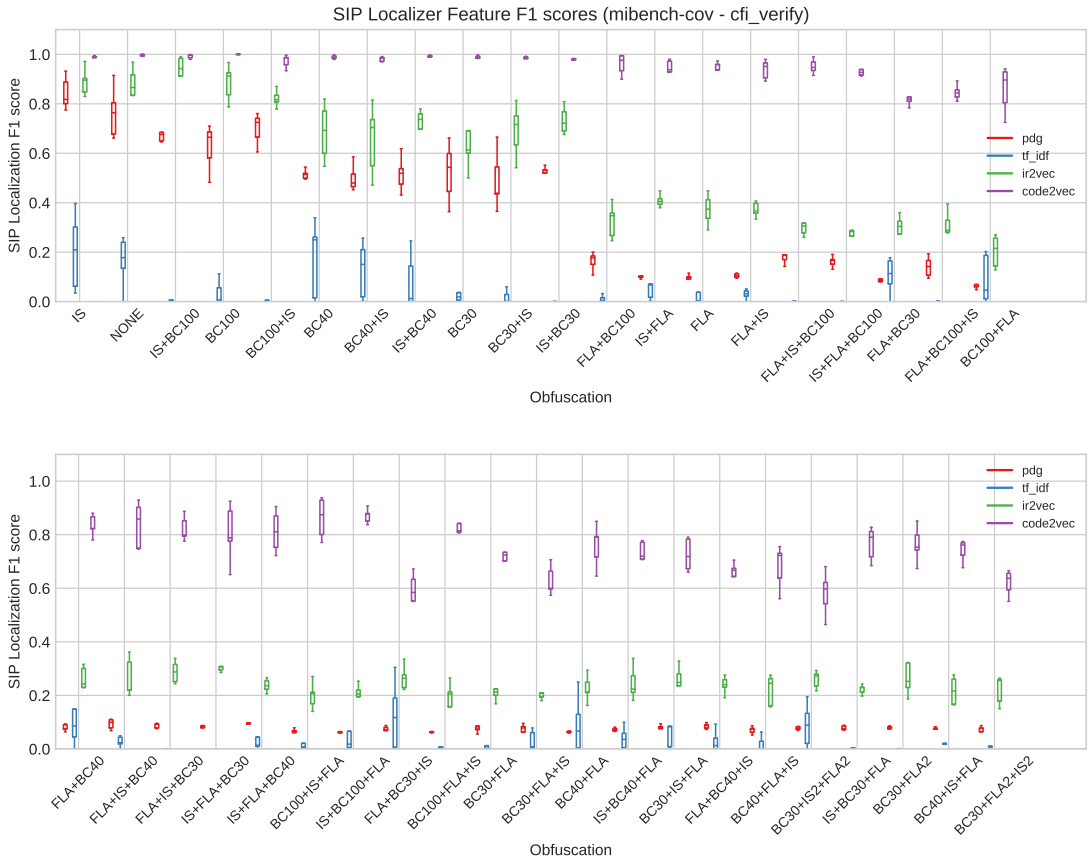


Figure 5.10.: SIP localization F1 scores of different features on a simple-cov dataset. Differently colored box plots show the median F1 scores (from k-fold splits) and interquartile ranges around it for different features and corresponding obfuscations.

5.4.3. Research Question 3

How does obfuscation of protected programs impact SIP localization performance?

The tables A.3 and A.4 show the strongest obfuscations and best protections in combinations to use with them (words F1 score of the best feature combinations) according to our experiment results. The table 5.9 shows the best obfuscations by protection scheme and the dataset.

data_source	sc_guard		oh_verify		cfi_verify	
	Best Obfs.	F1 score	Best Obfs.	F1 score	Best Obfs.	F1 score
mibench-cov	BC30+IS2+FLA2	0.759	BC30+FLA2+IS2	0.568	BC30+FLA2+IS2	0.561
simple-cov	BC100+FLA	0.121	BC100+IS+FLA	0.290	BC100+FLA+IS	0.093

Table 5.9.: Best SIP localization results (lowest mean F1 score) against the best model.

BC30+FLA+IS obfuscation with CFI seems to be the most effective on a simple dataset, with IS+BC30+FLA and CFI being the close second. The same can be seen in Figure 5.1. Interestingly, BC30+FLA performs quite well, but the same obfuscation combination, with the reversed order, i.e., FLA+BC30 is much less effective. For all the top obfuscation instances in the simple-cov dataset, CFI protection performs the best.

From the larger mibench dataset, we discovered that those obfuscation combinations where BC comes before FLA usually perform better than those with the reversed order.

Surprisingly, FLA+IS has a lower mean F1 score than FLA+IS+BC100 or FLA+IS+BC40. Adding BC obfuscation in combination with IS and FLA does not seem to provide additional resilience against ML attacks.

5.4.4. Research Question 4

Which obfuscation techniques are the most efficient relative to increase in program size?

From the previous research questions, we see that stacking and combining obfuscations generally improve resilience against ML attacks. However, they also increase the total number of instructions in a program. Obfuscation size increase motivates us to find the most cost-effective obfuscations for SIP. In practical terms, one might have some maximum program size multiplier and choose the most effective protection & obfuscation combination. We display the exact answer to this results according to our experiments in the figure 5.11. At any given maximum multiplier, the lowest line chart defines the most effective protection scheme and obfuscation. For example, at a max multiplier of 5 through 9.3, SC's most effective obfuscation is with IS+FLA obfuscation. In the 11.3 - 15.3 maximum multiplier range, OH with FLA+IS becomes the best choice.

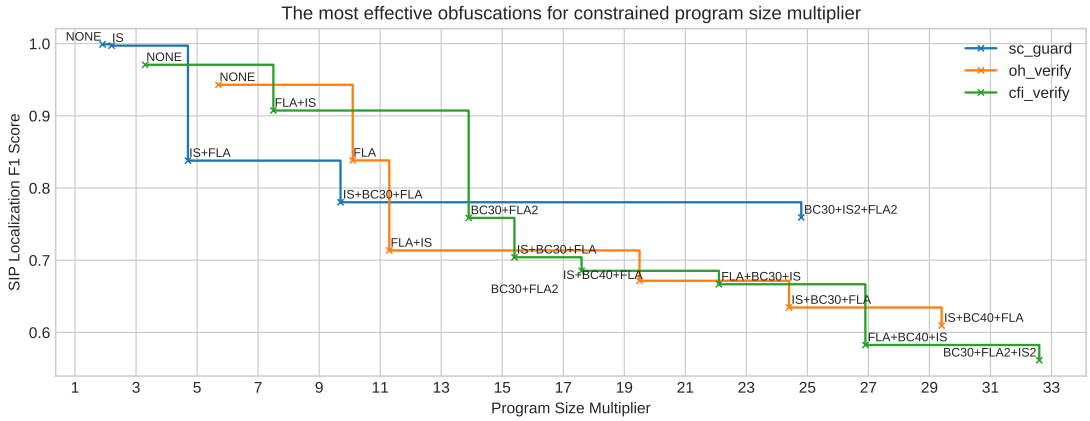


Figure 5.11.: Given the maximum program size multiplier, the figure shows a step function for the most effective protection & obfuscation. The figure uses the data from mibench-cov.

We also closely look at the efficiency of all the other obfuscations. In order to reconcile the obfuscation cost with SIP performance, we display the results on a scatter plot, where one axis shows the obfuscation impact on the program size and another shows SIP localization performance. The most effective obfuscation data points should be near the corner of low SIP localization score and low program size increase.

Simple-Cov Dataset

The Figures 5.12, 5.13, 5.14 and 5.15 show SIP localization scores and obfuscation impacts for (NONE, SC, OH, CFI) protections respectively. They all have very similar structure. The best strongest obfuscation against our model seems to be some combination of IS, BC30 and FLA, with about 430% IR lines increase. However, there is another, possibly preferable cluster of obfuscations (FLA, FLA+IS, IS+FLA), with much smaller cost ($100\% - 150\%$) and slightly worse performance.

Mibench-Cov Dataset

Mibench-cov results have a much more comprehensive range of trade-offs between localization score and IR lines increase.

- NONE (Figure 5.16) - As we have already seen, unprotected blocks are generally classified very accurately regardless of obfuscation used. Still, BC30+FLA2+IS2 has the lowest unprotected basic block F1 score of 0.98, but it is costly (1600%). Similarly performing but cheaper alternatives are IS+BC40+FLA (800%) and IS+BC30+FLA (400%).
- SC (Figure 5.17) - When using SC protection, the most robust obfuscation choice against our model is BC30+IS2+FLA2. It increases program size by 2000%, but our best model achieved an F1 score of only 0.75. However, the F1 score of 0.81 could be achieved by IS+BC30+FLA, with a much lower impact - 400%.
- OH (Figure 5.18) - OH protection is most robust when combined with BC30+FLA2+IS2 (1500% size increase, 0.55 F1 score), but IS+BC30+FLA (400% size increase, 0.65 F1 score) could be much more practical. FLA+IS is another interesting data point, which achieves a 0.67 F1 score but at a 200% size increase.
- CFI (Figure 5.19) - The figure is similar to OH, BC30+FLA2+IS2 being the most effective obfuscation (0.55 F1, 1500% size increase) and BC30+IS+FLA potentially better choice at 0.66 F1 score with 500% size increase. However, FLA, FLA+IS, and IS+FLA obfuscations no longer work well. They should not be considered as practical obfuscations with CFI.

Overall, BC30+IS+FLA and IS+BC30+FLA obfuscation combinations seem to offer a good trade-off sweet-spot between protection and program size increase in all protection schemes and source programs.

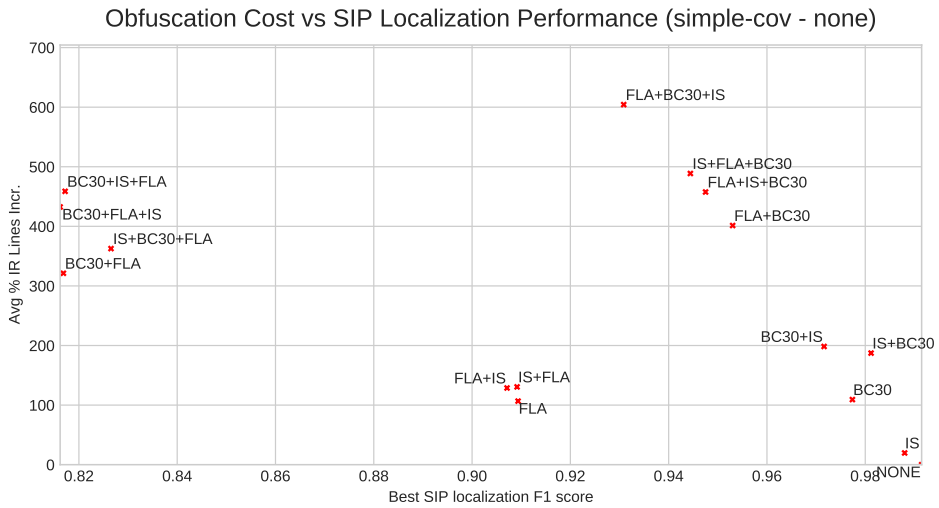


Figure 5.12.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and unprotected blocks.

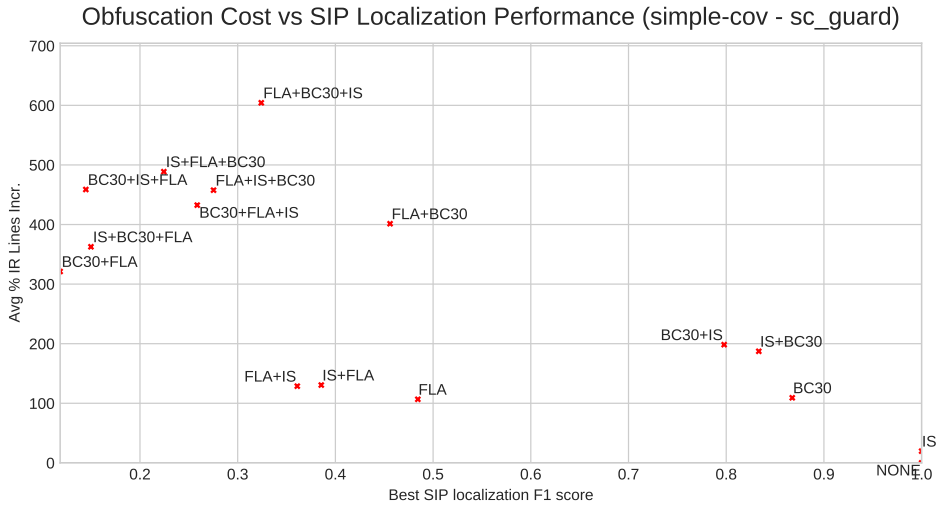


Figure 5.13.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and SC blocks.

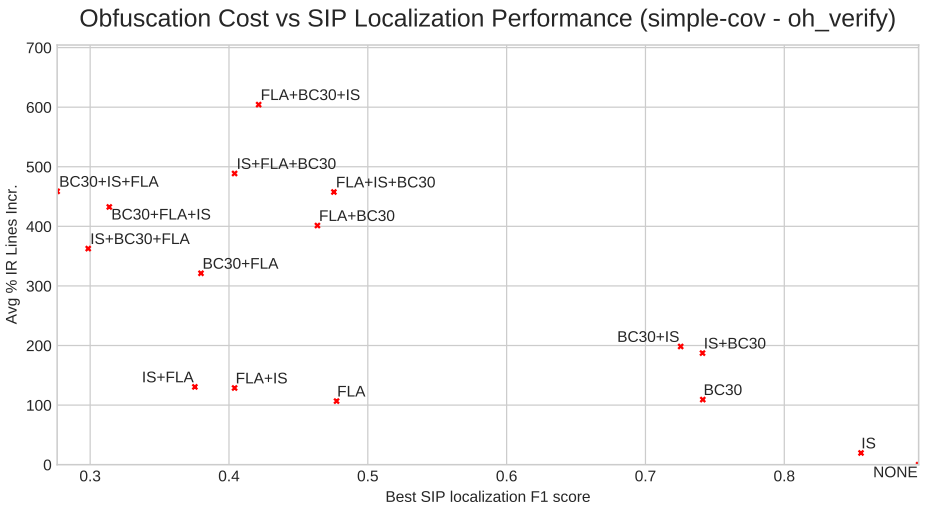


Figure 5.14.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and OH blocks.

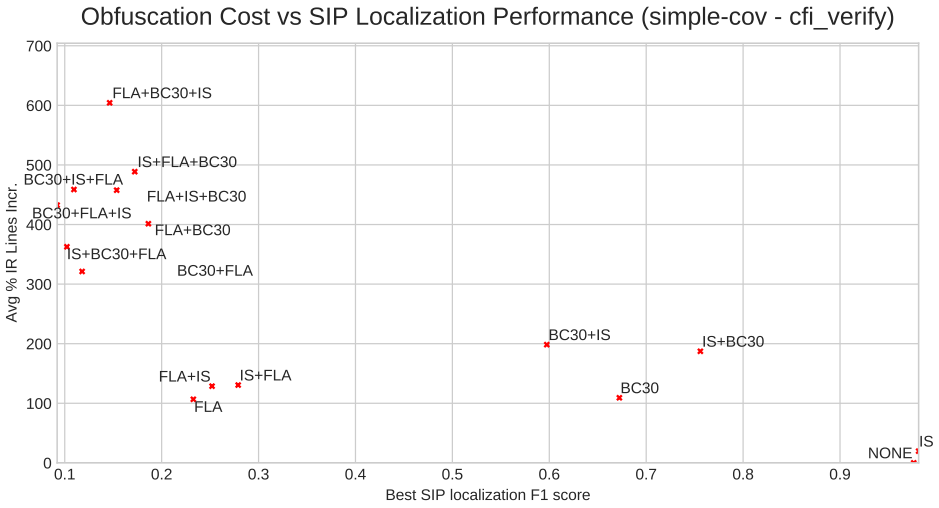


Figure 5.15.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and CFI blocks.

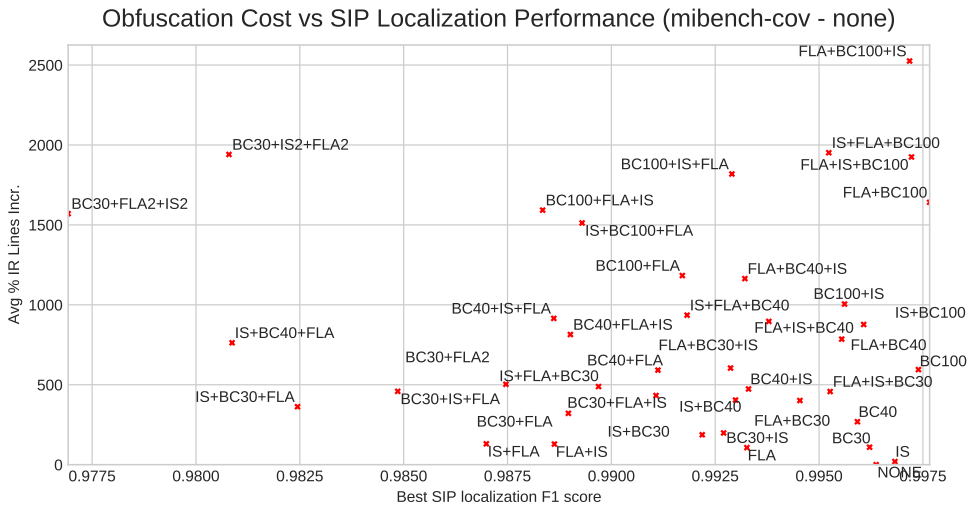


Figure 5.16.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and unprotected blocks.

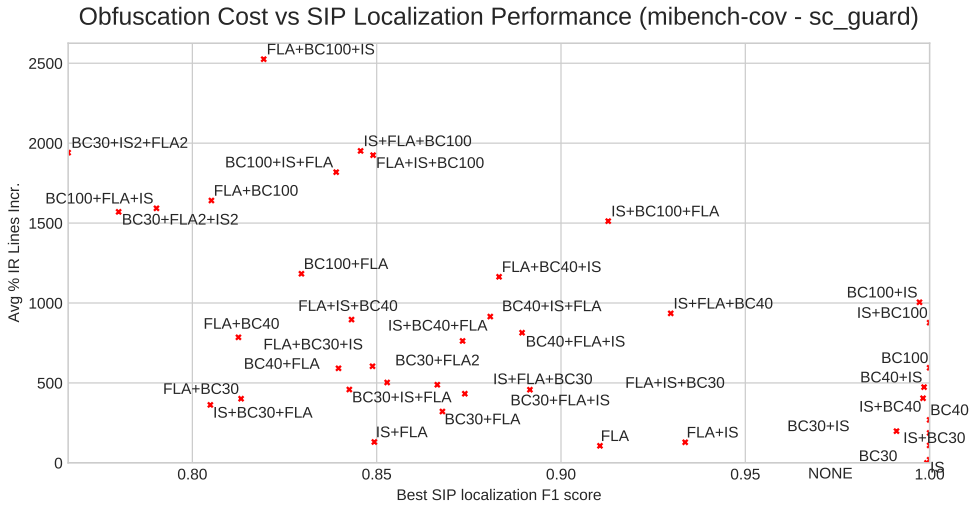


Figure 5.17.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and SC blocks.

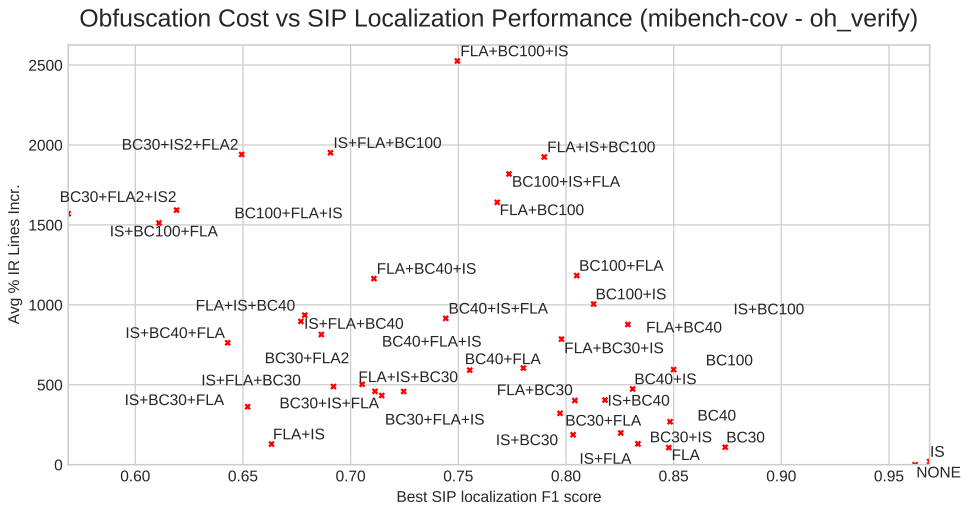


Figure 5.18.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and OH blocks.

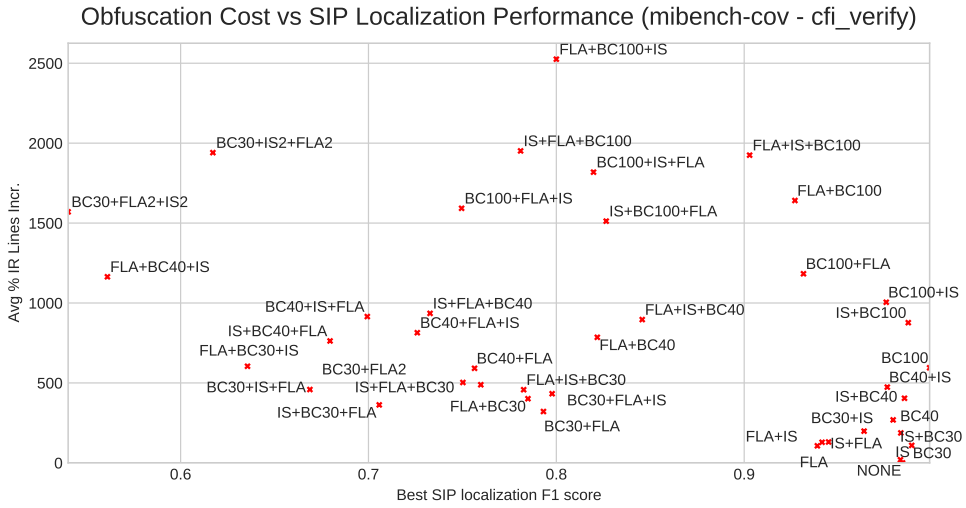


Figure 5.19.: The scatter plot showing obfuscation cost (Average % increase in LLVM IR lines) on the Y-axis and best model SIP localization F1 score on the X-axis. The results are from the simple-cov dataset and CFI blocks.

5.4.5. Research Question 5

How reliable are Machine Learning attacks against SIP with different obfuscations?

From the Figures (5.1, 5.2) showing SIP localization F1 scores on various SIP schemes obfuscations and feature sets, we can see that in some cases mean F1 score of the model doesn't tell the whole story. In some cases, the difference between the best and the worst k-fold F1 test scores can be 0.6 (Figure 5.6 simple-cov - cfi_verify, no obfuscation). Tendency for the high variance is especially apparent when using TF-IDF features.

In other cases, the k-fold results are always within a few percentage point (5.7 - unprotected blocks). Unprotected blocks are always classified reliably, but for all the other types of protections, we see a high standard deviation of the F1 score, typically about 0.1. A notable data point here is CFI protection with the BC30+FLA2+IS2 obfuscation, showing a mean F1 score of 0.56 and std. of just 0.057 on the mibench-cov dataset.

We postulate two possible explanations for these results:

- Some programs in the training dataset could be more similar to each other than others. This could cause the k-fold results to be highly dependent on the particular train/test split. If similar programs end up in different groups, the model would probably generalize better.
- The optimization convergence highly depends on initialization weights and hyper-parameters. We do not do any extensive hyper-parameter tuning and keep them fixed between different k-fold training runs. It could be the case that our optimization space is highly sensitive to these parameters, causing a high variance of results.

The first explanation suggests that we would see decreased variance if we kept the train/test split fixed and ran the ML pipeline multiple times. So, the explanation could be strengthened or discarded by a simple experiment of running the training procedure some number of times without changing the train/test split.

The second explanation requires including an additional layer of hyper-parameters tuning to the ML pipeline. There are several existing approaches for this, including grid search or Bayesian optimization (Turner, Eriksson, McCourt, *et al.* 2021). For instance, we found a few training/testing curves (Figure 5.20) on the mibench-cov dataset that suggests continuing the training process for a few more epochs would improve the performance.

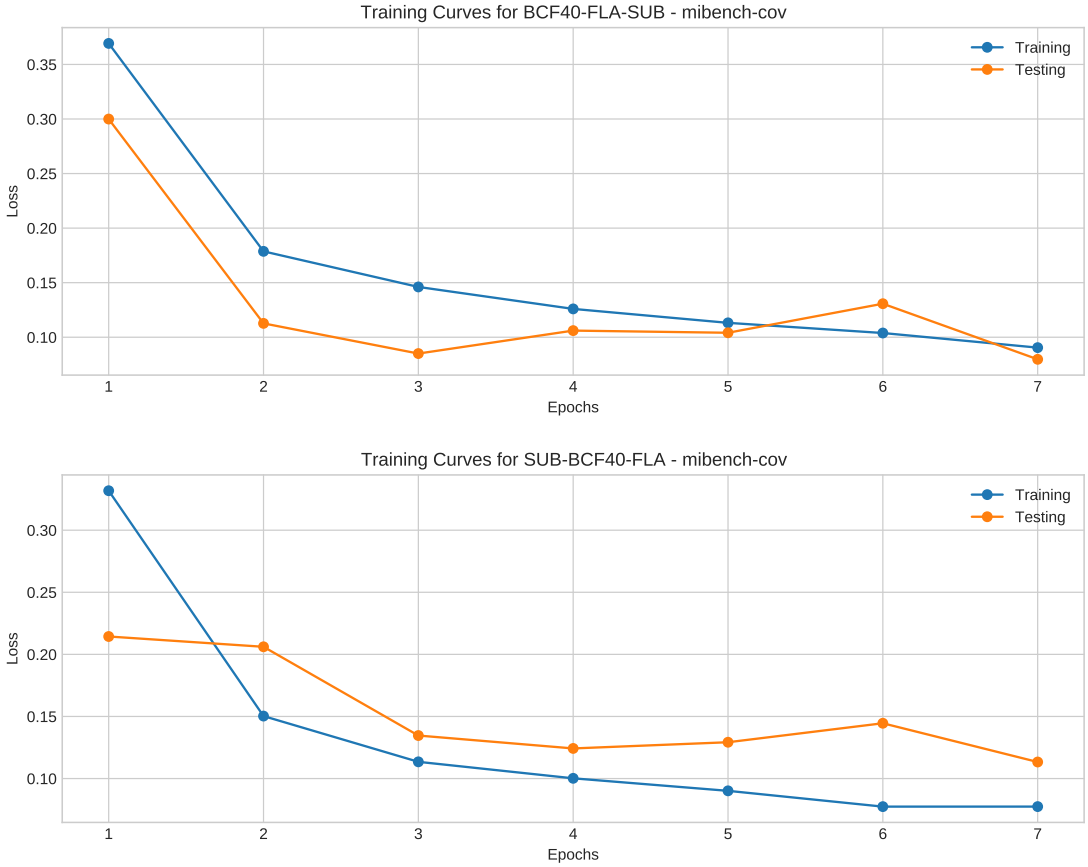


Figure 5.20.: Hand-picked loss curves that suggest an increasing number of epochs would be beneficial for test F1 scores. Training is run on the mibench-cov using the combination of all the features.

5.4.6. Research Question 6

How well does the SIP localizer model perform on a new program outside of its training data distribution?

To answer this question, we switched the model testing procedure from using the same source programs to training on mibench-cov and testing on simple-cov2 dataset. We already know from RQ1 that the deep model trained on mibench-cov using all 4 (PDG, TF-IDF, Ir2Vec, Code2Vec) features performs well on the unseen programs from the same mibench-cov. However, the attacker generally would not have the source programs from the same distribution in the training set. Thus, we evaluate the model F1 score on an entirely different distribution of programs (simple-cov2). Figure 5.21 shows the testing results on simple-cov2 dataset of the model that was trained on the identical obfuscation from mibench-cov dataset. Compared to Figure 5.8 showing same results for mibench-cov, we see strong decrease in performance.

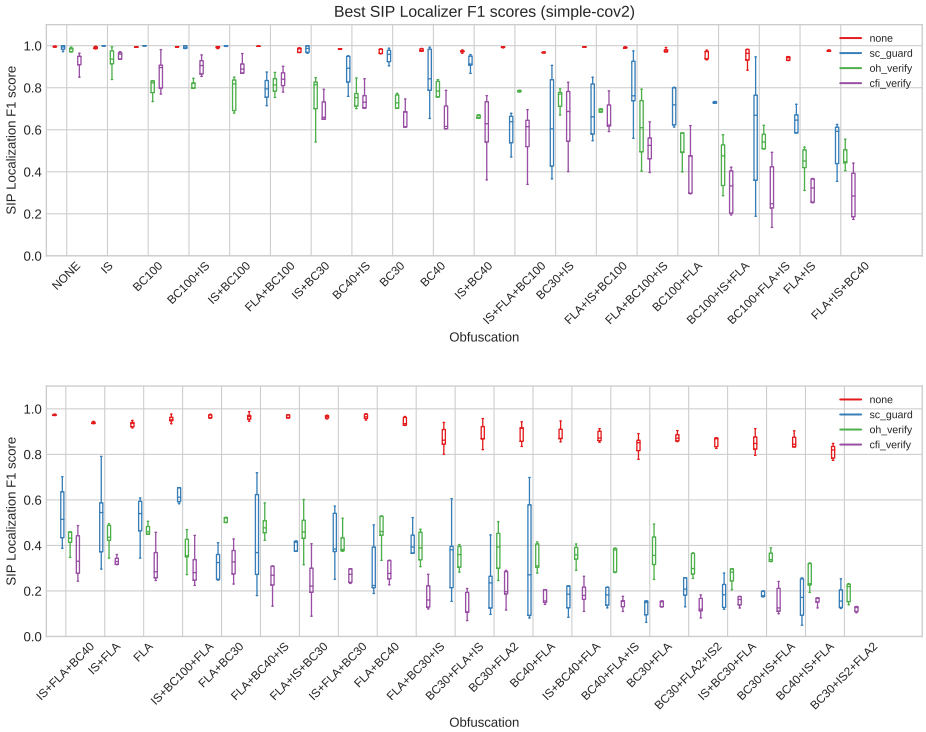


Figure 5.21.: The plot shows SIP localizer model F1 score, trained on mibench-cov dataset and tested on the same obfuscation in simple-cov2. Scores are split by protection schemes

We also include precision and recall metric graphs of this experiment on Figures (5.22, 5.23). The leading cause of the low F1 scores are low precision values. Our models frequently incorrectly attribute protection targets to plain basic blocks. On the other hand, basic blocks that are marked as protection blocks are mainly classified correctly.

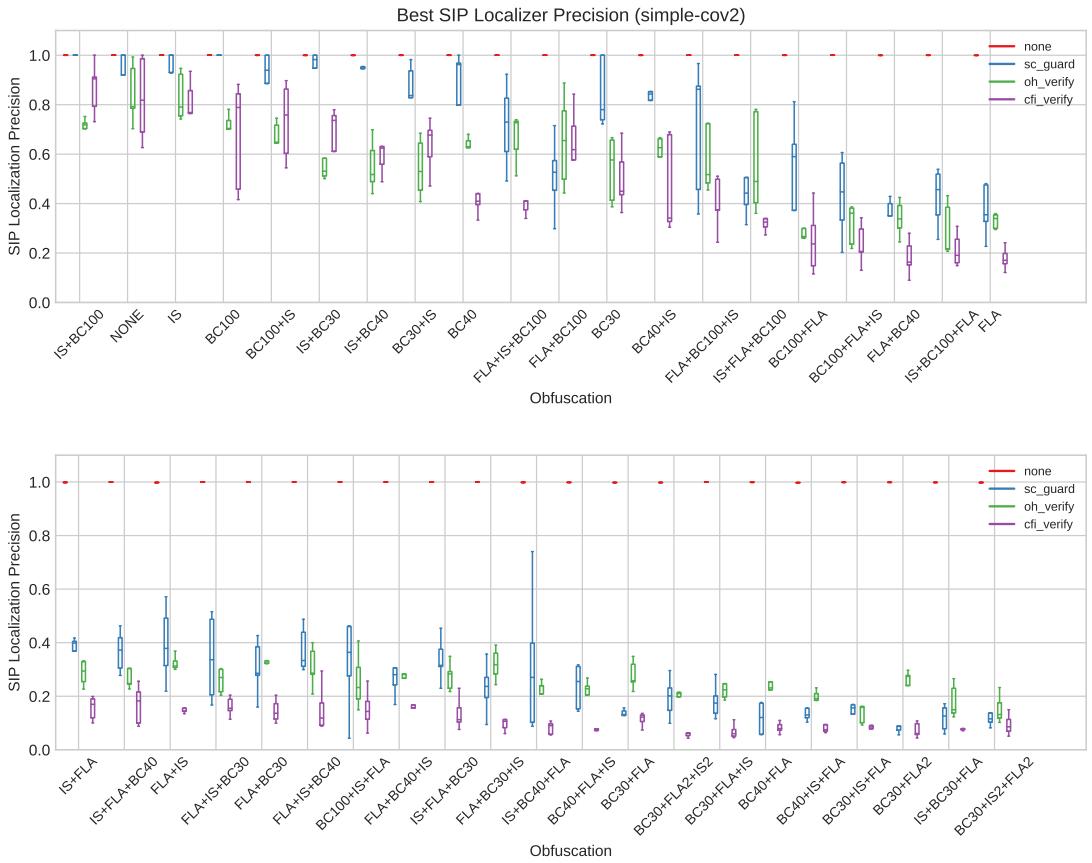


Figure 5.22.: The plot shows SIP localizer model precision score, trained on mibench-cov dataset and tested on the same obfuscation in simple-cov2. Scores are split by protection schemes.

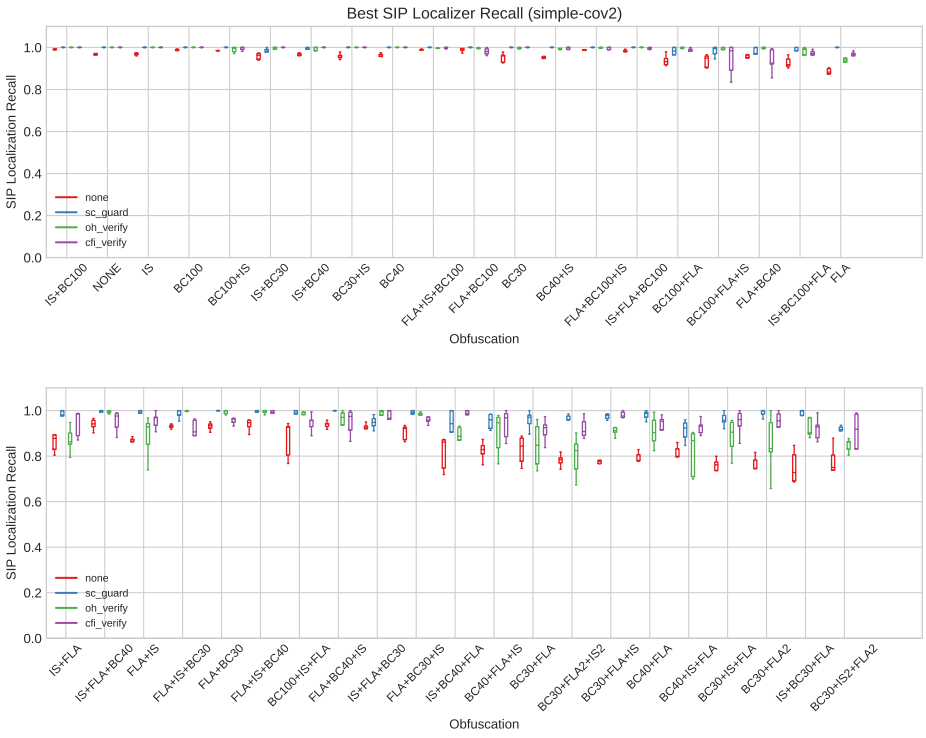


Figure 5.23.: The plot shows SIP localizer model recall score, trained on mibench-cov dataset and tested on the same obfuscation in simple-cov2. Scores are split by protection schemes.

6. Discussion

The following chapter further discusses results from research question experiments and some of the tangential findings.

6.1. Research Question 1

From the figure 5.1, it seems that generally OH protection blocks is more difficult to identify than SC, CFI or plain blocks. However, in some cases, this changes depending on the dataset size and obfuscations.

Even if the attacker only has access to small dataset, no obfuscation or light obfuscations make the program very vulnerable to SIP localization attacks (Figure 5.1).

Our model mostly correctly classifies plain basic blocks, especially on the larger dataset. It only struggles with distinguishing between protection blocks of different types. This suggests that we would get even better results if we trained a binary classifier with (NONE vs. others) classes. In many cases, this type of identification could very well be sufficient for the attacker.

6.2. Research Question 2

We evaluated four different program embeddings (PDG, TF-IDF, Ir2Vec, Code2Vec) and discovered advantages and disadvantages to each.

If the training dataset or computational resources available to the attacker are relatively scarce, then PDG program embedding is preferred. On a larger dataset (mibench-cov), Code2Vec outperforms Ir2Vec on all obfuscations. However, it requires much more computational resources because pre-trained LLVM IR Code2Vec is not available. So, as the dataset size increases, Ir2Vec also becomes a viable choice in case of limited compute. TF-IDF performs very poorly in classifying within protection schemes but has promising results on non-target basic blocks. So it still could be pretty effective in (no protection vs. any protection) binary classification scenario.

Overall, we obtained the best results from combining all the available features. The problem of compact and practical neural program embeddings needs further examination.

6.3. Research Question 3

We run all the experiments under a large number of obfuscations and their combinations. When looking at the obfuscation impact on SIP localization performance, we got very similar results to Wessel 2019 work. Obfuscation is a crucial part of adequate software integrity protection as without it, localizing protection checking parts of the code is trivial even with pattern-matching attacks.

As obfuscation strength increases, machine learning model performance generally goes down. It is also important to note that the order in which the obfuscations are applied has a significant impact in some cases.

We also discovered some patterns that could help develop a new set of obfuscations to test in future work. Generally, it seems that obfuscation combinations where bogus control flow precedes control flow flattening are more effective than those in reversed order. This result is somewhat intuitive because adding bogus nodes becomes much more apparent after the flow has been flattened. They are the ones that do not conform to the flattened structure. This could also explain why stronger BC100 has no advantage over BC30 if they are placed after FLA. More bogus nodes out of the flattened structure make the pattern even more apparent, thus easier to learn for the model.

6.4. Research Question 4

We looked at the same SIP localization performance results from the previous research question and considered the potential cost of the obfuscation this time. Unsurprisingly, stronger obfuscations are generally better. However, we found a subset that is more efficient relative to the program size increase.

Figure 5.11 only shows the most effective protection & obfuscation combinations at any "price" point. We found that different protection and obfuscation choices are appropriate depending on the maximum program size multiplier one can tolerate. For example, at $\times 10$ multiplier SC with S+BC30+FLA is the best choice. In contrast, OH with FLA+IS is the best at $\times 14$.

Additionally, SC protection seems to be suffering from diminishing returns from stronger obfuscations. However, both OH and CFI lines on figure 5.11 seem to suggest that stronger obfuscations could significantly impact model performance even further.

6.5. Research Question 5

Incorporating the k-fold cross validation training procedure allowed us to get SIP localization performance metrics and their variance for a particular train/test split of

the dataset. Without k-fold, we would have only gotten one of these results.

We defined model reliability using the variance of its F1 score between different k-fold iterations. Low standard deviation corresponds to a highly reliable model. The intuition for this definition is based on a hypothetical attacker, using one of our models to attack SIP. Lower the standard deviation of our results, fewer training attempts the attacker would need to be sure that close to maximum performance is obtained.

Our models' results display high standard deviation on any protection labels other than NONE and moderately strong obfuscations, even on the large mibench-cov dataset. Variance is especially high when using a single TF-IDF feature for basic block embeddings. We postulate that this could be improved by doing the local hyper-parameter tuning, meaning that each obfuscation and protection combination should have its own set of hyper-parameters. Some of these parameters are, *learning rate*, *batch size*, *number of epochs*, *GraphSAGE network layer sizes*, *GraphSAGE dropout rate* and *Adam optimizer exponential average decaying speeds*. We can see from some of the hand-picked train/test loss curves (5.20) that final results would likely benefit from increasing the number of epochs.

6.6. Research Question 6

We evaluated the models trained on mibench-cov source programs on the identical obfuscations but simple-cov source programs. Ideally, models would generalize well on the same obfuscation but different source data distribution. We see some generalization, especially on weaker obfuscations and non-target blocks. However, our results (Figure 5.21) show large decrease in performance compared to same models evaluated on mibench-cov (Figure 5.2). This indicates that source programs from mibench-cov are somewhat more similar to each other than to simple-cov programs. Further experiments with a larger, more comprehensive list of source programs are needed to see if better generalization properties can be achieved.

7. Conclusion & Future Work

In this chapter, we re-iterate some of the limitations of this work and discuss potential improvements in future work. We also briefly summarize the main insights and contributions of this paper.

7.1. Limitations & Threats to validity

Despite having quite a large overall dataset in terms of protection & obfuscation variations, the number of initial source programs is still relatively small (total of about 60). Such a small set of starting programs can hardly represent the overall distribution of programs that might be subject to SIP. Though we tried to pick programs with various functionality, it is entirely possible that our results wouldn't generalize well to different source programs.

Our work focused on the particular implementation of protection and obfuscation techniques. We did get strong prediction results, but we can not advocate for the ability of our models to work well on other SIP or obfuscation implementations. However, it is essential to note that we did not make any adjustments to the model or hyperparameter tuning specific to our dataset.

One of the most substantial advantages of deep learning has been end-to-end learning capability. Given a sufficiently large dataset, end-to-end neural models generally outperform handcrafted features. We did not run the optimization process end-to-end despite our features being based on deep learning models (Ir2Vec, Code2Vec). Instead, we split it into feature extraction first and then model optimization based on fixed features. This approach has the advantage of modularity and allowing a combination of features from different sources easily. However, it could also be the limiting factor of our SIP localization scores.

Due to limited computational resources, we did not do the extensive hyper-parameter tuning and set global hyper-parameters for all obfuscation models. Likely, having different parameters for different models automatically set by some searching algorithm would yield significantly better overall results.

We only consider size increase as a cost of integrating protection & obfuscation into a program. This is clearly not a comprehensive approach. In many cases, runtime increase or memory usage could be much more critical.

7.2. Summary

In this work, we implemented a 5-stage machine learning pipeline to generate protected & obfuscated datasets synthetically. Then train & evaluate a deep learning model on this dataset to access plausibility of such machine learning attacks against given SIP-s. We consider a significant strength of this implementation to be the modularity of the approach. Any stage can be easily modified, including new source programs, additional obfuscation combinations, different features, and even models can be easily integrated without considering other pipeline stages. We make all the code available in a public Github[1] repository. Together with all the experimental results, the entire dataset can be generated with a single script from `/sip_ml_pipeline` directory.

We hope to lay the ground for much future research based on our pipeline components.

To the best of our knowledge, this work is the first to extend Code2Vec model into LLVM IR language embeddings successfully. This could potentially be very useful in other fields, where deep learning-based program analysis is needed.

We made a number of conclusions from our experiments from the integrity protection and the attacker's side.

7.2.1. Software Integrity Protection

Having pure software integrity protection does not increase the attacker's cost to tamper with the program, but only when the program is obfuscated too. The complexity of the obfuscation generally determines how good deep learning-based SIP localization attacks will perform. However, some combinations of protections and obfuscations are quite inefficient in program size increase and should be avoided. Figure 5.11 shows a map that allows to choose the most efficient protection & obfuscation based on maximum tolerable program size multiplier. When creating a novel obfuscation combination is better to put bogus control flow transformation before control flow flattening.

7.2.2. Attacking SIP

It is relatively easy to localize SIP protection blocks in non-obfuscated programs using pattern-matching from the attacker's side. When the program is obfuscated, a machine learning attack can be very successful. Generally, deep learning-based approaches work better. If a large dataset or possibility of generating it is available, then Code2Vec is a good choice. In a limited dataset or computation scenario, Ir2Vec works better. Because

[1] https://github.com/tum-i4/sipvsml

we saw a high variance of results in many cases, the attacker should consider running the training process multiple times and picking the best model, but hyper-parameter tuning could alleviate this issue.

7.3. Future Work

A noticeable improvement of the current work would include a more extensive, more diverse set of source programs. This would give better chances of both SIP protection and attacking it to generalize in the real world.

One of the most significant limitations of our results seems to be the lack of an obfuscation-dependent hyper-parameter search process. Adding this component to the existing pipeline could significantly improve the results obtained here.

Our current approach was limited to static code analysis only. Running the program at least once and combining the control flow path or memory footprint with existing features could be a more robust methodology for approaching the SIP localization problem.

Furthermore, one could try to additional pre-processing steps over the dataset by using readily available LLVM optimizers. Some obfuscation transformations could be made less complex, making it easier for the model to learn patterns.

We also see that several general obfuscation structures perform better than others. One could capitalize on these findings to build a new set of more potent and more efficient obfuscation combinations.

Finally, on some of the weaker obfuscations, we see that the SIP localizer has near-perfect scores. The model localizes integrity check conditions with very high accuracy. Natural continuation over this would be to implement a deep learning-based, automatic SIP purging program.

A. Appendix

A.1. All the obfuscation statistics

Obfuscation	Blocks	Avg IR Lines / Program	Avg % IR Lines Incr.
NONE	8,167	1075.40	0.00
SUB	8,167	1286.68	19.65
FLA	22,594	2222.54	106.67
BCF30	21,643	2249.99	109.22
FLA-SUB	22,594	2459.44	128.70
SUB-FLA	22,594	2479.41	130.56
SUB-BCF30	21,793	3088.96	187.24
BCF30-SUB	21,487	3208.76	198.38
BCF40	23,122	3966.40	268.83
BCF30-FLA	38,441	4529.66	321.21
SUB-BCF30-FLA	37,542	4975.28	362.65
FLA-BCF30	64,858	5391.42	401.34
SUB-BCF40	22,486	5421.24	404.11
BCF30-FLA-SUB	37,711	5726.83	432.53
FLA-SUB-BCF30	64,990	5996.68	457.62
BCF30-SUB-FLA	38,468	6008.89	458.76
BCF40-SUB	23,284	6169.17	473.66
SUB-FLA-BCF30	65,050	6330.79	488.69
BCF30-FLA2	34,015	6482.56	502.81
BCF40-FLA	37,914	7436.54	591.52
BCF100	45,838	7472.11	594.82
FLA-BCF30-SUB	65,374	7574.04	604.30
SUB-BCF40-FLA	38,080	9276.67	762.63
FLA-BCF40	69,733	9520.95	785.34
BCF40-FLA-SUB	38,663	9831.24	814.20
SUB-BCF100	45,838	10506.95	877.03

Continued on next page

Obfuscation	Blocks	Avg IR Lines / Program	Avg % IR Lines Incr.
FLA-SUB-BCF40	68,917	10713.98	896.28
BCF40-SUB-FLA	38,255	10917.32	915.19
SUB-FLA-BCF40	68,977	11132.37	935.19
BCF100-SUB	45,838	11879.84	1004.69
FLA-BCF40-SUB	69,133	13593.67	1164.06
BCF100-FLA	64,889	13797.84	1183.05
SUB-BCF100-FLA	64,889	17337.41	1512.19
BCF30-FLA2-SUB2	33,919	17963.03	1570.36
BCF100-FLA-SUB	64,889	18200.06	1592.40
FLA-BCF100	141,127	18724.81	1641.20
BCF100-SUB-FLA	64,889	20631.71	1818.52
FLA-SUB-BCF100	141,127	21775.17	1924.85
BCF30-SUB2-FLA2	34,044	21945.02	1940.64
SUB-FLA-BCF100	141,127	22062.43	1951.56
FLA-BCF100-SUB	141,127	28232.13	2525.27

Table A.1.: The table lists all the obfuscation combinations and their impact on number of LLVM IR lines without any SIP schemes.

A.2. RQ2 feature combinations

This section includes the extension experiments to RQ2. We conducted the exact same experiments as in RQ2, but with combining increasing number of features. We start with the basic *PDG* features and add **TF-IDF**, **IR2Vec** and **Code2Vec** one by one. The table and diagrams below show SIP localization model results for various obfuscations, datasets and increasing feature combinations.

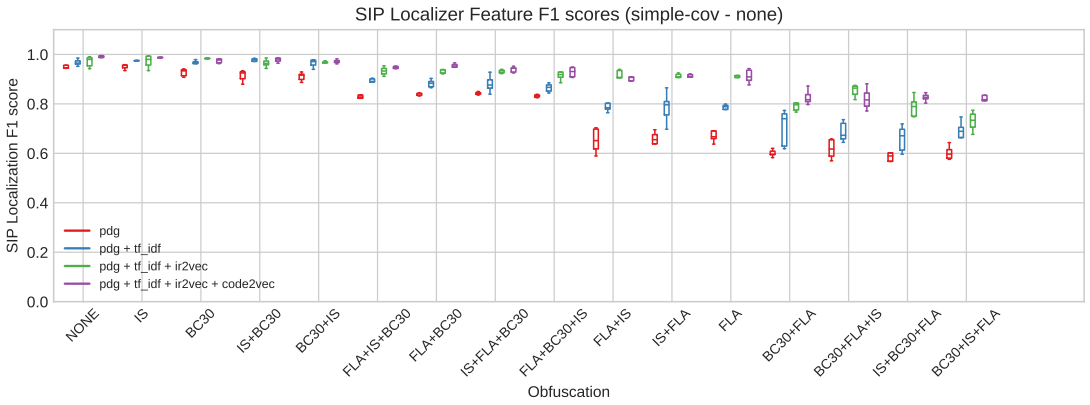


Figure A.1.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis simple-cov F1 score of basic blocks with no protection labels.

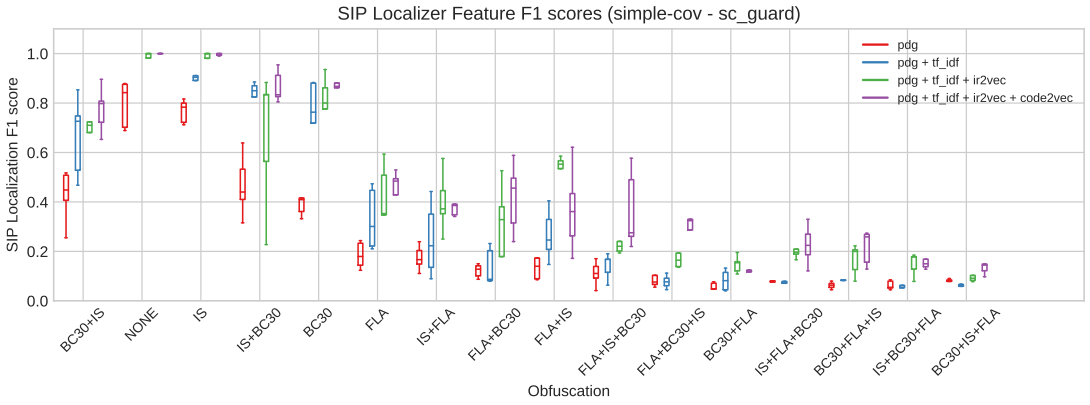


Figure A.2.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis simple-cov F1 score of basic blocks with SC labels.

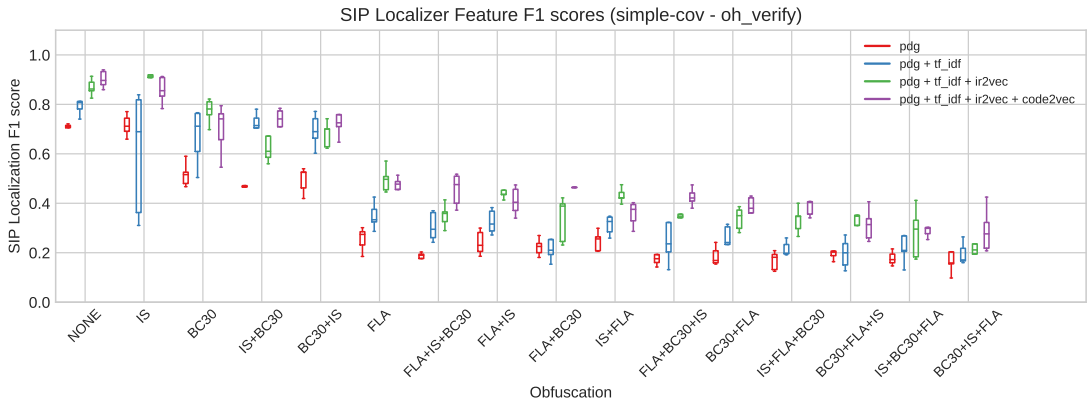


Figure A.3.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis simple-cov F1 score of basic blocks with OH labels.

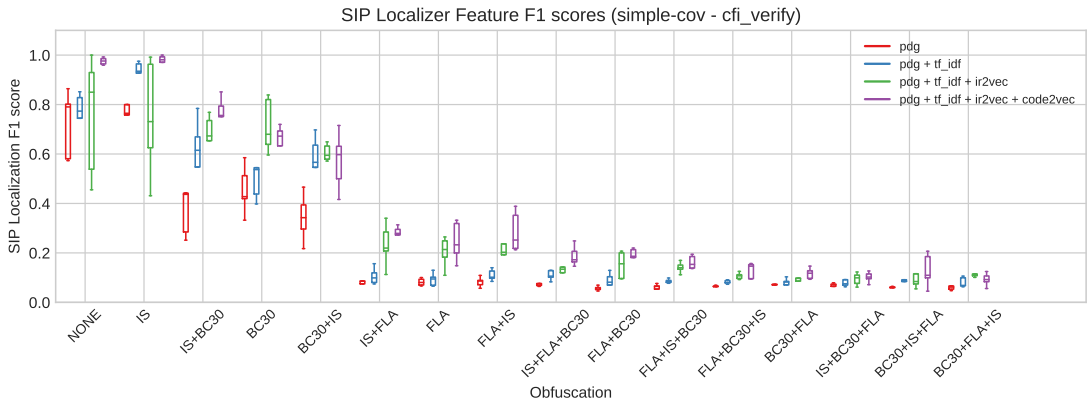


Figure A.4.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis simple-cov F1 score of basic blocks with CFI labels.

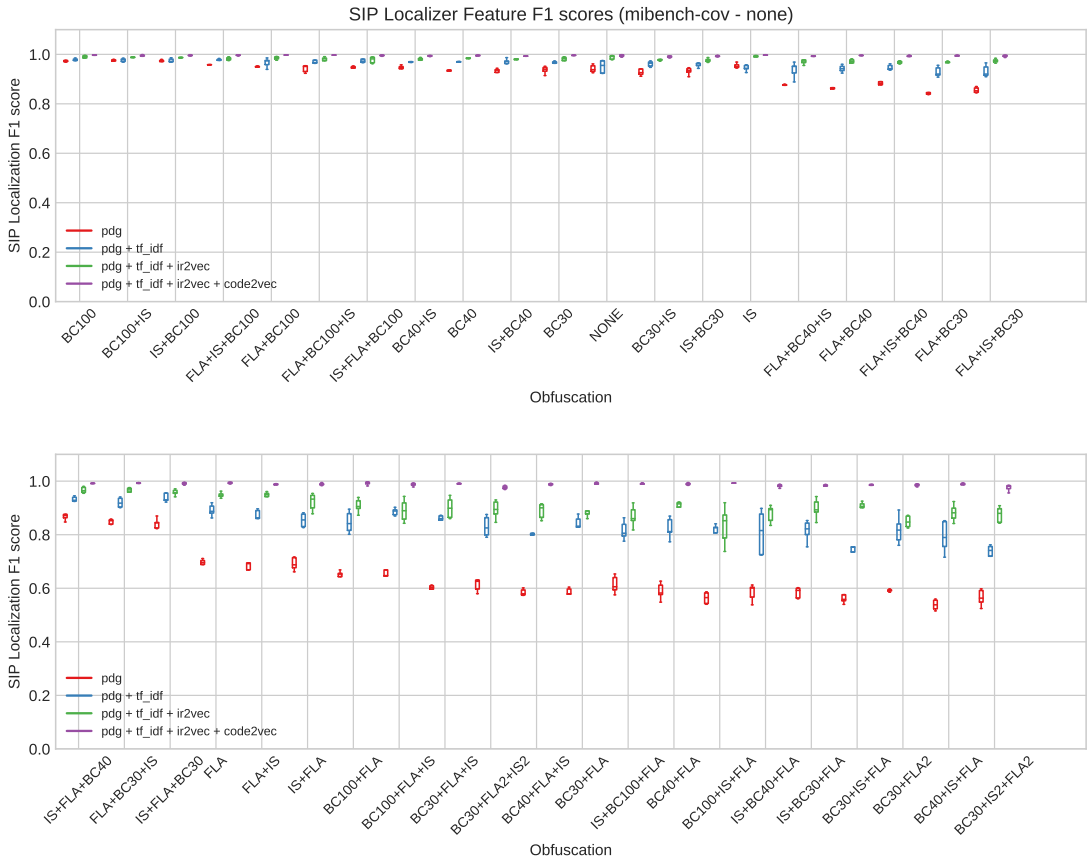


Figure A.5.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis mibench-cov F1 score of basic blocks with no protection labels.

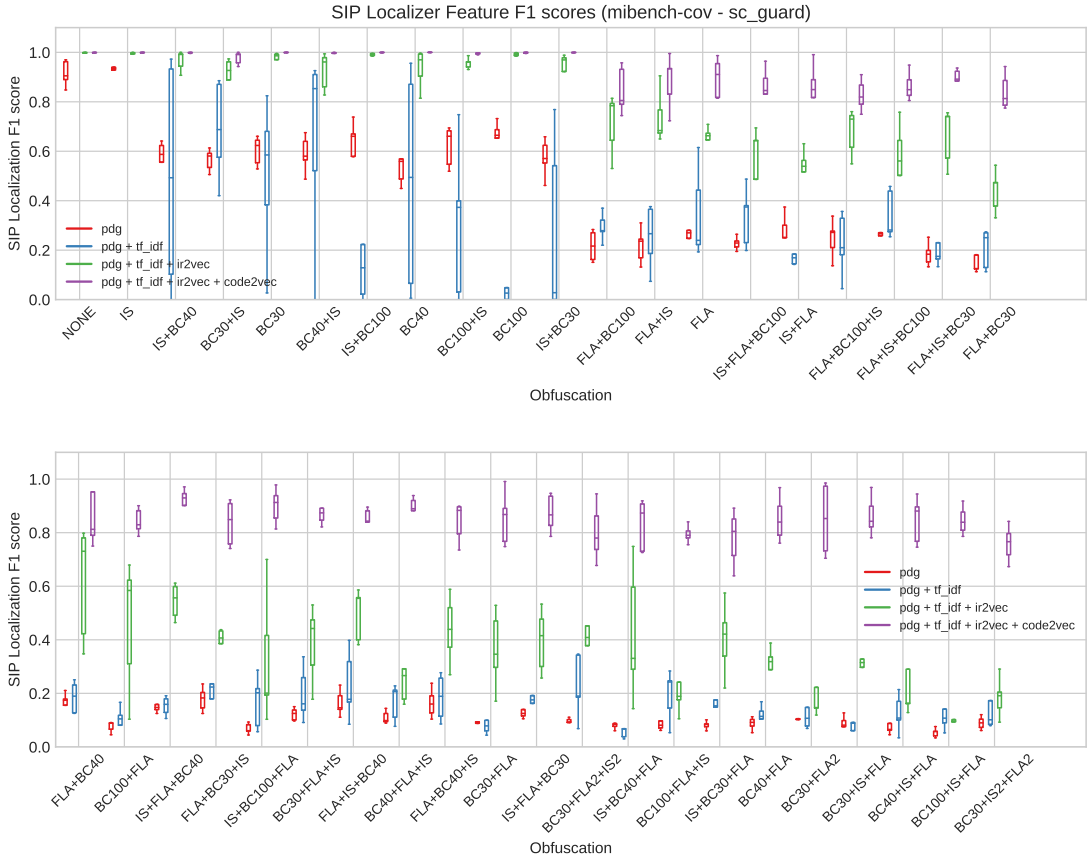


Figure A.6.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis mibench-cov F1 score of basic blocks with SC labels.

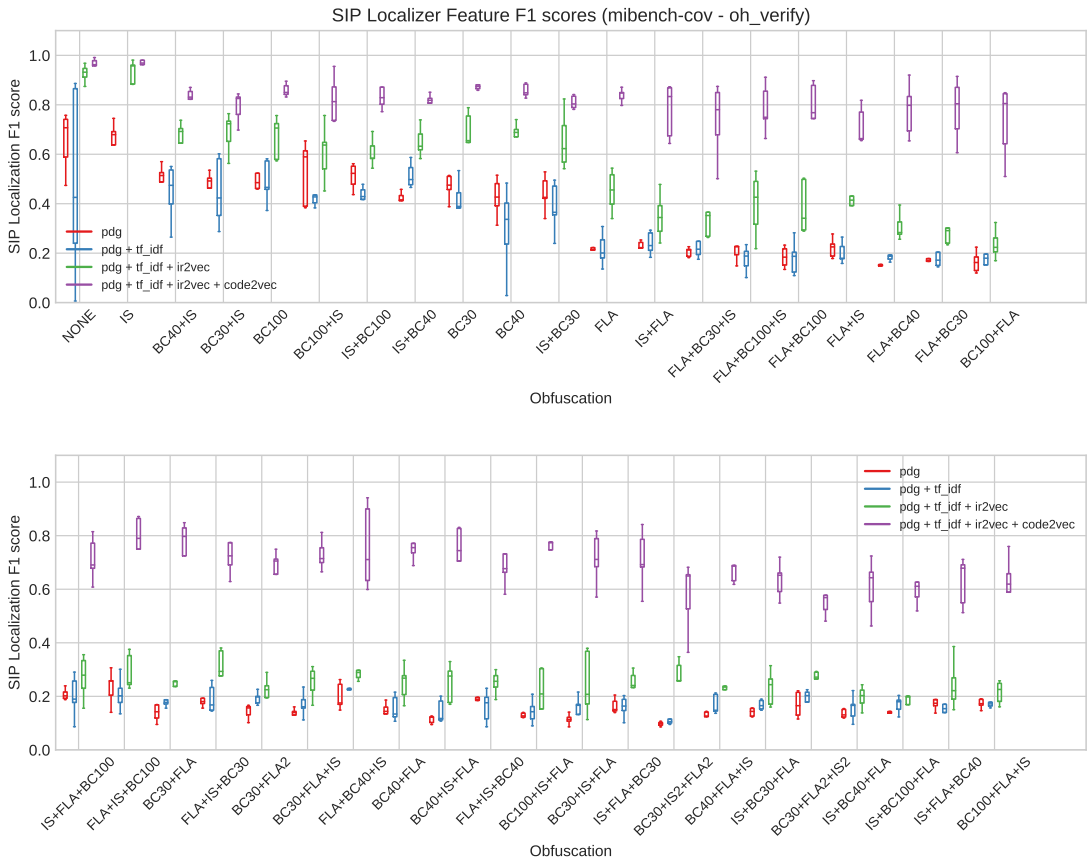


Figure A.7.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis mibench-cov F1 score of basic blocks with OH labels.

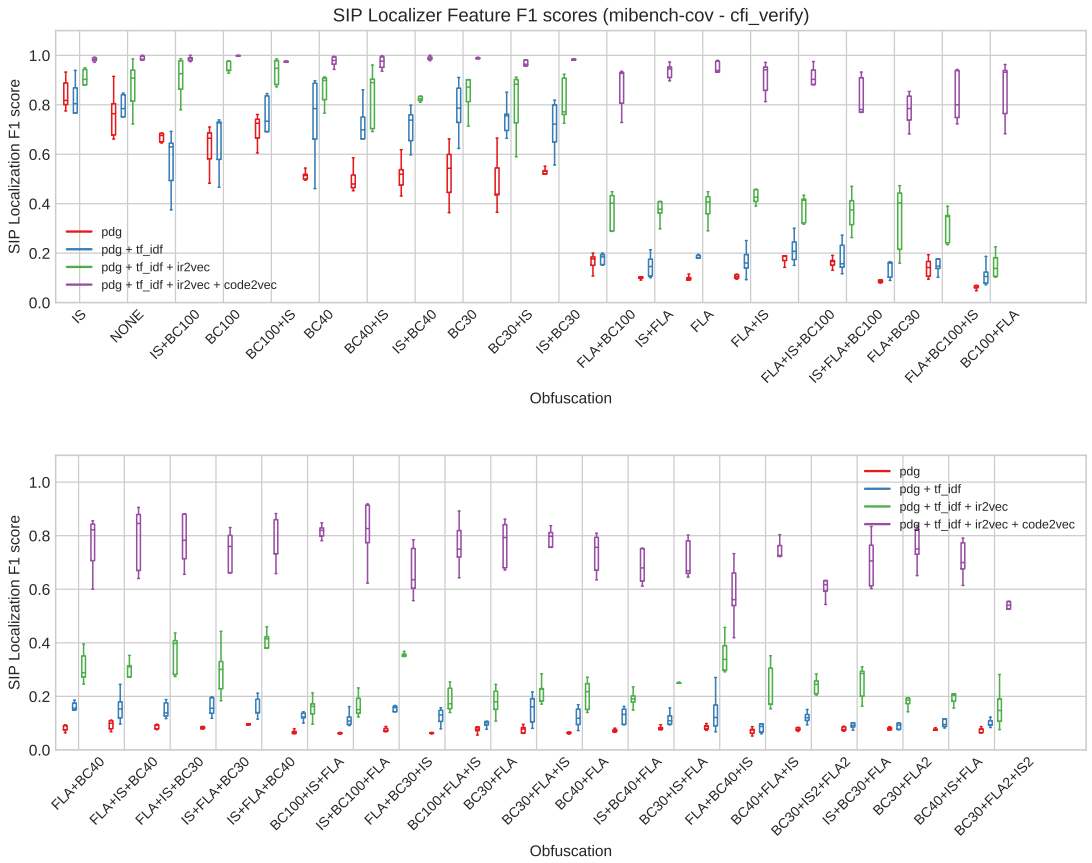


Figure A.8.: The table shows how the SIP localization F1 scores change when gradually adding new features into program representation. X axis shows obfuscations, Y axis mibench-cov F1 score of basic blocks with CFI labels.

A.3. Best model full training results

data_source	obfuscation	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std	
mibench-cov	BC100	0.997	0.001	0.998	0.002	0.994	0.010	0.859	0.026
mibench-cov	BC100+FLA	0.992	0.006	0.843	0.048	0.856	0.125	0.730	0.149
mibench-cov	BC100+FLA+IS	0.987	0.006	0.795	0.032	0.765	0.095	0.618	0.107
mibench-cov	BC100+IS	0.995	0.003	0.995	0.004	0.970	0.022	0.822	0.094
mibench-cov	BC100+IS+FLA	0.989	0.010	0.846	0.053	0.815	0.026	0.712	0.214
mibench-cov	BC30	0.996	0.001	0.999	0.002	0.984	0.012	0.885	0.035
mibench-cov	BC30+FLA	0.988	0.009	0.853	0.099	0.770	0.089	0.745	0.132
mibench-cov	BC30+FLA+IS	0.989	0.005	0.879	0.054	0.746	0.127	0.729	0.056
mibench-cov	BC30+FLA2	0.985	0.005	0.850	0.131	0.759	0.076	0.672	0.083
mibench-cov	BC30+FLA2+IS2	0.977	0.006	0.800	0.105	0.561	0.057	0.568	0.078
mibench-cov	BC30+IS	0.991	0.003	0.977	0.025	0.968	0.013	0.792	0.062
mibench-cov	BC30+IS+FLA	0.987	0.004	0.863	0.073	0.711	0.074	0.715	0.097
mibench-cov	BC30+IS2+FLA2	0.975	0.012	0.759	0.066	0.618	0.058	0.575	0.132
mibench-cov	BC40	0.995	0.002	0.994	0.012	0.975	0.022	0.858	0.027
mibench-cov	BC40+FLA	0.990	0.004	0.852	0.084	0.733	0.077	0.762	0.062
mibench-cov	BC40+FLA+IS	0.988	0.003	0.879	0.068	0.734	0.055	0.691	0.083
mibench-cov	BC40+IS	0.994	0.002	0.992	0.013	0.971	0.027	0.826	0.045
mibench-cov	BC40+IS+FLA	0.985	0.011	0.847	0.086	0.711	0.072	0.694	0.192
mibench-cov	FLA	0.993	0.003	0.897	0.079	0.938	0.047	0.838	0.028
mibench-cov	FLA+BC100	0.996	0.003	0.846	0.093	0.866	0.094	0.765	0.144
mibench-cov	FLA+BC100+IS	0.996	0.002	0.827	0.063	0.831	0.104	0.785	0.098
mibench-cov	FLA+BC30	0.994	0.004	0.841	0.072	0.779	0.070	0.780	0.126
mibench-cov	FLA+BC30+IS	0.991	0.005	0.836	0.084	0.667	0.098	0.737	0.152
mibench-cov	FLA+BC40	0.994	0.003	0.852	0.095	0.765	0.110	0.780	0.107
mibench-cov	FLA+BC40+IS	0.991	0.005	0.842	0.073	0.583	0.120	0.757	0.156
mibench-cov	FLA+IS	0.987	0.005	0.883	0.107	0.907	0.068	0.714	0.076
mibench-cov	FLA+IS+BC100	0.996	0.002	0.864	0.057	0.894	0.078	0.759	0.143
mibench-cov	FLA+IS+BC30	0.993	0.004	0.878	0.073	0.783	0.100	0.746	0.107
mibench-cov	FLA+IS+BC40	0.994	0.002	0.847	0.047	0.788	0.124	0.708	0.113
mibench-cov	IS	0.996	0.003	0.997	0.005	0.983	0.008	0.958	0.033
mibench-cov	IS+BC100	0.996	0.002	0.999	0.001	0.986	0.010	0.829	0.043

Continued on next page

data_source	obfuscation	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std	
mibench-cov	IS+BC100+FLA	0.988	0.005	0.900	0.066	0.811	0.122	0.620	0.095
mibench-cov	IS+BC30	0.993	0.002	0.999	0.001	0.983	0.010	0.810	0.027
mibench-cov	IS+BC30+FLA	0.982	0.006	0.780	0.103	0.704	0.099	0.634	0.066
mibench-cov	IS+BC40	0.994	0.002	0.998	0.002	0.988	0.008	0.816	0.027
mibench-cov	IS+BC40+FLA	0.982	0.006	0.831	0.096	0.685	0.066	0.610	0.102
mibench-cov	IS+FLA	0.989	0.004	0.838	0.127	0.936	0.031	0.779	0.111
mibench-cov	IS+FLA+BC100	0.995	0.002	0.852	0.088	0.790	0.150	0.713	0.082
mibench-cov	IS+FLA+BC30	0.991	0.004	0.873	0.069	0.695	0.166	0.712	0.110
mibench-cov	IS+FLA+BC40	0.992	0.001	0.907	0.071	0.773	0.095	0.629	0.091
mibench-cov	NONE	0.995	0.004	0.999	0.001	0.971	0.046	0.943	0.067
simple-cov	BC30	0.974	0.009	0.845	0.116	0.649	0.075	0.700	0.100
simple-cov	BC30+FLA	0.827	0.029	0.121	0.027	0.117	0.022	0.366	0.077
simple-cov	BC30+FLA+IS	0.821	0.043	0.217	0.069	0.093	0.026	0.313	0.064
simple-cov	BC30+IS	0.972	0.008	0.775	0.092	0.572	0.116	0.736	0.071
simple-cov	BC30+IS+FLA	0.812	0.062	0.146	0.044	0.129	0.066	0.290	0.089
simple-cov	FLA	0.912	0.028	0.442	0.102	0.246	0.079	0.459	0.060
simple-cov	FLA+BC30	0.949	0.018	0.419	0.141	0.176	0.058	0.461	0.043
simple-cov	FLA+BC30+IS	0.929	0.020	0.310	0.126	0.129	0.032	0.425	0.035
simple-cov	FLA+IS	0.908	0.020	0.370	0.172	0.285	0.081	0.409	0.056
simple-cov	FLA+IS+BC30	0.947	0.005	0.365	0.159	0.162	0.027	0.455	0.066
simple-cov	IS	0.987	0.005	0.997	0.005	0.970	0.038	0.859	0.054
simple-cov	IS+BC30	0.978	0.009	0.866	0.064	0.745	0.105	0.704	0.111
simple-cov	IS+BC30+FLA	0.826	0.016	0.175	0.066	0.102	0.021	0.296	0.035
simple-cov	IS+FLA	0.909	0.012	0.386	0.050	0.278	0.032	0.358	0.049
simple-cov	IS+FLA+BC30	0.939	0.012	0.226	0.080	0.187	0.041	0.383	0.032
simple-cov	NONE	0.991	0.004	0.998	0.004	0.976	0.014	0.902	0.034
simple-cov2	BC100	0.994	0.002	0.997	0.006	0.876	0.073	0.798	0.037
simple-cov2	BC100+FLA	0.941	0.040	0.432	0.227	0.216	0.089	0.540	0.146
simple-cov2	BC100+FLA+IS	0.943	0.055	0.536	0.279	0.281	0.125	0.506	0.147
simple-cov2	BC100+IS	0.984	0.004	0.878	0.150	0.640	0.164	0.683	0.096
simple-cov2	BC100+IS+FLA	0.935	0.046	0.389	0.279	0.254	0.110	0.448	0.122
simple-cov2	BC30	0.978	0.006	0.831	0.120	0.666	0.096	0.744	0.058
simple-cov2	BC30+FLA	0.850	0.047	0.164	0.107	0.138	0.027	0.337	0.089
simple-cov2	BC30+FLA+IS	0.852	0.031	0.223	0.077	0.104	0.043	0.377	0.058

Continued on next page

data_source	obfuscation	none		sc_guard		cfi_verify		oh_verify	
	mean	std	mean	std	mean	std	mean	std	
simple-cov2	BC30+FLA2	0.819	0.045	0.115	0.058	0.125	0.033	0.359	0.069
simple-cov2	BC30+FLA2+IS2	0.855	0.019	0.266	0.121	0.106	0.017	0.298	0.058
simple-cov2	BC30+IS	0.971	0.014	0.748	0.178	0.625	0.174	0.738	0.073
simple-cov2	BC30+IS+FLA	0.818	0.024	0.115	0.040	0.124	0.028	0.311	0.048
simple-cov2	BC30+IS2+FLA2	0.774	0.041	0.127	0.055	0.114	0.032	0.201	0.042
simple-cov2	BC40	0.977	0.008	0.840	0.080	0.600	0.133	0.736	0.070
simple-cov2	BC40+FLA	0.770	0.090	0.157	0.181	0.110	0.043	0.348	0.083
simple-cov2	BC40+FLA+IS	0.848	0.028	0.214	0.082	0.098	0.014	0.335	0.016
simple-cov2	BC40+IS	0.977	0.005	0.839	0.056	0.635	0.078	0.705	0.073
simple-cov2	BC40+IS+FLA	0.757	0.051	0.082	0.023	0.110	0.017	0.236	0.058
simple-cov2	FLA	0.919	0.014	0.462	0.104	0.274	0.097	0.430	0.035
simple-cov2	FLA+BC100	0.993	0.004	0.695	0.173	0.603	0.209	0.721	0.108
simple-cov2	FLA+BC100+IS	0.988	0.008	0.664	0.173	0.467	0.200	0.627	0.091
simple-cov2	FLA+BC30	0.959	0.004	0.343	0.109	0.226	0.036	0.500	0.093
simple-cov2	FLA+BC30+IS	0.942	0.018	0.310	0.124	0.166	0.043	0.462	0.030
simple-cov2	FLA+BC40	0.951	0.008	0.290	0.102	0.179	0.054	0.406	0.056
simple-cov2	FLA+BC40+IS	0.964	0.015	0.423	0.145	0.254	0.091	0.453	0.087
simple-cov2	FLA+IS	0.924	0.014	0.421	0.091	0.268	0.053	0.445	0.045
simple-cov2	FLA+IS+BC100	0.994	0.002	0.743	0.183	0.621	0.133	0.745	0.094
simple-cov2	FLA+IS+BC30	0.938	0.013	0.355	0.079	0.147	0.048	0.428	0.082
simple-cov2	FLA+IS+BC40	0.962	0.028	0.533	0.054	0.272	0.168	0.464	0.107
simple-cov2	IS	0.989	0.005	0.999	0.003	0.963	0.033	0.879	0.056
simple-cov2	IS+BC100	0.993	0.003	0.998	0.003	0.886	0.049	0.754	0.062
simple-cov2	IS+BC100+FLA	0.939	0.041	0.472	0.250	0.267	0.145	0.426	0.126
simple-cov2	IS+BC30	0.976	0.009	0.847	0.057	0.626	0.153	0.736	0.047
simple-cov2	IS+BC30+FLA	0.809	0.038	0.105	0.016	0.116	0.015	0.334	0.037
simple-cov2	IS+BC40	0.980	0.004	0.814	0.094	0.693	0.054	0.734	0.059
simple-cov2	IS+BC40+FLA	0.796	0.037	0.108	0.045	0.113	0.032	0.305	0.053
simple-cov2	IS+FLA	0.888	0.021	0.386	0.159	0.178	0.017	0.426	0.054
simple-cov2	IS+FLA+BC100	0.989	0.005	0.605	0.124	0.484	0.124	0.590	0.166
simple-cov2	IS+FLA+BC30	0.951	0.008	0.271	0.081	0.238	0.047	0.387	0.025
simple-cov2	IS+FLA+BC40	0.956	0.019	0.492	0.178	0.171	0.022	0.452	0.136
simple-cov2	NONE	0.987	0.005	0.964	0.063	0.965	0.021	0.872	0.051

A.4. Best Protections given an obfuscation

The tables below contain the best protection schemes and their F1 score given an obfuscation.

obfuscation	Best Protection	F1 score mean	F1 score std
BC30+FLA+IS	cfi_verify	0.093	0.026
IS+BC30+FLA	cfi_verify	0.102	0.021
BC30+FLA	cfi_verify	0.117	0.022
BC30+IS+FLA	cfi_verify	0.129	0.066
FLA+BC30+IS	cfi_verify	0.129	0.032
FLA+IS+BC30	cfi_verify	0.162	0.027
FLA+BC30	cfi_verify	0.176	0.058
IS+FLA+BC30	cfi_verify	0.187	0.041
FLA	cfi_verify	0.246	0.079
IS+FLA	cfi_verify	0.278	0.032
FLA+IS	cfi_verify	0.285	0.081
BC30+IS	cfi_verify	0.572	0.116
BC30	cfi_verify	0.649	0.075
IS+BC30	oh_verify	0.704	0.111
IS	oh_verify	0.859	0.054
NONE	oh_verify	0.902	0.034

Table A.3.: The list of obfuscations in the simple-cov dataset and the best protection results against the ML model with the strongest features. Rows are sorted ascending by mean F1 scores

obfuscation	Best Protection	F1 score mean	F1 score std
BC30+FLA2+IS2	cfi_verify	0.561	0.057
BC30+IS2+FLA2	oh_verify	0.575	0.132
FLA+BC40+IS	cfi_verify	0.583	0.120
IS+BC40+FLA	oh_verify	0.610	0.102
BC100+FLA+IS	oh_verify	0.618	0.107

Continued on next page

obfuscation	Best Protection	F1 score mean	F1 score std
IS+BC100+FLA	oh_verify	0.620	0.095
IS+FLA+BC40	oh_verify	0.629	0.091
IS+BC30+FLA	oh_verify	0.634	0.066
FLA+BC30+IS	cfi_verify	0.667	0.098
BC30+FLA2	oh_verify	0.672	0.083
BC40+FLA+IS	oh_verify	0.691	0.083
BC40+IS+FLA	oh_verify	0.694	0.192
IS+FLA+BC30	cfi_verify	0.695	0.166
FLA+IS+BC40	oh_verify	0.708	0.113
BC30+IS+FLA	cfi_verify	0.711	0.074
BC100+IS+FLA	oh_verify	0.712	0.214
IS+FLA+BC100	oh_verify	0.713	0.082
FLA+IS	oh_verify	0.714	0.076
BC30+FLA+IS	oh_verify	0.729	0.056
BC100+FLA	oh_verify	0.730	0.149
BC40+FLA	cfi_verify	0.733	0.077
BC30+FLA	oh_verify	0.745	0.132
FLA+IS+BC30	oh_verify	0.746	0.107
FLA+IS+BC100	oh_verify	0.759	0.143
FLA+BC100	oh_verify	0.765	0.144
FLA+BC40	cfi_verify	0.765	0.110
IS+FLA	oh_verify	0.779	0.111
FLA+BC30	cfi_verify	0.779	0.070
FLA+BC100+IS	oh_verify	0.785	0.098
BC30+IS	oh_verify	0.792	0.062
IS+BC30	oh_verify	0.810	0.027
IS+BC40	oh_verify	0.816	0.027
BC100+IS	oh_verify	0.822	0.094
BC40+IS	oh_verify	0.826	0.045
IS+BC100	oh_verify	0.829	0.043
FLA	oh_verify	0.838	0.028
BC40	oh_verify	0.858	0.027
BC100	oh_verify	0.859	0.026

Continued on next page

obfuscation	Best Protection	F1 score mean	F1 score std
BC30	oh_verify	0.885	0.035
NONE	oh_verify	0.943	0.067
IS	oh_verify	0.958	0.033

Table A.4.: The list of obfuscations in the mibench-cov dataset and the best protection results against the ML model with the strongest features. Rows are sorted ascending by mean F1 scores

List of Figures

List of Tables

Bibliography

1. Abadi, M., Budiu, M., Erlingsson, Ú. & Ligatti, J. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* **13.** ISSN: 1094-9224 (Nov. 2009).
2. Abrath, B., Coppens, B., Broeck, J. V. D., Wyseur, B., Cabutto, A., Falcarin, P. & Sutter, B. D. Code Renewability for Native Software Protection. *ACM Trans. Priv. Secur.* **23.** ISSN: 2471-2566 (Aug. 2020).
3. Ahmadvand, M., Fischer, D. & Banescu, S. SIP Shaker: Software Integrity Protection Composition. *CoRR* **abs/1909.11401.** arXiv: 1909.11401 (2019).
4. Ahmadvand, M., Hayrapetyan, A., Banescu, S. & Pretschner, A. *Practical Integrity Protection with Oblivious Hashing* in *Proceedings of the 34th Annual Computer Security Applications Conference* (Association for Computing Machinery, San Juan, PR, USA, 2018), 40–52. ISBN: 9781450365697.
5. Ahmadvand, M., Pretschner, A. & Kelbert, F. in (Jan. 2018).
6. Aizawa, A. An information-theoretic perspective of tf–idf measures. *Information Processing & Management* **39,** 45–65. ISSN: 0306-4573 (2003).
7. Akhunzada, A., Sookhak, M., Anuar, N. B., Gani, A., Ahmed, E., Shiraz, M., Furnell, S., Hayat, A. & Khurram Khan, M. Man-At-The-End Attacks. *J. Netw. Comput. Appl.* **48,** 44–57. ISSN: 1084-8045 (Feb. 2015).
8. Alon, U., Zilberstein, M., Levy, O. & Yahav, E. *code2vec: Learning Distributed Representations of Code* 2018. arXiv: 1803.09473 [cs.LG].
9. Amir, I., Koren, T. & Livni, R. *SGD Generalizes Better Than GD (And Regularization Doesn't Help)* 2021. arXiv: 2102.01117 [cs.LG].
10. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. & Yang, K. On the (Im)possibility of Obfuscating Programs. *IACR Cryptology ePrint Archive* **2001,** 69 (Jan. 2001).
11. Barry, K. *CR Engineers Show a Tesla Will Drive With No One in the Driver's Seat* https://www.consumerreports.org/autonomous-driving/cr-engineers-show-tesla-will-drive-with-no-one-in-drivers-seat/. [Online; accessed 02-May-2021]. Apr. 2021.

12. Ben-Nun, T., Jakobovits, A. S. & Hoefler, T. in *Advances in Neural Information Processing Systems 31* (eds Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N. & Garnett, R.) 3588–3600 (Curran Associates, Inc., 2018).

13. Bengio, Y., Courville, A. & Vincent, P. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **35,** 1798–1828 (2013).

14. Bishop, C. *Pattern Recognition and Machine Learning (Information Science and Statistics)* isbn: 0387310738 (Berlin, Heidelberg: Springer-Verlag, 2006).

15. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B. & Varoquaux, G. *API design for machine learning software: experiences from the scikit-learn project* in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning* (2013), 108–122.

16. Burke, M. G. & Cytron, R. K. Interprocedural Dependence Analysis and Parallelization. *SIGPLAN Not.* **39,** 139–154. ISSN: 0362-1340 (Apr. 2004).

17. Caruana, R. & Niculescu-Mizil, A. *Data Mining in Metric Space: An Empirical Analysis of Supervised Learning Performance Criteria* in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery, Seattle, WA, USA, 2004), 69–78. ISBN: 1581138881.

18. Chang, H. & Atallah, M. J. *Protecting Software Code by Guards* in *Security and Privacy in Digital Rights Management* (ed Sander, T.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 2002), 160–175. ISBN: 978-3-540-47870-6.

19. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S. & Jakubowski, M. *Oblivious Hashing: A Stealthy Software Integrity Verification Primitive* in (Oct. 2002), 400–414. ISBN: 978-3-540-00421-9.

20. Collberg, C. & Thomborson, C. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on Software Engineering* **28,** 735–746 (Aug. 2002).

21. Collberg, C., Thomborson, C. & Low, D. *A Taxonomy of Obfuscating Transformations* 1997.

22. Collberg, C., Thomborson, C. & Low, D. *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs* in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Association for Computing Machinery, San Diego, California, USA, 1998), 184–196. ISBN: 0897919793.

23. Costan, V. & Devadas, S. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* **2016,** 86 (2016).

24. Data61, C. *StellarGraph Machine Learning Library* `https://github.com/stellargraph/stellargraph`. 2018.

25. Dedić, N., Jakubowski, M. & Venkatesan, R. *A Graph Game Model for Software Tamper Protection* in *Proceedings of the 9th International Conference on Information Hiding* (Springer-Verlag, Saint Malo, France, 2007), 80–95. ISBN: 354077369X.

26. Dietterich, T. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* **27,** 326–327. ISSN: 0360-0300 (Sept. 1995).

27. Eklind, R. *LLVM IR and Go* `https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/`. [Online; accessed 12-May-2021]. Dec. 2018.

28. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M. & Boneh, D. *Terra: a virtual machine-based platform for trusted computing* in (ACM Press, 2003), 193–206.

29. Giffin, J., Christodorescu, M. & Kruger, L. *Strengthening software self-checksumming via self-modifying code* in *21st Annual Computer Security Applications Conference (ACSAC'05)* (2005).

30. Goldberg, Y. *A Primer on Neural Network Models for Natural Language Processing* 2015. arXiv: `1510.00726 [cs.CL]`.

31. Gomez-Uribe, C. A. & Hunt, N. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Manage. Inf. Syst.* **6.** ISSN: 2158-656X (Dec. 2016).

32. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T. & Brown, R. *MiBench: A free, commercially representative embedded benchmark suite* in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)* (2001), 3–14.

33. Hamilton, W. L., Ying, R. & Leskovec, J. *Inductive Representation Learning on Large Graphs* in *NIPS* (2017).

34. Jozefowicz, R., Zaremba, W. & Sutskever, I. *An Empirical Exploration of Recurrent Network Architectures* in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (JMLR.org, Lille, France, 2015), 2342–2350.

35. Junod, P., Rinaldini, J., Wehrli, J. & Michielin, J. *Obfuscator-LLVM – Software Protection for the Masses* in *2015 IEEE/ACM 1st International Workshop on Software Protection* (2015), 3–9.

36. Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M. & Tang, P. T. P. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR* **abs/1609.04836.** arXiv: `1609.04836` (2016).

37. Kingma, D. P. & Ba, J. *Adam: A Method for Stochastic Optimization* 2017. arXiv: 1412.6980 [cs.LG].

38. Kipf, T. N. & Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* **abs/1609.02907.** arXiv: 1609.02907 (2016).

39. Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S. & Willing, C. *Jupyter Notebooks – a publishing format for reproducible computational workflows* in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (eds Loizides, F. & Schmidt, B.) (2016), 87–90.

40. Larrañaga, P., Calvo, B., Santana, R., Bielza, C., Galdiano, J., Inza, I., Lozano, J. A., Armañanzas, R., Santafé, G., Pérez, A. & Robles, V. Machine learning in bioinformatics. *Briefings in Bioinformatics* **7,** 86–112. ISSN: 1467-5463. eprint: https://academic.oup.com/bib/article-pdf/7/1/86/23992771/bbk007.pdf (Mar. 2006).

41. Lattner, C. & Adve, V. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation* in (San Jose, CA, USA, Mar. 2004), 75–88.

42. Le, Q. V. & Mikolov, T. *Distributed Representations of Sentences and Documents* 2014. arXiv: 1405.4053 [cs.CL].

43. Leung, A. & George, L. Static Single Assignment Form for Machine Code. *SIGPLAN Not.* **34,** 204–214. ISSN: 0362-1340 (May 1999).

44. Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu & Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015.

45. Masters, D. & Luschi, C. *Revisiting Small Batch Training for Deep Neural Networks* 2018. arXiv: 1804.07612 [cs.LG].

46. Mikolov, T., Chen, K., Corrado, G. & Dean, J. *Efficient Estimation of Word Representations in Vector Space* 2013. arXiv: 1301.3781 [cs.CL].

47. Moghimi, D., Sunar, B., Eisenbarth, T. & Heninger, N. *TPM-FAIL: TPM meets Timing and Lattice Attacks* 2019. arXiv: 1911.05673 [cs.CR].

48. Ngabonziza, B., Martin, D., Bailey, A., Cho, H. & Martin, S. *TrustZone Explained: Architectural Features and Use Cases* in (Nov. 2016), 445–451.

49. Nilsson, A., Bideh, P. N. & Brorsson, J. A Survey of Published Attacks on Intel SGX. *CoRR* **abs/2006.13598.** arXiv: 2006.13598 (2020).

50. Nuutila, E. & Soisalon-soininen, E. On Finding the Strong Components in a Directed Graph (Nov. 1995).

51. Pan, S. & Yang, Q. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* **22,** 1345–1359 (2010).

52. Qiu, J., Yadegari, B., Johannesmeyer, B., Debray, S. & Su, X. *Identifying and Understanding Self-Checksumming Defenses in Software* in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (Association for Computing Machinery, San Antonio, Texas, USA, 2015), 207–218. ISBN: 9781450331913.

53. Ram, P. & Gray, A. G. *Density Estimation Trees* in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery, San Diego, California, USA, 2011), 627–635. ISBN: 9781450308137.

54. Rokach, L. & Maimon, O. in *Data Mining and Knowledge Discovery Handbook* (eds Maimon, O. & Rokach, L.) 321–352 (Springer US, Boston, MA, 2005). ISBN: 978-0-387-25465-4.

55. Salem, A. & Banescu, S. *Metadata Recovery from Obfuscated Programs Using Machine Learning* in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering* (Association for Computing Machinery, Los Angeles, California, USA, 2016). ISBN: 9781450348416.

56. Salem, A. & Banescu, S. *Metadata Recovery from Obfuscated Programs Using Machine Learning* in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering* (Association for Computing Machinery, Los Angeles, California, USA, 2016). ISBN: 9781450348416.

57. Salton, G. & McGill, M. J. Introduction to modern information retrieval (1986).

58. Scanzio, S., Cumani, S., Gemello, R., Mana, F. & Laface, P. *Parallel implementation of artificial neural network training* in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing* (2010), 4902–4905.

59. Schmidhuber, J. Deep Learning in Neural Networks: An Overview. *Neural Networks* **61.** Published online 2014; based on TR arXiv:1404.7828 [cs.NE], 85–117 (2015).

60. Shinde, P. P. & Shah, S. *A Review of Machine Learning and Deep Learning Applications* in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)* (2018), 1–6.

61. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature* **529,** 484–503 (2016).

62. Sonoda, S. & Murata, N. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis* **43,** 233–268. ISSN: 1063-5203 (2017).

63. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* **15,** 1929–1958. ISSN: 1532-4435 (Jan. 2014).

64. Sui, Y. & Xue, J. *SVF: interprocedural static value-flow analysis in LLVM* in (Mar. 2016), 265–266.

65. Thimm, G. & Fiesler, E. *Neural network initialization* in *From Natural to Artificial Neural Computation* (eds Mira, J. & Sandoval, F.) (Springer Berlin Heidelberg, Berlin, Heidelberg, 1995), 535–542. ISBN: 978-3-540-49288-7.

66. Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z. & Guyon, I. *Bayesian Optimization is Superior to Random Search for Machine Learning Hyperparameter Tuning: Analysis of the Black-Box Optimization Challenge 2020* 2021. arXiv: 2104.10201 [cs.LG].

67. Van Bulck, J., Piessens, F. & Strackx, R. *Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic* in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Association for Computing Machinery, Toronto, Canada, 2018), 178–195. ISBN: 9781450356930.

68. VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M. S., Upadrasta, R. & Srikant, Y. N. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* **17.** ISSN: 1544-3566 (Dec. 2020).

69. Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E. & Andina, D. Deep Learning for Computer Vision: A Brief Review. *Intell. Neuroscience* **2018.** ISSN: 1687-5265 (Jan. 2018).

70. Wehle, H.-D. *Machine Learning, Deep Learning, and AI: What's the Difference?* in (July 2017).

71. Wessel, D. *Resilience of SIP against ML-based attacks* Master's Thesis (Technical University of Munich, Arcisstraße 21, 80333 München, Germany, 2019).

72. Zheng, A. & Casari, A. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists* 1st. ISBN: 1491953241 (O'Reilly Media, Inc., 2018).

73. Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H. & He, Q. A Comprehensive Survey on Transfer Learning. *CoRR* **abs/1911.02685.** arXiv: 1911.02685 (2019).