Scalable EM-Based Anomaly Detection using Generative Adversarial Networks

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Kurt A. Vedros

Major Professors: **Dr. Konstantinos Kolias, and Dr. Min Xian**

Committee Members: **Dr. Alex Vakanski**, and **Dr. Robert C. Ivans**

Department Administrator: **Dr. Terence Soule**

May 2023

Abstract

Embedded devices are omnipresent in modern networks, including those facilitating mission-critical applications. However, due to their constrained nature, novel mechanisms are required to provide external, and non-intrusive defenses. Among such approaches, one that has gained traction is based on analyzing the emanated electromagnetic (EM) signals. Unfortunately, one of the most neglected challenges of this approach is the manual gathering and fingerprinting of the corresponding EM signals. Indeed, even simple programs are comprised of numerous branches, making the fingerprinting stage extremely time-consuming, and requiring the manual labor of an expert. To address this issue, we first considered manually synthesizing EM directly from machine code. However, such an approach requires an exhaustive capturing process not for entire execution paths but rather the "building blocks" of those. In this context, "building blocks" can be defined as instruction sequences. For this reason, we propose proposed an automated, data-driven approach for generating EM signals from machine code using Generative Adversarial Networks (GANs). In comparison to the previous approach, synthetically generating EM signals also removes the need for an elaborate and error-prone fingerprinting stage while requiring a fraction of captured signals. Preliminary, small-scale experimental evaluations indicate that our GANs-based approach provides near to perfect detection accuracy against code injection attacks when considering the full signal.

Keywords: Side-Channel Analysis; Anomaly Detection; Electromagnetic Signals; Synthetic Signals; Generated Signals; Generative Adversarial Networks

Acknowledgments

First and foremost, I would like to thank my Major Advisors, Dr. Konstantinos Kolias and Dr. Min Xian, for their mentoring. Furthermore, the experience and opportunities they have provided me throughout my academic years has been invaluable. I look forward to continuing to working with them while I work towards my Doctorate Degree. I would like to also thank the members of my committee for dedicating time and effort towards my goals and education. Finally, I would also like to thank the faculty and staff at the University of Idaho for a friendly, supportive and intellectually stimulating experience. It has been a joy to work with everyone at the Idaho Falls campus.

This study was supported by funds from the Department of Energy under U.S. DOE Idaho Operations Office Contract DE-AC07-05ID14517, as part of Laboratory Directed Research and Development, Resilient Control and Instrumentation Systems (ReCIS) program of Idaho National Laboratory.

Dedication

Most importantly, I would like to thank my parents, Kurt G. Vedros and Donna M. Vedros, for all their love and support throughout my life. I cannot thank them enough for what they have done to provide me with a quality education, bountiful opportunities, and a loving home. I would also like to show my gratitude towards my sisters Kristin, Kira, and Katherine for their encouragement and for always being someone to talk to and decompress with.

Finally, to all of my family near and far, **thank you for always being there for me.**

Table of Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

EM	Electromagnetic
NN	Neural Network
ANN	Artificial Neural Network
CNF	Continuous Normalizing Flows
VAE	Variational Autoencoders
GANs	Generative Adversarial Networks
WGANs-GP	Wasserstein GANs with Gradient Penalty
GANs-COD2EM	GANs for Code to Electromagnetic Signal Translation
ASM	Assembly
ED	Euclidean Distance
N-IDS	Network Intrusion Detection Systems
PLC	Programmable Logic Controllers
COTS	Commercial Off-The-Shelf
ICs	Integrated Circuits
AUC	Area Under the Curve
ROC	Receiver Operating Characteristic
ACC	Accuracy
TPR	True Positive Rate

FPR	False Positive Rate
PPV	Precision

Chapter 1

Introduction

Nowadays, a large portion of corporate, government, military networks, and critical infrastructures consists of *embedded devices*. Typically, these mission-critical assets are severely constrained in terms of processing, memory, and energy resources. Since standard cryptographic algorithms were designed according to the hardware specification of high-end systems, traditional crypto libraries, and the corresponding protection tools are not applicable to such environments (at least not without modifications). At the same time, in many cases, embedded devices are directly exposed to the Internet and its cyber threats. Therefore, there is a dire need for the development of novel security mechanisms specifically designed to respect the limitations and peculiarities of such critical systems.

As a potential solution to this problem, researchers have relied on the analysis of patterns of analog signals emitted by the CPU of embedded devices. In this context, such signals are considered a *side-channel* because they get emitted involuntarily by devices during their regular operation. Even though these analog signals are often treated as noise in most applications, they may bear valuable information. In principle, certain characteristics of the emitted analog signals have a strong correlation to the instructions being executed by the CPU. Thus, numerous *side-channel-based anomaly detection* approaches have been proposed particularly to provide external protection for embedded devices [7], [8], [9], [10], [11].

Today, the dominant methods of side-channel-based anomaly detection rely on the analysis of *power-consumption patterns* [8], [11]. This is primarily due to the ease of data collection and the robustness of this modality against environmental noise. Nevertheless, when compared to power-based approaches, *electromagnetic (EM) based* methods are theoretically more advantageous because the EM spectrum offers higher bandwidth, and the EM signals can be sampled at higher rates [9], [12]. Moreover, depending on the type of antenna, the approach can be less invasive as the monitoring can be performed from a distance in real time. In fact, EM-based anomaly detection tools have proven to be successful for the detection of extensive [13], [14], or even minimal modifications, say, down to the injection of a few instructions (at the assembly level) [15], [16].

1.1 Problem Statement

Nevertheless, the development of EM-based defenses and the deployment of corresponding real-life solutions remain stagnant due to the limitations of traditional workflows. More specifically, a well-known challenge of these approaches revolves around the requirement for *exhaustive fingerprinting of all normal execution states of the targeted program*. This issue is severely neglected by the research community even though it may be one of the most important practical roadblocks that prevent the deployment of corresponding tools in real-life environments.

1.2 Proposed Solution

To address this issue, an obvious solution is synthetically reconstructing EM signals. As such, a *novel framework for generating synthetic EM signals directly from machine code is presented*. Most importantly, the generated synthetic signals can be used instead of real ones for anomaly detection purposes as part of the model training/fingerprinting stages. In further detail, our approach relies upon first constructing a library of the EM signatures of minimum execution units (i.e., in this case, assembly instructions) that can be used to synthesize the EM footprint of longer sequences of code. The advantage of the proposed approach is that it *completely removes the need for an elaborate and error-prone fingerprinting stage*. The EM signals used for training do not need to be captured, but rather they are *inferred* directly from a model that accepts ASM code as input in an offline step. This fact alone makes the entire process *more scalable*.

However, to a large degree, certain operations of this process are not fully automated and require manual labor. The proposed approach for manually synthesizing EM signals has two major flaws, namely *having to create an exhaustive library* and *requiring to capture of a large number of observations corresponding to the same instruction*. As such, we question if we can achieve similar accuracy while utilizing a smaller library that doesn't contain all possible execution paths. From our experience, some sequences of execution occur more often than others. Additionally, some sequences of executions don't make sense to process, e.g., jmp instruction followed by another jmp. Therefore, we experimentally prove that only a fraction of the possible executions is necessary to cover the majority of programs.

Furthermore, we propose the adoption of a generative model for code-to-EM signal translation instead of manually synthesizing EMs. Generative models are data-driven approaches that can synthesize a near-infinite number of samples close to the normal distribution of real observations for both seen and unseen inputs. Thus, in theory, a generative model can produce realistic EMs from binary code using only the common sequences during training. Reducing the necessary library size to a fraction and further improving the *scalablility of the approach*. As such, the prior framework is modified to use a generative model to automatically synthesize the EMs rather than manually reconstructing them. Specifically, Generative Adversarial Networks (GANs) were chosen due to their success in text-to-speech translation, a task that bears significant similarity to code-to-EM signal synthesis as both convert textual input into a signal waveform.

1.3 Contributions

In summary, the main contributions of this work are (a) *the identification of the requirements and structure of a database* of signal blocks that can be used for the generation of EM sequences, as well as (b) *a methodology for properly synthesizing such sequences of instructions* corresponding to entire execution paths/code-sequences, also a (c) *framework for GANs models to perform code-to-EM signal generation*, furthermore (d) *the creation of a fingerprinting process* that is more scalable than traditional methods, and thus can be applied to various device types and other side-channel classes, and finally (e) *an anomaly detection method* that capitalizes on synthetic signals to distinguish between normal and anomalous program executions.

Chapter 2

Technical Background & Definitions

2.1 Morphology of EM Signals

During a CPU's normal operation, EM signals are produced regularly due to the EM field produced when the magnetic and electrical fields interact. This interaction is caused by the change in the flow of electrical current inside the device's circuitry when executing instructions. Consequently, components of the printed circuit board act as antennas. Thus, unintentionally transmitting EM signals that are highly correlated to the instructions executed by the CPU.

Typically, the corresponding EM signals have (for the most part) static frequencies i.e., the clock of the CPU. Interestingly, different instructions executed by the CPU modulate the amplitude of the EM signal along with a carrier signal that has a base frequency modulated to that of the clock of the monitored CPU [17]. This is due to the clock of the CPU determining the execution time of each instruction. We have hypothesized that different signal amplitude levels correspond to different instructions. However, the amplitudes are not static in nature and have slight variations between each run, even when comparing the EMs produced even when executing the same binary code. Figure 2.1, illustrates this by comparing five EMs from the same program at the exact same execution point. Notice that the amplitude peak varies by roughly 0.03.

In more detail, by placing a near-field probe near the source (CPU) of the EM signal, one can capture the morphology of the signal accurately. As shown in Figure 2.2, EM signals of a CPU during the execution of a program follow a sinusoidal pattern. More specifically, approximately two complete waves starting from a high peak correspond to one execution cycle as illustrated in Figure 2.2. In actuality, one execution cycle starts at the top to the second bottom peak, however for simplicity, we state and show that it is from a high peak to the second next high. For certain CPU architectures, it is common for each instruction when executed to span one to three cycles. The amount of execution cycle given a specific instruction is determined by the device and can be obtained via manuals such as the

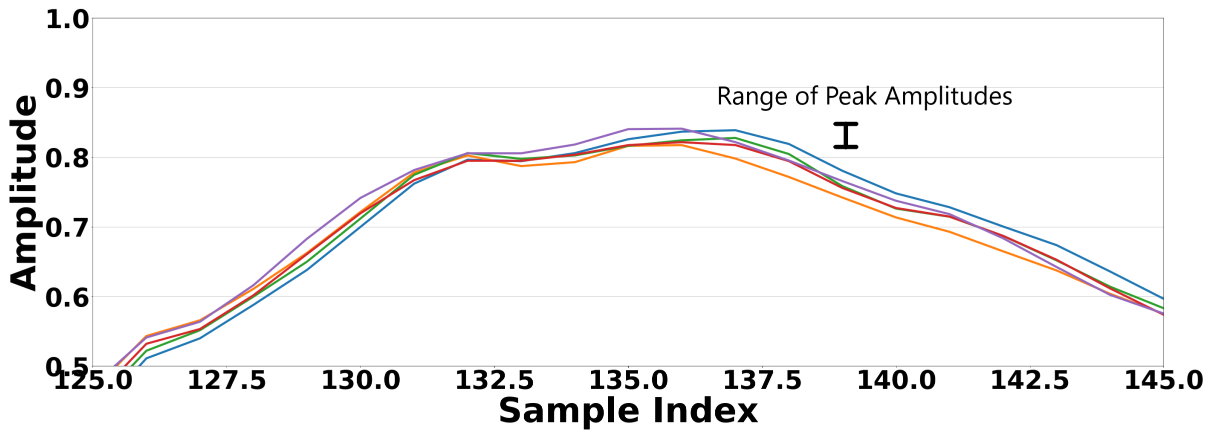


Figure 2.1: Zoomed in comparison between the amplitude peaks from five observations of the same program at the same execution point.

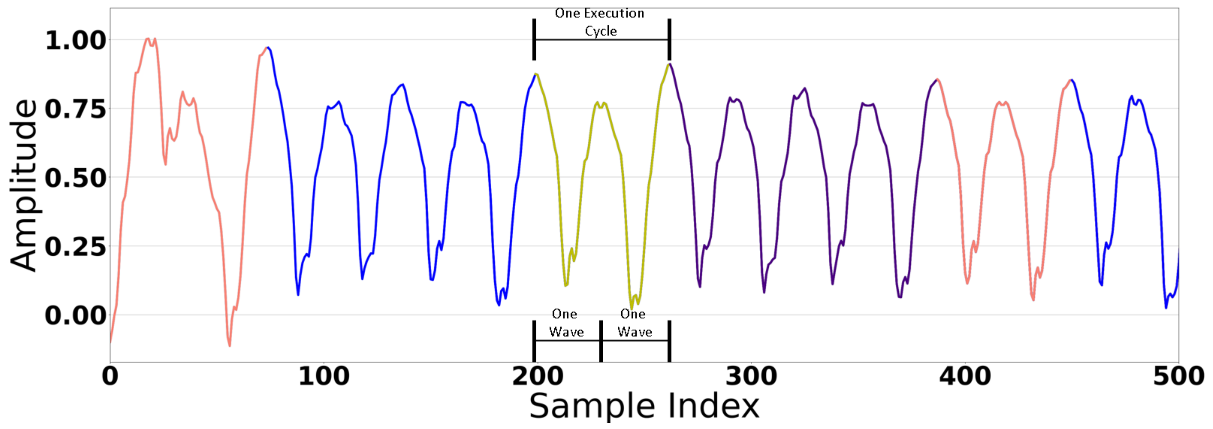


Figure 2.2: Sample of EM signals morphology. Color variation indicates individual instructions.

AVR Instruction Set Manual [18]. While it is true that the current instruction contributes the most to the amplitude, from empirical observations and experimentation, prior executions may also affect the EM signal.

2.1.1 Prior Instruction Influence

The morphological characteristics of the EM signal are mainly determined by the currently executed instruction. However, some residual influence remains from the prior instructions in the execution sequence, slightly affecting the EM signal in terms of amplitude. Moreover, instructions possibly get influenced by other random events that occur at the hardware level or due to parallel processes that are executed at the same time (software), as well as environmental noise. The first two factors have been studied in prior works [15], [16] but they remain open issues and such will be the main topic for future research. However, regarding the impact of previous instructions on subsequent ones, we have made the following observations:

- Although the same instructions may have roughly the same amplitude and general phenotype when observed within the same sequence, they *may appear different* when preceded by different previous instructions or tracked within a totally different sequence of instructions.
- The directly previous instruction I_{i-1} impacts the examined instruction I_i *significantly* but in some cases even previous instructions $\ldots, I_{i-2}$ may impact I_i to a lesser extent.
- Certain instructions impact subsequent instructions less than others.
- Instructions that perform *similar* operations (e.g., mathematical operations *and*, *xor*, etc.) may similarly impact subsequent instructions.

To showcase what was previously discussed we shall provide two concrete examples.

Example 1: The sequence $\ldots, ses, cls, ser, clv, \ldots$ is observed in two different programs. However, for the former, ses is preceded by the lsr instruction as $\ldots,$ **lsr**, $ses, cls, ser, clv, \ldots$ while for the latter the ses instruction is preceded by the sub instruction as in the following sequence $\ldots,$ **sub**, $ses, cls, ser, clv, \ldots$. Nevertheless, both the lsr and sub instructions perform similar (i.e., mathematical) operations. The former performs division and then shift, while the latter performs subtraction. Therefore, the amplitude of the first instruction in that sequence, (i.e., the ses instruction) is only marginally impacted.

Example 2: The sequence $\ldots, rjmp, sbi, \ldots$ is observed in two different programs. In this case, for *Program A* the $rjmp$ is preceded by the clv instruction, while in *Program B* the same instruction is preceded by the clr instruction. The former instruction simply clears the value of a flag while the latter resets the values of all registers. The reader can understand that the two instructions perform drastically different operations thus, it does not come as a surprise that the amplitude of the signal that corresponds to the $rjmp$ instruction looks significantly different in the two programs. A comparison between the signals corresponding to the two programs at the sections of interest is given in Figure 2.3.

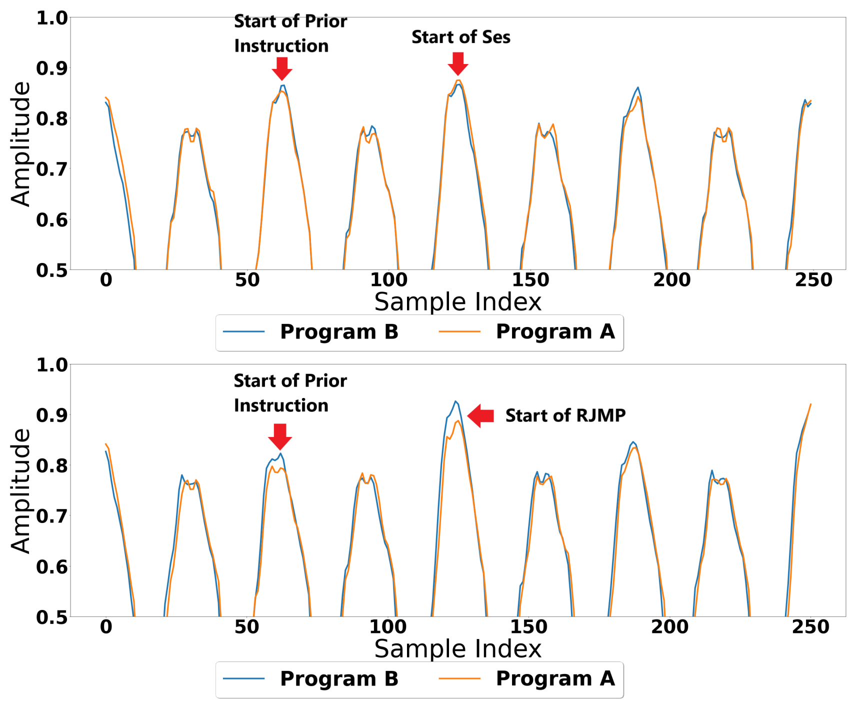


Figure 2.3: Comparison of the amplitude between the corresponding EM output of two different programs. Example 1 is provided in the top figure and example 2 in the bottom.

2.2 Modern Generative Models

In the past decades, many generative models have been developed and applied in different domains such as audio [19], [20], and images [2], [21] generation. Modern generative models that are gaining traction are *Transformers*, *Diffusion*, and *Generative Adversarial Networks*.

2.2.1 Transformers

A *Transformer* [1] network is a type of sequence-to-sequence NN that is based on attention mechanisms forcing the model to heed specific portions of the data. Structurally, *Transformers* contain an encoder and a decoder built by stacks of attention blocks containing two subnetworks: a multi-headed attention layer and a feed-forward network.

Attention is a mapping function that correlates a query Q and a set of key K value V pairs to output, all of which are vectors. The output is estimated as the weighted sum of the values, where each value is assigned a weight that is calculated by a compatible function of the query with respect to the corresponding key.

Transformers make use of multi-headed attention layers. Multi-headed attention jointly attends to information from different subspaces at different positions. This is done by projecting the query Q, keys K, and values V a number of h times with different learned linear projections to the dimensions of d_k, d_k, and d_v respectively. An attention function is performed in parallel on each projected version of Q, K, and V, providing d_v dimensional output values. The d_v dimensional output values are concatenated and projected again to obtain the output. Mathematically, multi-headed attention can be defined as:

$$\begin{gathered} MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O \\ where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \end{gathered} \tag{2.1}$$

Originally, the *Transformer* model was structurally comprised of three attention blocks. The encoder has one that takes the input embedding added with its positional encoding. Positional encoding is required as no recurrence or convolutions are used. The decoder contains two attention blocks. One takes the embedding of the output shifted right and adds it to its positional encoding. The output is passed to a final attention block with the output from the encoder. An abstract architecture of a transformer is

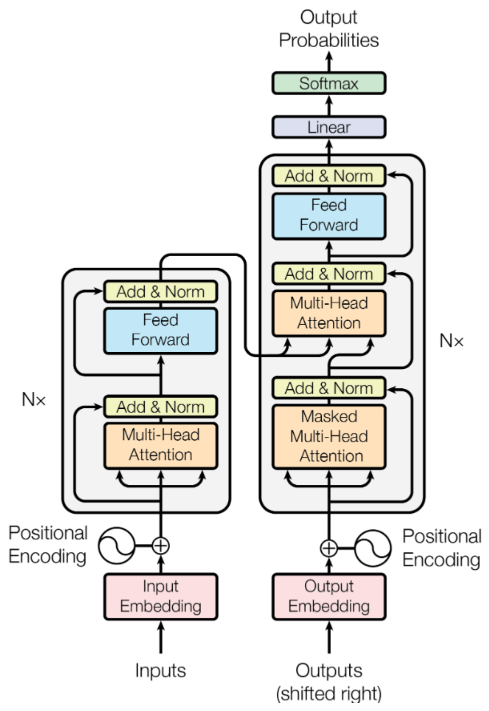


Figure 2.4: Architecture of Transformer models from the original paper [1].

illustrated in Figure 2.4.

The limitation of *Transformer* models is that it requires a fixed length of the input. This is due to having to split the input into a number of segments to feed into the attention layer. Additionally, each query needs access to every key-value pair, which can result in a high memory requirement.

2.2.2 Denoise Diffusion

Denoise Diffusion, or simply *Diffusion*, is a model that takes steps to convert noise into a clear representation of a sample. In the original paper [2], *Diffusion's* training process consists of a forward and reverse process. The forward process takes training samples (x) and goes through a forward process of applying small increments of Gaussian noise in steps s till n number. In the reverse process, a mean function approximator is trained to predict one step lower ($s_i - 1$) by the current step (s_i). This process continues till step s_1. The authors of the original paper state that it is possible to train till s_0, but the variations become more minute from the training set. The process of learning in smaller steps allows for the model to focus on slight alterations within a standard Gaussian distribution. Consequently, the final

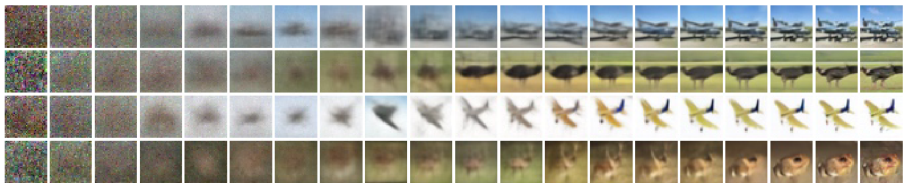


Figure 2.5: Sample of the diffusion process. The left images are high step count samples with high noise values. Lower to no noise steps are to the right. Image is taken from the original paper [2].

results are similar but not exact to the training samples. Sample of the diffusion model process in Figure 2.5.

The main challenge of *Diffusion* models revolves around their requirements in computational resources. Diffusion requires many steps to generate better data. Each of the steps applies another round of Gaussian noise during the forward process and then performs an additional round of training for the new step. For this reason, a larger amount of time and memory are required in comparison to other generative models.

2.2.3 Generative Adversarial Networks

Generative Adversarial Networks (*GANs*) [22] is a type of neural network (NN) model that is widely used for synthetically generating new data. *GANs* has had a variety of uses including text-to-image [4], [5], image-to-image [23], [24], and text-to-speech [25], [26] translation. Furthermore, *GANs* has been used to generate data to train other NN models.

Structurally, *GANs* contains two models, the generator (G) and discriminator (D) that compete against each other. D tries to identify the real from the fake data, while G works to generate fake data that fools D into labeling it as real. Specifically, D tries to maximize and G attempts to minimize the following equation:

$$Loss = \min_{G} \max_{D} E_x[\log D(x)] + E_z[\log 1 - D(G(z))] \tag{2.2}$$

where $D(x)$ is the discriminator's estimate of the probability that real data x is real, $G(z)$ is the generator's output when given noise z, $D(G(z))$ is the discriminator's estimate of the probability that fake instance is real, and E_x and E_z is the expected value over all real instances and overall fake instances respectively. The min-max game continues till the generative model is able to consistently produce realistic fake data or in other words when the Loss is at its lowest.

The main problems with *GANs* are that the model can suffer from *vanishing gradients*, *model collapse*, and failure to converge [27], [28]. Some approaches have been proposed to address GANs main issues such as *Wassterstein GANs* and Gradient Penalty [29].

Some GANs models of note are provided in the following sections.

2.2.3.1 Wasserstein GANs

Wasserstein GANs utilizes the Wasserstein distance function to change the discriminator to a critic, giving a value to each instance rather than classifying between real and fake. Therefore, $D(s,t) \longrightarrow \{0, \infty\}$ where the higher the value, the more likely the instance is real. This allows the generator and discriminator to compete in a min-max situation more effectively as the loss provides more feedback when training. Overall, the minmax_loss is reworked to the following equation:

$$W\_Loss = \min_{G} \max_{D} \mathop{\mathbb{E}}_{x \sim \mathbb{P}_r} [D(x)] - \mathop{\mathbb{E}}_{G(z) \sim \mathbb{P}_g} [D(G(z))] \tag{2.3}$$

Where $\mathbb{P}_r$ is the data distribution of real data, $\mathbb{P}_g$ is the model distribution defined by $G(z)$, $D(x)$ is the discriminator's output for a real instance, $G(z)$ is the generator's output when given noise z and $D(G(z))$ is the discriminator's output for a fake instance. The discriminator attempts to maximize the W_Loss by increasing the difference between the output of the real and the fake instances. Consequently, the generator can only influence the right side of the algorithm, $D(G(z))$, and tries to maximize this output to minimize the W_Loss.

2.2.4 ConditionalGANs

ConditionalGANs [30] extends traditional GANs into a conditional based model. This is done by passing extra information y to the generator and discriminator. y can be any auxiliary information, e.g., class labels or data from other modalities.

In *ConditionalGANs*, conditioning is performed by feeding y to both the discriminator and generator as an additional input layer. In the generator's case, prior input noise z and y are combined in joint hidden representation. It should be noted that the adversarial training framework allows for considerable flexibility on how the hidden representation is composed, however, the organization must remain the same throughout, e.g., y always be appended to z at the end. In the discriminator's case, x and y are combined and presented as inputs to the discriminative function.

Given the extra information of y, the loss function of the traditional GANs model is updated to:

$$Loss = \min_{G} \max_{D} E_x[\log D(x\|y)] + E_z[\log 1 - D(G(z\|y))] \tag{2.4}$$

2.2.4.1 CycleGANs

CycleGANs is known to be useful for unpaired translation. Furthermore, with *CycleGANs* two generators are created, each converting one to another datatype (e.g., one generator for code-to-EM and one for EM-to-Code). This is of particular interest as having an EM-to-Code in addition to a Code-to-EM translator could benefit the EM-based side-channel analysis community.

CycleGANs attempts to learn how to generate synthetic data by performing two transformations. These two transformations convert one instance into another and then back (e.g., code-to-signal and back signal-to-code). Consequently, CycleGANs have two generators and two discriminators. Additionally, an additional loss value is given for how well the cycle is able to reproduce the original input. Furthermore, during training, the model also attempts to learn via the opposite direction (i.e., signal to code and back to signal).

Structurally, *CycleGANs* contain two generator mapping functions $G : X \longrightarrow Y$ and $F : Y \longrightarrow X$, and two associated adversarial discriminators D_y and D_x. D_y encourages G to translate X into outputs indistinguishable from domain Y, and vice versa for D_x and F. Furthermore, an additional metric namely, *cycle consistency loss* is calculated for both the forward and backward processes. Forward Cycle consistency loss regularizes the mappings by capturing the intuition that $x \longrightarrow G(x) \longrightarrow F(G(x)) \approx x$. Backward Cycle consistency loss is similar except in the opposite direction (e.g., $y \longrightarrow F(y) \longrightarrow G(F(y)) \approx y$). Structure of *CycleGANs* is illustrated in Figure 2.6.

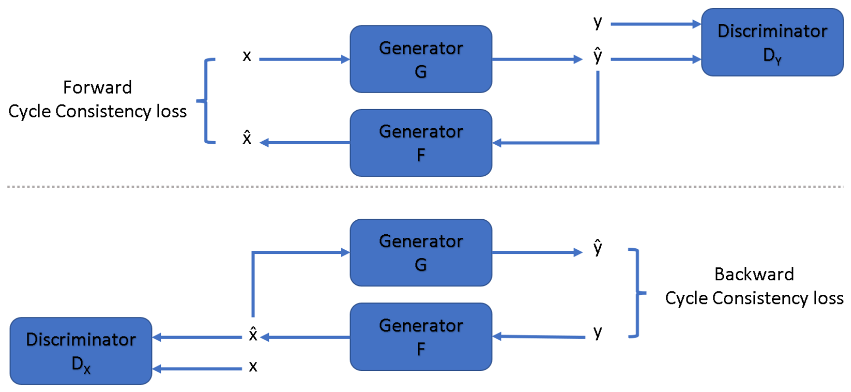


Figure 2.6: Basic structure of CycleGANs.

Chapter 3

Problem Statement

Typically, software supporting embedded devices designed to control critical processes is considered of *low complexity* when compared to the analogous software running on servers and desktop systems. Indeed, corresponding workflows involve cycles of sensing, processing, and then actuation, all executed in a loop fashion. However, realistically, even the simplest examples of this family of software may be comprised of hundreds if not thousands of execution paths spawned by conditional branching instructions. When the objective is to *model the characteristics of normalcy* then *all of these execution paths must be observed, analyzed and fingerprinted*. Particularly, EM-based fingerprinting is mainly a human-expert-centric process revolving around tasks such as the correct positioning of probes, deciding the optimal recording parameters like the sampling rate, and synchronizing EM signals, among others. This, in turn, renders EM fingerprinting an extremely time-consuming, error-prone, and costly process.

This challenge is further amplified by two real-life restrictions. Firstly, execution branches may exist in a program that *are meant to be rarely followed*. Even techniques such as the forceful execution [31] of specified branches might not be an option as such paths may be associated with critical failures (e.g., shutting down of critical systems as part of the emergency sequence). For this reason, these branches are likely to be left out of the fingerprinting phase. In this case, the resulting models will yield wrong predictions for these normal-yet-rare-situations.

Secondly, *embedded devices occasionally receive firmware/software updates*. These modifications in the executable generate the requirement for fingerprinting the behavior of the device from scratch. These challenges are illustrated in Figure 3.1.

For this reason, it is important to minimize any manual processes involved in the fingerprinting of EM signals. Towards this end, the chapters below will present a comprehensive framework for synthetically generating EM signals directly from ASM code. While this undertaking has certain challenges, it can significantly reduce the cost and manual labor involved.

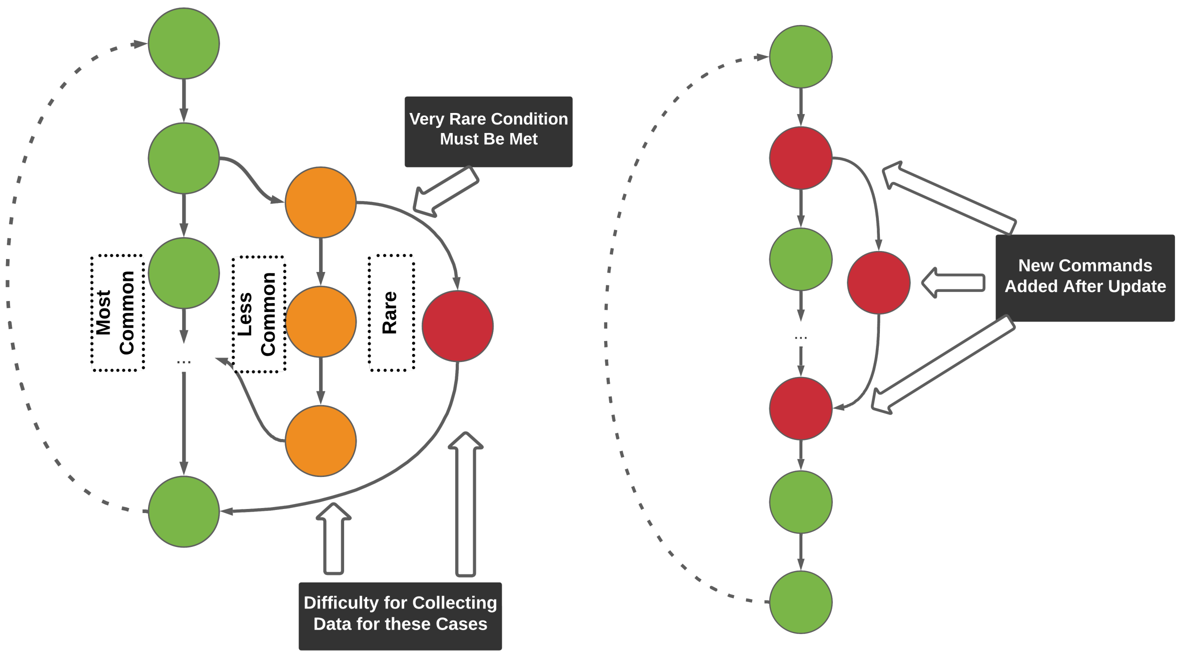

Very Rare Condition Must Be Met
Most Common
Less Common
Rare
...
Difficulty for Collecting Data for these Cases
...
New Commands Added After Update

Figure 3.1: Scenarios where fingerprinting which is based on synthetic data could be valuable. Rare execution paths of programs are depicted in red and orange (left). New states/commands introduced after software updates are depicted in red (right).

Chapter 4

A Framework for Synthesizing EM Signals: An Expert-guided Approach

This Chapter provides a detailed description of a novel approach and evaluation for manually generating EM signals for anomaly detection purposes. More specifically, a framework [32] for conducting the fingerprinting stage completely offline through the use of synthetically generated EM signals is proposed. The effectiveness of the proposed approach is evaluated for the detection of code injection attacks against the software of embedded devices, accounting for multiple programs or execution paths. Later on, elements of this methodology are extended to create a totally automated data-driven approach.

4.1 Proposed Framework

The purpose of the proposed framework is to conduct anomaly detection with high accuracy using synthetically generated versions of the EM signals that correspond to the *normal execution branches only*. A high-level overview of the proposed framework is given in Figure 4.1. In summary, the main steps involved in the process are as follows. During an offline step, a database of instructions-to-signal pairs is created (this is denoted as step ❶ in Figure 4.1). Next, synthetic signals are created using the database of EM signals and the target binary (step ❷ in Figure 4.1). Then, these sequences are used to train the baseline during the fingerprinting phase (step ❸). After this phase, the target device is expected to be deployed on the field. At this point, the anomaly detection phase takes place (step ❹). Afterward, real EM signals emanated by the device are captured once again, this time to be evaluated for anomalies. This process also capitalizes on the baseline that was already created during the previous step. Under the hood, the process involves the execution of machine learning algorithms that judge whether the new signal bears significant morphological similarities with the synthetic ones that were used to construct the baseline. Hereunder, we shall analyze the basic steps of the process in further detail.

This framework assumes that a reliable mechanism for capturing EM signals from the elements of

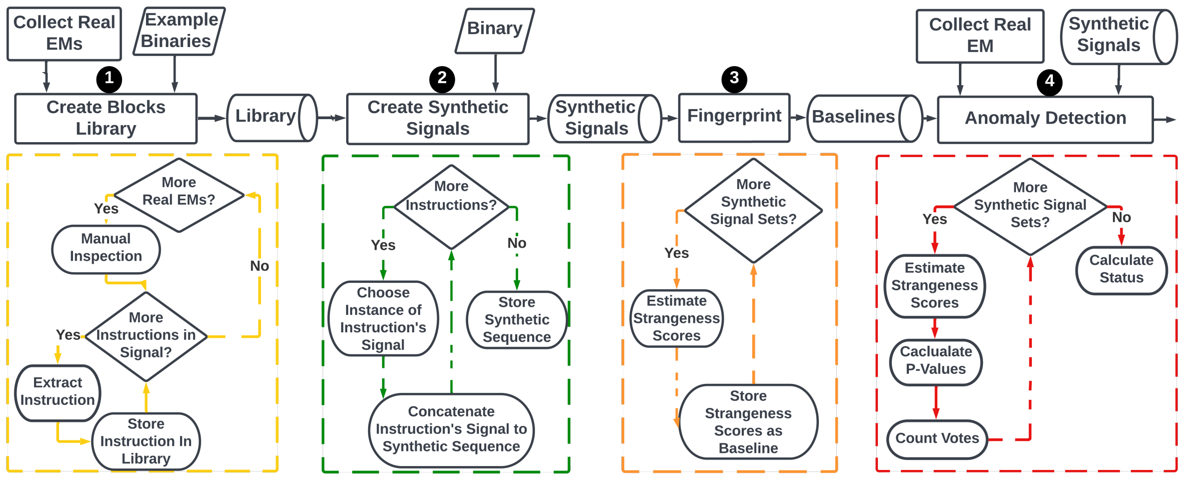


Figure 4.1: Workflow of the proposed framework.

devices (e.g., CPU) is available. Today, this can be achieved by solely relying on COTS components. Such an assembly of components typically consists of (a) a near-field antenna for gathering the raw signals, (b) an amplifier for increasing the strength of the captured signal, (c) an oscilloscope for digitizing the collected analog signals, and finally (d) hard disks for storing the captured signals in their discrete form. In this framework, the process of capturing signals is performed in two separate stages i.e., during the construction of the library of basic building blocks (step ❶), a process that is completed offline, and during run-time for actively monitoring the health status of a target device (step ❹). Typically, the signals are collected by placing the antenna in close proximity to the CPU. However, in more advanced settings signals can be collected from multiple onboard components (e.g., the network module), and create more sophisticated correlations regarding the behavior of the device. Particularly for the latter case, an extra step of pre-processing that may involve noise elimination procedures may be included as part of steps ❸ and/or ❹ In this work, we have omitted such processes for purposes of simplicity.

4.1.1 Building a Library of Signal Blocks

A library of *basic building blocks of signals* is assumed to have been created a priori in an offline step. This library should be available during the *fingerprinting* of any program, or more accurately any subsequence of any execution path inside a program. Theoretically, the term *basic building block* corresponds to the EM signature of patterns frequently observed among multiple different signals. While there are many possibilities, in our case, we relied on assembly-level instructions, e.g., *and, nop*, etc. which also constitutes the minimum building block of all programs.

Experimentally, we have identified that the main challenge with this approach is that one instruction in a sequence influences the shape and amplitude characteristics of the EM wave and the subsequent instructions. Typically, the direct next instruction is influenced only. However, depending on the type of instruction (e.g., instructions involved in I/O operations) multiple subsequent instructions may also be affected but to a lesser extent. In this work, we have assumed that only one instruction gets affected for reasons of simplicity, but further investigation is required. Therefore, the structure of the database can be defined as:

$$\mathcal{M} = \begin{bmatrix} (I_0 \mid I_1) & S_1^{(I_0|I_1)} \\ (I_1 \mid I_2) & S_2^{(I_1|I_2)} \\ \cdots & \cdots \\ (I_{n-1} \mid I_n) & S_n^{(I_{n-1}|I_n)} \end{bmatrix} \tag{4.1}$$

where the $(I_{n-1} \mid I_n)$ operation indicates that instruction I_n has been observed after I_{n-1}. The reader should notice that an entry $S^{(I_{n-1}|I_n)}$ is comprised of the same number of samples as S^{I_n} would.

Let us examine the requirements for the construction of such a database through a simple example. The x86 architecture supports 981 [33] unique instructions while a simpler CPU architecture like AVR supports unique 123 instructions [18]. Let us focus on the AVR architecture since it is widely deployed in embedded systems. Let us assume that 1000 examples of each instruction are captured then the original size of the database is estimated to have $1000 * 123 = 123K$ entries. In the lieu of the described restriction, the database needs to have a total of $1,000 * 123^2 \approx 15M$ entries which is approximately two orders of magnitude larger than the original estimation. It is obvious, that the process of creating a database of all possible instructions is time-consuming. Regardless, this needs to be conducted only once. One can argue that once constructed, a database for a specific architecture can be open-sourced and made publicly available. Moreover, in practice, certain instructions are never observed together, while there are certain combinations of instructions that are frequently executed together. Thus, the requirements of constructing such a database are not prohibitive.

4.1.2 Manually Generating Synthetic EM Signals

The process of manually generating synthetic signals for anomaly detection is comprised of the following distinct steps:

- based on the sequence of instructions included in the binary, identify the next instruction that will be executed
- fetch a random EM sample that is associated with this instruction from the library
- append the EM at the end of a collective synthetic signal

The above steps are repeated until no more instructions are contained in the target sequence.

Algorithm 1 Fingerprinting Phase

```
 1: function CALC_STRANGENESS:(benign dataset $X$, set of query signals $Q$, number of neighbors $\kappa$):
 2:     $Strangeness\_Scores = []$
 3:     for $\forall q \in Q$ do
 4:         $D = []$
 5:         for $\forall x \in X$ do
 6:             $D \leftarrow distance(x, q)$
 7:         end for
 8:         $Nearest\_Neighbors = get\_min(D, \kappa)$
 9:         $Strangeness\_Scores \leftarrow Sum(Nearest\_Neighbors)$
10:     end for
        return $Strangeness\_Scores$
11: end function
12: function FINGERPRINT:(benign dataset $X_s$, number of neighbors $\kappa$, number of benign execution paths $s$):
13:     $Baselines = []$
14:     for $\forall X \in X_s$ do
15:         $Baselines \leftarrow calc\_strangeness(X, X, \kappa)$
16:     end for
        return $Baselines$
17: end function
```

4.1.3 Fingerprinting Phase

In this work, the discovery of malicious EM signals was approached as a *semi-supervised anomaly detection* problem as opposed to a *supervised classification one*. The reason for this decision is that nearly infinite alterations to a benign program can be performed by an attacker. This makes collecting instances of all possible known and unknown malicious versions of a program unrealistic. However, since the normal modes of operation of a device are finite, it is valid to assume that the corresponding EM signals can be collected, or in the context of this work, be synthetically generated. Therefore, we relied on and extended an existing semi-supervised anomaly detection method [34]. This method is based on the principles of *transduction* and *hypothesis testing*. Transduction is a technique of placing an example in a set of known normal observations and understanding whether that sample is a good fit in the set. From the perspective of our experiment, the terms *example* and *observations* refer to EM signals that correspond to a repetitive operation e.g., a loop.

The method calculates a distribution of normalcy, namely, a baseline, between all the known benign cases corresponding to the same operational mode (i.e., an entire or parts of the same execution path). Realistically, a program can have several execution paths, with each execution path corresponding to a different aspect of normal operation. This, in turn, creates a unique EM signal.

In further detail, during this phase, a set of benign signals, X, is provided for each execution path. X must contain a significantly large number of EM signals because as explained in previous sections, observations of the same path can deviate due to random phenomena occurring during the capture. In order to calculate the distribution, the *strangeness* (similarity) score of each sample point x with the rest in X must be calculated. Any algorithm that calculates the similarity (e.g., euclidean distance) can be used to estimate the *strangeness*. These include rudimentary approaches such as the mean of distances, or more sophisticated metrics like the Local Outlier Factor [35] (which internally relies on Euclidean distance). The processes involved in the fingerprinting phase are given in Algorithm 1. We relied on the sum of the κ-nearest (most similar) neighbors (signals) and the Euclidean distance metric. The outcome of this process is one (or multiple) lists that contain the similarity scores, referred to as $Strangeness\_Scores$, (lines 5-9) in the algorithm. The $Strangeness\_Scores$ reflect the distribution of normalcy or simply put a *baseline*, (lines 13-16). This process is repeated for all possible execution paths.

Algorithm 2 Anomaly detection Phase

```
1:  function DETECT:(sets of benign signals $X_s$, strangeness baselines $B_s$, signal for evaluation $q$, number
    of neighbors $\kappa$, threshold $\tau$, number of benign execution paths $s$ ):
2:      $Votes = []$
3:      for $\forall X \in X_s$ do
4:          $size = length(B_i)$
5:          $score_q = calc\_strangeness(X, q, \kappa)$
6:          $Sorted\_Baseline_i = sort(B_i, ascending)$
7:          $index = 0$
8:          for $\forall score_x \in Sorted\_Baseline_i$ do
9:              if $score_q < score_x$ then
10:                 $index = index + 1$
11:             end if
12:         end for
13:         $Vote = anomalous$
14:         $p\_value \leftarrow \frac{1+size-index}{1+size}$
15:         if $p\_value > \tau$ then
16:             $Vote = normal$
17:         end if
18:         $Votes \leftarrow Vote$
19:     end for
20:     $status = anomalous$
21:     for $\forall vote \in Votes_q$ do
22:         if $vote = normal$ then
23:             $status = normal$
24:         end if
25:     end for
         return $status$
26: end function
```

4.1.4 Anomaly Detection Phase

The deployment phase assumes that the baselines, B_s, have already been produced successfully during the fingerprinting phase. Furthermore, the original sets of benign signals used to create the baselines, X_s, and the number of benign execution paths, s, are provided. Additionally, a signal for evaluation, q, is available. Finally, user-provided parameters that correspond to the number of neighbors (κ) and the threshold used to separate the normal from abnormal (τ) signals are given. The overall process is provided in Algorithm 2.

During this process, a benign set of each execution path, X, is obtained from X_s. Then the strangeness of the new observation, $score_q$, is evaluated by comparing q to X using the same algorithm implemented in the fingerprinting phase, (line 5). Afterward, $score_q$ is compared against the respective baseline, B_i, that was created from X in the fingerprinting phase. The comparison process is executed using transduction, creating a p_value for q, (lines 8-13). If the p_value is above the threshold τ, then q is considered within the norm of the execution path, and a $vote$ is saved as normal. Otherwise, the $vote$ is saved as abnormal, (lines 14-19). This process is repeated for all execution paths in X_s to check if q falls within the norm of *any* of the benign execution paths. Under normal conditions, benign signals are expected to be considered normal for one execution path. As such, only one $vote$ for the unknown signal q being normal is required to flag it as benign. If no $vote$ is given as normal, then q is flagged as anomalous, (lines 21-26).

The voting mechanism was an extension to the original algorithm implemented to account for the certainty of a program being comprised of numerous paths. A comprehensive fingerprinting of a target program must consider, as *normal*, all possible paths inside that program.

Chapter 5

Manual Synthesis Experimental Setup & Data Gathering

To evaluate the proposed framework, we have created an experimental setup consisting exclusively of low-cost, off-the-shelve components. A simple control process emulating a *tank filling system* is used as the software to be protected against malicious modifications. Two alternative versions of that software are compiled, each one corresponding to the version of the control logic after the malicious injection of a single instruction. During the fingerprinting stage, EM signals are obtained from the benign version of the software. These signals are used to create the baseline and train the detection engine. At the deployment stage, we collected signals while the device was operating under normal as well as anomalous states (i.e., after the injection). The predictive accuracy of our system is evaluated. Details regarding the process described above are given in this section.

5.1 Experimental Setup

An illustration of our setup, along with the relevant placement of the mentioned components, is given in Figure 5.1. For all considered experiments, the subject device was the Arduino Mega ❶. While the framework assumes any type of near-field antenna placed at a distance of a few cm away from the device, in our experiments, we used a near-field probe placed directly on top of the CPU; namely, an EMRSS RF Explorer H-Loop ❷. This was done to obtain EM readings that are virtually noise-free.

Since such signals are emitted involuntarily using no dedicated antennas they are typical of extremely low amplitude. Therefore, each signal captured ❸ was first amplified using a Beehive 150A EMC probe amplifier ❺. The captured signals were saved in a digital format with the use of a PicoScope 3403D oscilloscope ❻ connected to a laptop ❼. For all experiments, the sampling rate was set to 250MS/sec, i.e., a sampling interval of 4ns.

Notice that this sampling rate is almost 15 times more than the CPU clock of the chosen platform.

Using an external signal from a specific I/O pin (❹ in Figure 5.1), we were able to identify the beginning and end of the program's execution loop by toggling between high/low values on the I/O pin. This was done to maintain a strong synchronization among all the signals.

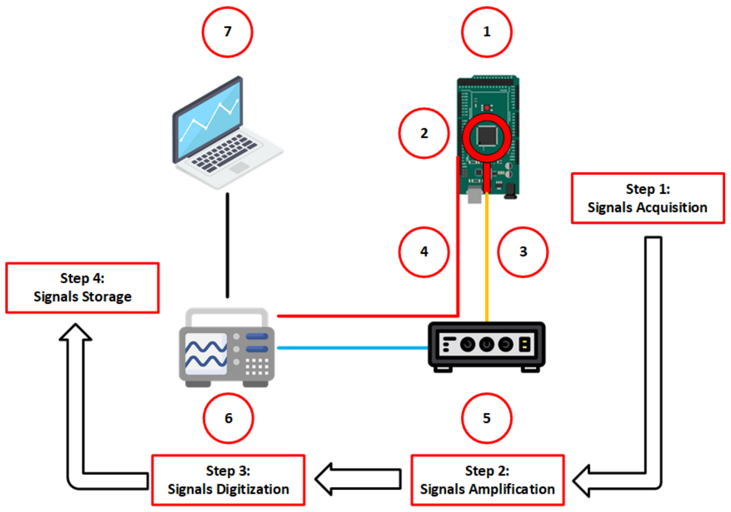


Figure 5.1: The proposed framework along with the major components used in our experimental setup.

5.2 Dataset

For evaluating purposes, we considered the following scenario: a benign program with just one execution path is already installed in the target platform. The original software is comprised of just *17 instructions being executed inside a loop*. At some point, there was a need to modify the original program. In the update *several instructions were substituted, a new one was added and one was removed from the original sequence*. The task is to synthetically generate EM signals of the modified version of the program directly from the assembly (ASM) code so that we do not have to engage in the data gathering process from scratch. The original (*Program A*) and the updated version of the program (*Program B*) are given in Listings 5.1 and 5.2.

```
setup:
  sbi ddrb, 6  ; set pb6 as output (our sync artifact)
loop:
  sbi   pinb, 6
  ; simulates a decision
  clr   r20
  ldi   r20,  1
  ; Will go to the first_label
  ldi r22, 1
  cp  r20, r22
  breq  first_label
  rjmp loop

first_label:
  clr r2
  ldi  r23, 1
  mov  r1, r23
  cp r1,  r2
  lsl  r1 ; multiply r1 by 2 with logical shift left
  lsr  r2 ; divide r2 by 2 with logical shift right
  ses ; Set signed flag
  cls ; Clear signed flag
  sev ; Set Overflow Flag
  clv ; Clear Overflow Flag

  rjmp loop
```

Listing 5.1: Assembly code of *Program A*.

Next, we assumed that a malicious entity performs a modification (i.e., code injection) to our program. To illustrate the occurrence of such an attack, we developed two contaminated versions of the updated program (*Program B*), each with differing amounts of injected code. The first contaminated version assumes that four malicious instructions were injected, while in the second case we have the injection of only two instructions. Consequently, the second version will be harder to detect due to the shorter length of the foreign code. The point of injection for both contaminated versions is in the middle of the sequence of the ASM instructions. The two malicious programs (*easy and hard*) are given in Listings 5.3 and 5.4.

```
setup:
  sbi ddrb, 6  ; set pb6 as output (our sync artifact)
loop:
  sbi   pinb, 6
  ; simulates a decision
  clr   r20
  ldi   r20,  1
  ; Will go to the first_label
  ldi r22, 1
  cp  r20, r22
  breq  first_label
  rjmp loop

first_label:
  ldi r23, 0
  and  r2, r3 ; Bitwise AND (result in stored r2)
  add  r1, r2 ; Add r2 to r1 (r1=r1+r2)
  eor  r2, r3 ; Bitwise exclusive or between r2 and r3
  sub  r1, r2 ; Subtract r2 from r1
  ses ; Set signed flag
  cls ; Clear signed flag
  sev ; Set Overflow Flag
  clv ; Clear Overflow Flag
  clr r1

  rjmp loop
```

Listing 5.2: Assembly code of *Program B*. Different instructions from *Program A* are highlighted in red.

```
...
; up to here, same as "Program B"

  add  r1, r2 ; Add r2 to r1 (r1=
    r1+r2)
  ;====== 4 injected instructions
    ========
  asr r3 ; r3=r3/2
  com r3 ; Take one's complement
    of r3
  adc r3, r2
  sbc r3, r2
  ;==============
  eor  r2, r3 ; Bitwise exclusive
    or between r2 and r3

 ; the rest are same as "Program B
    "
...
```

Listing 5.3: Assembly code of *Malicious B Easy*.

```
...
; up to here, same as "Program B"

  add  r1, r2 ; Add r2 to r1 (r1=
    r1+r2)
  ;====== 2 injected instructions
    ========
  asr r3 ; r3=r3/2
  com r3 ; Take one's complement
    of r3
  ;
  ;
  ;==============
  eor  r2, r3 ; Bitwise exclusive
    or between r2 and r3

 ; the rest are same as "Program B
    "
...
```

Listing 5.4: Assembly code of *Malicious B Hard*.

Chapter 6

Evaluation of Proposed Framework

In this chapter we shall describe a series of experiments performed towards evaluating the efficiency of the proposed method.

As a *first experiment*, we relied upon *real observations of signals only*. More specifically, the real signals that correspond to the normal programs (before and after the update) were used to train the baseline. Thus, at this phase, no malicious observations were used. During the testing phase, examples of both normal and malicious cases were utilized. In fact, we performed two rounds of evaluation, for the first round the examples of malicious signals were drawn from the pool of signals that correspond to an *easier-to-distinguish* malicious case. For the second round, the malicious signals were chosen from the pool of *harder-to-detect* anomalies. The goal of this experiment was to estimate the accuracy of the anomaly detection method in the ideal situation where real signals are available. The results will be used as a baseline.

A *second experiment* was performed in a similar fashion, except *real examples for the original program* and *synthetic data for the modified version* were used to train the baseline. Obviously, synthetic EM signals are not expected to be identical to real ones. However, for the proposed task the goal is to *approximate* the predictive performance achieved when utilizing the real signals. Therefore, this experiment aims to quantify the expected penalty in terms of predictive accuracy due to reliance on synthetic data.

The two experiments were evaluated using the 10-fold cross-validation method. For experiment one, for each fold, the training set was comprised of 450 examples of *Program A*, and 450 of *Program B*. Furthermore, to evaluate with a balanced testing set, the testing dataset considered for each fold only 50 observations of each benign (original and modified) case along with 100 anomalous examples. For the second round, the number of signals of each different type of program used for the training/testing set was the same except that the training set contained synthetic EMs for *Program B*.

Preprocessing: Before the training and testing phases, feature engineering was performed. First, every signal was reduced to the size of the benign execution paths. The reader should keep in mind that the size of the benign sequence is known in advance. As such, we assume that every signal that is being evaluated should only be the size of the benign case if it is truly benign. The reader should recall that each instruction is amplitude modulated. Therefore, the main indicator for identifying various instructions is the difference in the amplitude of the signal at certain time frames (i.e., cycles). In fact, one challenge that we observed in raw signals is that occasionally there are minor clock drifts. By maintaining only the peaks, we effectively deal with this issue without the need for relying on computationally heavy techniques such as dynamic time warping (DTW).

Considered Parameters: After performing a grid search we identified the optimal *nearest neighbors* parameter to be 10. Moreover, the anomaly detection process made use of thresholds τ ranging from zero to one, with a step of 0.001.

Evaluation Metrics: Given the confusion matrix results, we obtained the area under the curve (AUC) of the receiver operating characteristic (ROC), and among the thresholds tested the one that gives the best accuracy (ACC) and F1 score for each fold was considered. ROC is a common metric used for evaluating the efficiency of anomaly detection systems. It graphs the true-positive rate (TPR) vs. the false-positive rate (FPR) under various thresholds. The formulas for calculating the TPR and the FPR respectively are:

$$TPR = \frac{TP}{TP + FN} \tag{6.1}$$

$$FPR = \frac{FP}{FP + FN} \tag{6.2}$$

Where TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives. The resulting graph usually creates a curve, and the AUC is a common metric for comparing ROCs. The ACC and the F1 scores are is computed as follows:

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.3}$$

where TN is the number of true negatives.

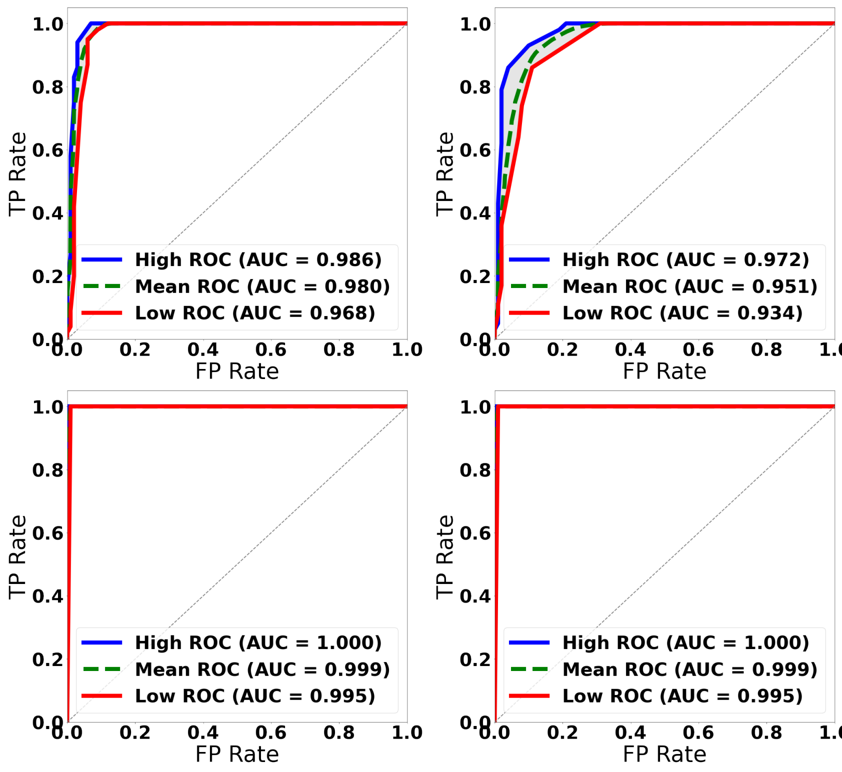


Figure 6.1: ROC graphs for the anomaly detection experiment. In the upper row, the results were obtained when using the *synthetic* signals for training. In the lower row, the results obtained when using only *real* signals for training (ideal case). The drop in the AUC score observed is only 1.3% for the injection of 4 instructions (an easy case) and 4.2% for the injection of 2 instructions (a hard case).

$$F1 = 2 * \frac{PPV * TPR}{PPV + TPR} \tag{6.4}$$

where in turn the precision (PPV) is defined as:

$$PPV = \frac{TP}{TP + TN} \tag{6.5}$$

The final reported metrics are average among all folds. The max, minimum, and average ROC curves observed across all folds are given in Figure 6.1.

Results: The results achieved for each of the experiments can be seen in Table 6.1. Using *synthetic* data gives above 90% AUC score for all considered metrics. More specifically, the AUC score achieved when using the easy malicious case is 98%, and 95.1% when using the hard version. The AUC score achieved for the same tests when *real* signals were used is 99.3% for both the easy and hard cases. In other words, the use of synthetic signals had a negative impact on the predictive AUC but it was relatively low i.e., 1.3% and 4.2% respectively. The reader should recall that despite the malicious programs being labeled *easy* and *hard* both cases correspond to exceptionally minimal injections and in reality, the attacker probably would try to inject much larger lengths of instructions.

In terms of ACC and F1 score, the use of synthetic signals achieved 90.1% and 90.6% respectively when using the hard malicious case. Furthermore, these metrics reach 95.4% for the ACC and 95.5% for the F1 score when evaluating against the easy version. When the same tests are performed using the real signals, the ACC and F1 is near perfect, that is 99.9% and 99.5% for the hard case and 99.9% and 99.8% for the easy version. Overall the difference in the use of synthetic signals was 4.5% to 9.8% for the ACC and 4.3% to 8.9% for the F1.

Table 6.1: Anomaly detection results using Synthetic EMs.

Training	**Test-Normal**	**Test-Anomaly**	**Scores (Avg.)**		
			AUC	**ACC**	**F1**
Real (Program A) and Synthetic (Synthetic B)	Real (Program A and Program B)	Malicious B (Easy)	0.980	0.954	0.955
		Malicious B (Hard)	0.951	0.901	0.906
Only Real (Program A and Program B)	Real (Program A and Program B)	Malicious B (Easy)	0.993	0.999	0.998
		Malicious B (Hard)	0.993	0.999	0.995

Conclusion: *While using real captured EM samples may provide near-perfect detection of even minimal code injections, synthetic fingerprinting can still effectively train models to distinguish between benign and anomalous cases with high accuracy. For example, the penalty in terms of AUC score is -1.3% for the detection of only four malicious instructions.*

Chapter 7

Transitioning to a Data-Driven Approach for EM Signal Synthesis

There are two main challenges regarding the process of manually synthesizing EMs for anomaly detection. Firstly, *the number of instructions contained in the library's database must be exhaustive*. Essentially, requiring I^R number of instructions to be captured, where I is the number of executable instructions for a target device and R is the number of prior instructions to account for. Secondly, *a large number of signals corresponding to the same instruction I_i must be captured because the phenotype of signals corresponding to the same sequence of instructions is not static*. This is further discussed in Section 2.1.

7.1 Considerations Regarding the Library of Reusable Basic Program Blocks

Let us suppose that the previous observations regarding the influence of prior instructions (Section 2.1) are not true. Then, it would be possible to construct a set of programs P comprised of the instruction to be fingerprinted I_i surrounded by sequences of nop instructions as:

$$P_i = \{\ldots, nop_{n-2}, nop_{n-1}, nop_n, I_i, nop_{n+1}, nop_{n+2}, \ldots\} \tag{7.1}$$

Notice nop instructions are considered neutral as they do not perform any function but simply consume a cycle thus, they are an ideal choice for this fingerprinting task. For the considered CPU architecture this would amount to creating 123 unique programs i.e., the same as the number of unique instructions. At a subsequent step, the instruction I_i would be stripped from surrounding the nop and entered in a database. In the future, for any given sequence of instructions, it would be possible to consult this database and retrieve the corresponding EM sequences. In this scenario, the entire workflow is deemed trivial, and the task of EM synthesis is reduced merely to a simple mapping of signal-to-instruction.

However, as explained in Section 2.1, in reality, the task is not trivial because each instruction I_i in a sequence is influenced by the previous instruction I_{i-1}. Thus, the database of reusable components must be constructed by considering at least two instructions. The situation becomes more challenging because in turn instruction I_{i-1} is expected to have been altered by I_{i-2}. Thus, when creating the database the previously examined instruction must be considered and explicitly specified.

To put things into perspective, for our considered CPU architecture the number of possible instruction combinations is 123x123 which is more than two orders of magnitude larger than the naïve case. Alternative CPU architectures may support a significantly higher number of instructions. It is obvious that this approach does not scale. Additionally, if we want to account for more prior instructions, this becomes even more daunting. For example, if we consider P prior instructions and have N number of executable instructions on the target device, then the total possible execution sequences to account for can be calculated as:

$$Possible\ Number\ of\ Sequences = N^P \tag{7.2}$$

Paired with the fact that multiple examples of each sequence's EM signal are necessary and that trained professionals must collect the samples, this task becomes very costly.

7.2 Reasoning Behind a Data-Driven Approach

Fortunately, it may not be necessary to capture all possible sequences. From our observations, some sequences are seen more frequently than others. At the same time, some sequences cannot exist because their corresponding operations do not make sense. Furthermore, it may be possible for deep learning models to learn how to predict the EM signal of a sequence, even though it was not specifically trained on it. As such, an evaluation is performed to determine how many instruction sequences must be provided to train a model so that it can achieve a level of coverage for an arbitrary number of alternative target programs.

7.2.1 Coverage of Programs using the Most Common Sequences

To validate that only a fraction of the total possible sequences is necessary, a study was performed to find how much the most common sequences would cover on the majority of programs. To conduct this study, the Arduino Mega 2560 platform was considered. The ASM code of the most popular Arduino Mega 2560 projects was retrieved from GitHub. Next, various amounts of the most common sequences of instructions from the manufacturer's example programs for Arduino Mega 2560 was obtained. Finally, the coverage (or percentage of program that contains the most common sequences) is calculated in for the popular programs. The overall process was done when considering sequences of two, three, and four instructions for comparison.

7.2.2 Results of the Study

Amount	Test Set Covered
100	60.30%
200	75.30%
300	82.77%
400	87.05%
500	90.48%
600	92.71%
700	94.09%
800	94.89%
900	95.88%
1000	96.61%

(a) Two Instructions

Amount	Test Set Covered
500	51.44%
1000	63.24%
1500	70.29%
2000	74.70%
2500	78.41%
3000	81.01%
3500	83.58%
4000	85.23%
4500	86.79%
5000	87.83%

(b) Three Instruction

Amount	Test Set Covered
1000	40.68%
2000	50.58%
3000	56.52%
4000	60.48%
5000	63.61%
6000	65.90%
7000	67.71%
8000	69.31%
9000	70.65%
10000	71.76%

(c) Four Instructions

Table 7.1: Estimated coverage for any program given amount of most common sequences. Results provided when considering two, three, and four instruction sequences.

The results of the test are provided in Table 7.1. The results show that when accounting for two instruction sequences, the average percent covered in different programs is slightly above 90% when utilizing only the top 500 sequences seen in the training set. The average converted percent decreased to 82.77%, 75.30%, and 60.30% when utilizing the top 300, 200, and 100 two instruction sequences respectively. When evaluating using three instruction sequences, the amount of the top number of sequences necessary to achieve minimal coverage drastically increases. To achieve above 80% coverage of any program, roughly 3000 of the most frequently seen three instruction sequences are necessary. The use of fewer most frequently seen sequences decreases the coverage even further, with 2000, 1500, and 1000 sequences providing an average coverage of 74.70%, 70.29%, and 63.24% respectively. Results using four instruction sequences requires an even greater number of frequently seen sequences. Achieving minimal coverage of 70.65%, 65.90%, and 60.48% when using 9000, 6000, and 4000 of the most frequently seen sequences respectively. It should be noted that even with four instruction sequences, the necessary amount to obtain around 70% coverage (9000 sequences) is significantly lower than all possible using an Arduino Mega 2560 CPU (125^4 sequences).

Hence, the question becomes, can a deep learning model that is trained on just the most common samples generate EMs similar to the real captured observations?

7.3 GANs model for generating EM signals

Out of multiple possible data-driven generative translation models, GANs stood out as a possible solution for code-to-EM signal translation while training on the only most common samples. This is due to the success that GANs models have in text-to-speech translation [19], [36], [25].

The task of text-to-speech synthesis bears significant similarities to code-to-EM signal translation in that raw text-based input is passed through a generator to create signal data. The main differences lie primarily in the morphology of the input and output. More specifically, the input for text-to-speech and code-to-EM signal translation can be of any length of words in a given language. Similarly, in this context *code* can be any one of the supported instructions for the target CPU.

Another reason for GANs is their ability to generate data that it is not specifically trained on. In fact, this ability has been used to train other models and algorithms [37], [38], [39]. The ability to generate new, accurate data to train other models is of particular interest as our main motivation is the development of a scalable EM-based anomaly detection model for the detection of malicious code alterations.

7.4 Addressing Challenges of Using GANs for Code-to-EM Signal Translation

Three major challenges may negatively impact the efficiency of GANs models to generate accurate EM signals. First, the morphology (mainly the amplitude) of EM signals that correspond to a specific instruction are influenced by prior instructions as previously indicated in Section 2.1.1. Second, GAN models are known to have convergence problems, such as vanishing gradients, mode collapse, and failure to improve [27], [28]. Third, GANs output must be in a specific shape and size. Unfortunately, the time index of an instructions EM signal can vary slightly. Furthermore, if a process of combining EM signals of instructions is done to create a full EM without a common point of intersection, the resulting synthetic EM sample may contain amplitude discrepancies. Below we will describe specific actions taken to resolve these issues.

7.4.1 Prior Execution Influence

To account for the issue of prior instruction influence, the generator, and discriminator are additionally fed a token for the prior instruction. Input to the generator and discriminator maintains the order of execution (i.e., prior then current instruction). While the direct prior instruction inflicts the majority of the distortion, instructions that are located further back may also play a role. Therefore, we hypothesize that including more tokens of prior instructions might further increase the fidelity of the resulting signal. However, providing long sequences of instructions as inputs may be challenging in practice. The reader should keep in mind that constructing a database of pairs of instruction archetypes has a significantly larger size. For example, the Atmel Atmega328P CPU architecture supports 123 unique ASM instructions which are relatively easy to fingerprint. However, fingerprinting all possible pairs of these instructions is significantly more challenging as the total number of unique pairs is 15,129. Furthermore, increasing the knowledge of past instructions has diminishing returns as an instruction's residual influence decreases over time.

7.4.2 Convergence

The problem of convergence was addressed using the method of Wasserstein GANs with Gradient Penalty (WGANs-GP) [29].

Wasserstein GANs implements Wasserstein loss and converts the discriminator into a critic, changing the traditional GANs loss function into:

$$W_{Loss} = \min_{G}\max_{D} \mathop{\mathbb{E}}_{x\sim\mathbb{P}_r}[D(x)] - \mathop{\mathbb{E}}_{G(z)\sim\mathbb{P}_g}[D(G(z))] \tag{7.3}$$

Wasserstein GANs is further explained in Section 2.2.3.1.

Additionally, a gradient penalty is implemented to constrain the gradient norm of the discriminator output with respect to its input, preventing vanishing gradients. As such the W_Loss is further improved to:

$$\min_{G}\max_{D} \underbrace{\mathop{\mathbb{E}}_{G(z)\sim\mathbb{P}_g}[D(G(z))] - \mathop{\mathbb{E}}_{x\sim\mathbb{P}_r}[D(x)]}_{Original\ W\_Loss} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}[(\nabla_{\hat{x}}D(\hat{x})_2 - 1)^2]}_{Gradient\ Penalty} \tag{7.4}$$

Where λ is the penalty coefficient and $\mathbb{P}_{\hat{x}}$ is the distribution sampling uniformly along straight lines between pairs of points sampled from $\mathbb{P}_r$ and $\mathbb{P}_g$.

The reader should note that the use of WGANs-GP only helps to prevent but not fix the issues of convergence. Methods to permanently fix such problems in GANs models are a topic of ongoing debate.

7.4.3 Length of Input and Output

Deep learning models must have a fixed size of data that is provided as input and output. Unfortunately, various EM signals of a given instruction can vary by several time indexes as shown in Figure 7.1.

To account for this, additional time indexes were taken for both the training samples and the generated single EM instruction. This amount of extra information is to the last bottom peak of the previous instruction. Furthermore, time indexes are equally cut from the beginning and end of the resulting signal to the minimum size to fulfill this requirement. For example, if the EM signals of all instructions with the additional information range from 76 to 80, all signals will be cut to 76.

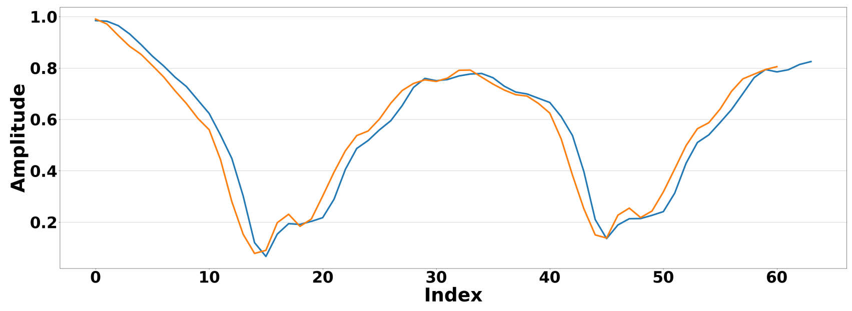


Figure 7.1: Two EM signals of the same instruction given by color variation. Note that the time length of the orange signal is less than the blue.

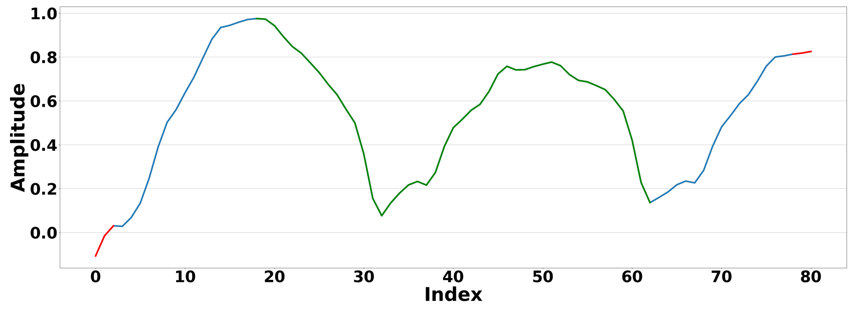


Figure 7.2: EM signal for an instruction stored in the library. Red lines indicate the removed extra information. Green indicates the actual information of the instruction.

To reiterate, the EM instructions are captured with a set index time u, which is the index between the bottom peak of the previous instruction to the second high peak. Then all EM signals are cut to the minimum possible time index u_{min}, removing from the start and end of the signal equally. This process is given in the following formula.

$$Signal = Signal[0 + (\frac{u - u_{min}}{2}) : u - (\frac{u - u_{min}}{2})] \tag{7.5}$$

Figure 7.2 illustrates the process outlined above.

Chapter 8

A GAN Framework for Code-to-EM Signal Translation

To address the challenges identified in the prior chapter, specifically *having to create an exhaustive library* and *requiring a large number of observations corresponding to the same instruction*, we evaluate using a data-driven generative model. Generative Adversarial Networks for Code to Electromagnetic Signal translation (GANs-COD2EM) is a framework involving GAN models that aims in converting a program given as a series of assembly (ASM) instructions into the corresponding electromagnetic (EM) signal. The resulting signal aims to be a realistic representation of the corresponding EM that would be produced by the CPU of a given device as a side-channel during the execution of that program in a real-life environment. The synthetic signals can later be used for various side-channel analysis tasks including anomaly detection.

GANs-COD2EM builds upon basic GAN architectures and draws inspiration from the work presented in [4], which introduces a framework for generating images from textual input. Similarly, GANs-COD2EM alters text into a structured and numerical format. Among the notable adjustments made is that internally, GANs-COD2EM makes use of a tokenizer, rather than text embeddings, to create tokens or unique number representations for each executable ASM instruction. This token is appended to the end of random noise and fed to the generator. The GANs-COD2EM generator learns which EM signal to generate based on the corresponding token. This process resembles the operation of Conditional GANs, (the reader should refer to Section 2.2.4). The generated and real EM signals are then paired with the corresponding token of the executed instruction to later be discerned by the discriminator. Iteratively, through this process, the entire model is trained. A high-level overview of the proposed framework is presented in Figure 8.1.

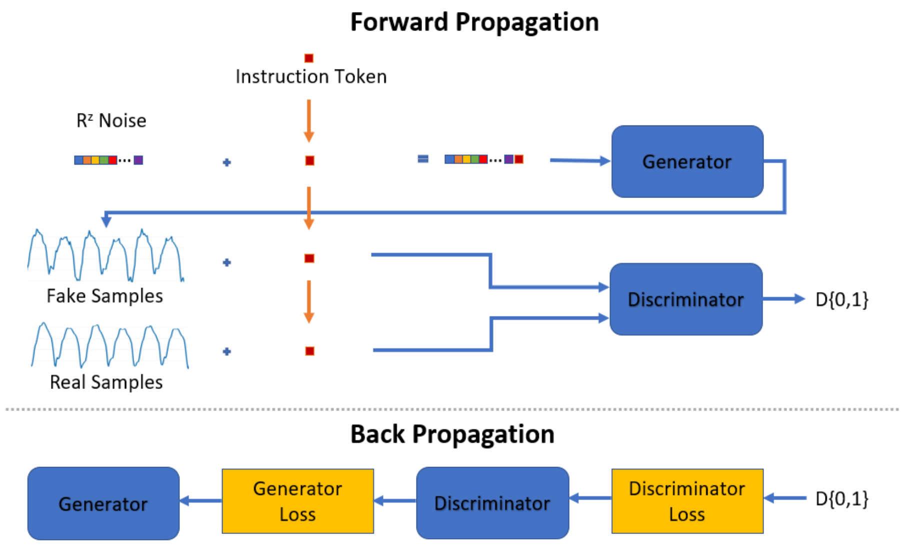


Figure 8.1: Workflow of the proposed framework.

8.1 Network Architecture

The network architecture is described using the following notations. The generator network is denoted as $G : \mathbb{R}^{Z+T} \to \mathbb{R}^S$, and the discriminator as $D : \mathbb{R}^{S+T} \to \{0, 1\}$, where T is the dimension of the token, S is the dimension of the signals, and Z is the dimension of the noise input given to G.

During a preprocessing step, a tokenizer dictionary for ASM instructions is set up. In this context, tokenization is a process that vectorizes programs by turning ASM instructions into a sequence of integers. Each of the resulting integers corresponds to an index of a token in the tokenizer dictionary. To achieve a complete tokenizer dictionary, all possible ASM instructions are fed to the tokenizer. Afterward, the tokenizer will output the unique corresponding index when given an instruction.

The generator first samples from the noise $z \in \mathbb{R}^z \sim N(0, 1)$ and the token for the instruction i is obtained using the tokenizer φ. The resulting token t is concatenated to the end of the noise vector z and feed-forward through the generator G. As such, a generated signal $\hat{s}$ is produced via $G(z, t) \to \hat{s}$. This process is illustrated in Figure 8.2.

The discriminator D acts similarly to a conditional GAN, labeling pairs of $\{$signal, instruction$\}$ as real or fake. The model takes a signal as a one-dimensional numerical sequence of values s, which is associated with the token t. The latter indicates the instruction that matches the corresponding signal. The discriminator determines the likelihood that the signal for the instruction is real via $D(s, t) \to \{0, 1\}$.

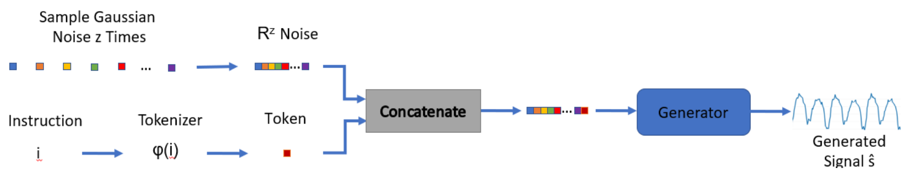


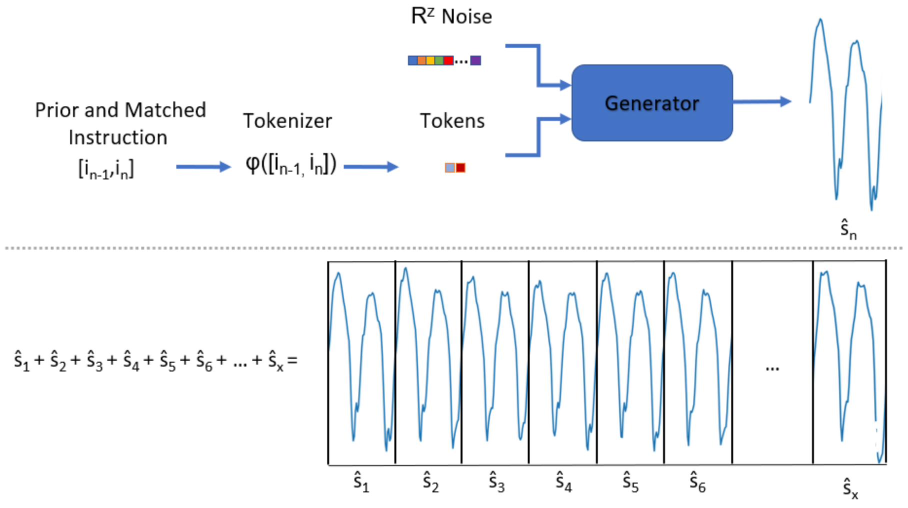
Figure 8.2: Process of generating signal.

8.1.1 Training

Algorithm 3 illustrates the training procedure. After obtaining the token for the instruction, a sample of random noise is taken and used with the token to generate the fake signal $\hat{s}$ (lines 3-5). Next, the scores for the real and fake signals paired with the token are calculated (lines 6-7). Lines 8 and 10 evaluate the loss and then updates the gradients, with L_D and L_G indicating the loss for the discriminator and generator respectively and α being the step size to update the weights. Lines 11 and 13 indicate the gradient step to update network parameters.

Algorithm 3 GAN_CTEM training algorithm:
with step size α

```
 1: function TRAIN:(minibatch signals $s$, matching instruction $\underline{i}$, number of training batch steps $S$):
 2:     for c = 1 to S do
 3:         $t \leftarrow \varphi(\underline{i})$                          comment: Retrieve token for matching instruction
 4:         $z \sim N(0,1)^Z$                          comment: Draw sample of random noise
 5:         $\hat{s} \leftarrow G(z,t)$                          comment: Forward through generator
 6:         $s_r \leftarrow D(s,t)$                          comment: Discriminate real signal
 7:         $s_f \leftarrow D(\hat{s},t)$                          comment: Discriminate fake signal
 8:         $L_D \leftarrow log(s_r) + (log(1-s_f))$                          comment: Discriminator loss
 9:         $D \leftarrow D - \alpha \partial L_D / \partial D$                          comment: Update discriminator
10:         $L_G \leftarrow log(s_f)$                          comment: Generator loss
11:         $G \leftarrow G - \alpha \partial L_G / \partial G$                          comment: Update generator
12:     end for
13: end function
```



R^z Noise
Prior and Matched Instruction
$[i_{n-1}, i_n]$
Tokenizer
$\varphi([i_{n-1}, i_n])$
Tokens
Generator
$\hat{s}_n$
$\hat{s}_1 + \hat{s}_2 + \hat{s}_3 + \hat{s}_4 + \hat{s}_5 + \hat{s}_6 + \ldots + \hat{s}_x =$
$\hat{s}_1$
$\hat{s}_2$
$\hat{s}_3$
$\hat{s}_4$
$\hat{s}_5$
$\hat{s}_6$
$\hat{s}_x$

Figure 8.3: Example of generating a full EM using GAN-COD2EM.

8.2 Generating Full Program EMs

GAN-COD2EM can further be used to synthetically generate the EM signal of any program through a process of appending. In the order of the program's code, each instruction's EM is generated. However, the reader should recall that each instruction contains additional time indexes portrayed in Section 7.4.3. As such, the extra indexes are cut and removed from the sample. Afterward, the sample is appended to the end of a collected EM signal. This process is repeated till all ASM instructions seen in a given program are processed. The process of generating a full EM given *x* instructions is illustrated in Figure 8.3.

8.3 Updated Training Algorithm

The training algorithm from Algorithm 3 is in a naive form and cannot address the challenges presented in Chapter 7. As such, the training algorithm is altered to account for these issues and is given in Algorithm 4. As a first step, the tokens for the previous and matching instructions are retrieved in Line 5. Next, in Lines (6-7) random noise is taken with the tokens and forwarded through the generator to obtain fake signal $\hat{s}$. In lines 8-9, the discriminator evaluates the real and fake with the corresponding tokens. Next, lines 10-13 estimate the loss and updates the weights for the discriminator by applying the gradient penalty to the W_{Loss}. Finally, the loss is calculated and weights are updated for the generator in lines 14 and 15.

Algorithm 4 GAN_CTEM training algorithm:
with step size α

```
1:  function GRADIENT_PENALTY:(initial discriminator parameters w, Discriminator output given fake data $\hat{x}$)
        return $(\nabla_{\hat{x}} D_w(\hat{x})_2 - 1)^2$
2:  end function
3:  function TRAIN:(minibatch signals s, matching code instructions i, index of matching instruction n, the previous number
    of training batch steps S, gradient penalty coefficient $\lambda$):
4:      for c = 1 to S do
5:          $t \leftarrow \varphi([i_{n-1}, i_n])$          comment: Retrieve token for prior and matching instruction
6:          $z \sim N(0,1)^Z$                               comment: Draw sample of random noise
7:          $\hat{s} \leftarrow G(z,t)$                     comment: Forward through generator
8:          $s_r \leftarrow D(s,t)$                         comment: Discriminate real signals
9:          $s_f \leftarrow D(\hat{s},t)$                   comment: Discriminate fake signals
10:         $W_{Loss} \leftarrow -mean(s_f) + mean(s_r)$
11:         $gp \leftarrow GRADIENT\_PENALTY(w_n, \hat{x})$
12:         $L_D \leftarrow W\_Loss + \lambda * gp$         comment: Discriminator loss
13:         $D \leftarrow D - \alpha \partial L_D / \partial D$   comment: Update discriminator
14:         $L_G \leftarrow -mean(s_f)$                     comment: Generator loss
15:         $G \leftarrow G - \alpha \partial L_G / \partial G$   comment: Update generator
16:     end for
17: end function
```

Chapter 9

GANs Experimental Setup & Data Gathering

To evaluate the proposed EM-based anomaly detection using GANs, we utilized the experimental setup defined in Section 5.1. Furthermore, we create a new program we call *Alpha* that is a shorter version of *Program B* in the prior experiments. Additionally, we utilize the framework from Chapter 6, with the exception of altering step ❷ in Figure 4.1 to utalize a GANs model to generate the EM signals. These generated EM signals are used during the fingerprinting stage to create the baseline and train the detection engine. Similar to before, we create malicious cases of the benign program by injecting four and two instructions to *Alpha*. At the deployment stage, we collected signals while the device was operating under normal as well as anomalous states (i.e., after the injection). The predictive accuracy of our new system using GANs is evaluated. Details regarding the process described above are given in this section.

9.1 Dataset

To evaluate our approach, we defined a base program to generate call *Alpha*. Program *Alpha* is the same as the updated program in the prior experiment in Chapter 4, except, only ten instructions are being synthetically generated from the original EM signal, in an attempt to emulate a software update scenario. The ASM code is provided in Listing 9.1.

Furthermore, malicious cases were also created that injected code into the middle of *Alpha*. This included a program called *Malicious Easy* that injected four instructions and is easier to detect out of the two cases. The other injected only two instructions and is harder to detect and as such is called *Malicious Hard*. Samples of the code for the two malicious cases are provided in Listings 9.2 and 9.3.

```
setup:
  sbi ddrb, 6  ; set pb6 as output (our sync artifact)
  loop:
  sbi   pinb, 6
  ; simulates a decision
  clr   r20
  ldi   r20,  1
  ; Will go to the first_label
  ldi r22, 1
  cp  r20, r22
  breq  first_label
  rjmp loop

first_label:
  ldi  r23, 0; start of ten instructions
  and  r2, r3 ; Bitwise AND (result in stored r2)
  add  r1, r2 ; Add r2 to r1 (r1=r1+r2)
  eor  r2, r3 ; Bitwise exclusive or between r2 and r3
  sub  r1, r2 ; Subtract r2 from r1
  ses ; Set signed flag
  cls ; Clear signed flag
  sev ; Set Overflow Flag
  clv ; Clear Overflow Flag
  clr r1 ; end of ten instructions

  rjmp loop
```

Listing 9.1: Assembly code of program *Alpha*. Highlighted instructions are the ones being captured and generated.

```
...
; up to here, same as "Alpha"

  add  r1, r2 ; Add r2 to r1 (r1=
    r1+r2)
  ;====== 4 injected instructions
    ========
  asr r3 ; r3=r3/2
  com r3 ; Take one's complement
    of r3
  adc r3, r2
  sbc r3, r2
  ;==============
  eor  r2, r3 ; Bitwise exclusive
    or between r2 and r3

; the rest are the same as "Alpha"
...
```

Listing 9.2: Assembly code of *Malicious Easy*.

```
...
; up to here, same as "Alpha"

  add  r1, r2 ; Add r2 to r1 (r1=
    r1+r2)
  ;====== 2 injected instructions
    ========
  asr r3 ; r3=r3/2
  com r3 ; Take one's complement
    of r3
  ;
  ;
  ;==============
  eor  r2, r3 ; Bitwise exclusive
    or between r2 and r3

; the rest are the same as "Alpha"
...
```

Listing 9.3: Assembly code of *Malicious Hard*.

To train the model, samples of the ten instructions were taken from the built library from the prior experiment in Chapter 4. The samples contained in the library are from different programs and contain 1000 samples of each instruction. Only the samples for the instructions seen in *Alpha* were used. In total, the training dataset contained ten instructions with 1000 samples each.

9.2 Comparing the Similarity between EM Signals

The following defines a method to evaluate the similarity of EM signals. For two sets of EM signals, a comparison process is first performed for each signal in the first set to every signal in the second set. Then an averaging process of the k-nearest neighbors or lowest distance results is done. This process is performed for all signals in the first set. Essentially, creating a range of values that indicated the range of similarity between two sets of EM signals that identify different programs.

For example, consider a case where there are 1000 signals in each set and we want to evaluate given 25 nearest neighbors. Then the process performed yields 1000 similarity scores for the first signal of set one. The scores of the 25 nearest neighbors are then averaged. The process is repeated for each signal in the first set, obtaining 1000 averaged scores for the given example.

The distance used to compare the two signals can use any distance measurement. The one used for evaluating GANs-COD2EM was Euclidean distance (ED), which is calculated as:

$$d(A,B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2} \tag{9.1}$$

where A and B are two EM signals. n is the number of time indices in the signals. A_i, B_i is the time index of the two signals at point i.

9.3 GANs models for Synthetic EM Generation

The following section describes several GANs models that were altered to utilize the framework of GAN-COD2EM, as portrayed in Section 8. These models are later used to validate the use of GANs for code-to-EM signal generation.

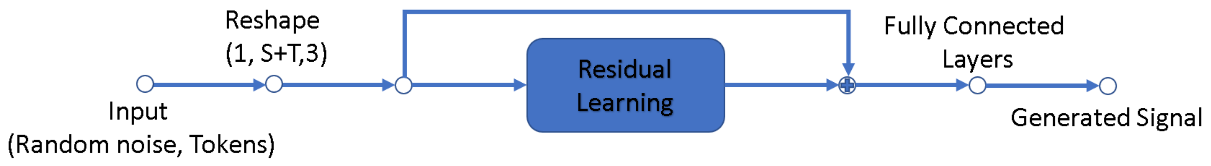


Figure 9.1: Basic structure of ResGANs-COD2EM Generator.

9.3.1 ResGANs-COD2EM:

ResGANs-COD2EM is a modified version of ResGANs [40] but adopts the framework of GANs-COD2EM. As with ResGANs, the generative model now utilizes ResNet [41], a commonly used Artificial Neural Network (ANN) model originally designed for image recognition.

Originally, ResGANs were used for image restoration tasks given coarse image features and attribute labels. ResGANs were able to achieve higher accuracy than traditional GANs through the use of ResNet inside the generator model. With ResNet, the input data is directly transferred to the outputs, providing residual learning and helping with vanishing gradients as data information is reapplied that is usually lost in traditional GANs. In terms of EM signal generation, the process can be denoted as:

$$X_g = f(g(X_r) + X_r) \tag{9.2}$$

where X_r is the input data including the random Gaussian noise and tokens, X_g denotes the output signal, and $g(X_r)$ is the residual signal to be learned by the generator.

The main challenge with code-to-EM translation is that the input is one-dimensional time series with tokens and not a three-dimensional image for which ResNet was originally designed. Fortunately, since the original paper, ResNet has been adapted with 1-d Convolution NN models for use with time-series data [42]. In order for ResNet to process a 1-d array, the input must be reshaped to include a third axis and proceeded through a 1-d convolution layer to create a 3-d input with the repeating 1-d time-series values along the third axis. After ResNet, the output can be processed through fully connected NN layers to further improve results. This process and use of ResNet are only used for the generator model. The discriminator does not make use of ResNet and only needs to contain fully connected NN layers. The basic structure of the generator with ResNet is shown in Figure 9.1.

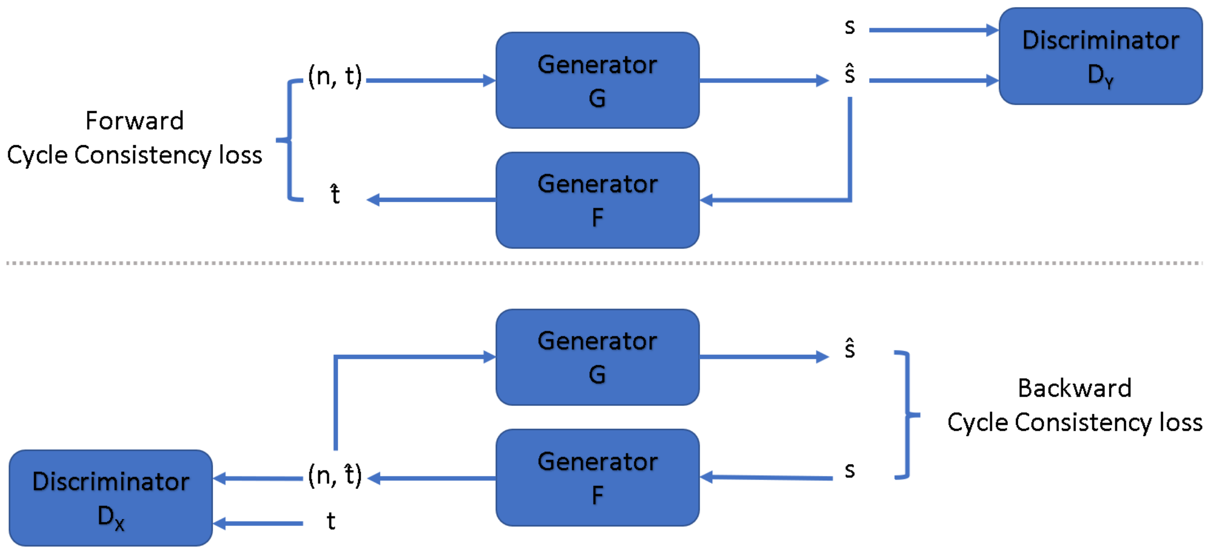


Figure 9.2: Structure of CycleGANs-COD2EM.

9.3.2 CycleGANs-COD2EM:

CycleGANs-COD2EM applies the GANs-COD2EM framework to the CycleGANs model, outlined in Section 2.2.4.1. Keeping with the same terms used in GANs-COD2EM, t is the instructions as indicated with tokens and s is an EM signal. For our purposes, the two forms being translated using CycleGANs, x and y in the CycleGANs structure given in Figure 2.6, will be t and s. Additionally, noise n will be passed to generator G as multiple EM signals s could exist for the pair t, s. The structure of CycleGANs-COD2EM is presented in Figure 9.2.

9.3.3 TransGANs-COD2EM:

TransGANs implement the attention techniques of transformer models as seen in Section 2.2.1, to the generator and discriminator of basic GANs. Attention-based methods improve GANs generation capabilities as it applies full knowledge of the entire sequence and forces the model to learn specifies of the training data, helping prevent failure to converge. In further detail, the general shape and prior instruction influence could be better modeled by implementing transformer-based techniques. Today, transformers have been found to outperform other techniques including various state-of-the-art NN architectures for sequence data such as analog/discrete signals. Moreover, compared to other NN approaches, they are found to require fewer data for model training.

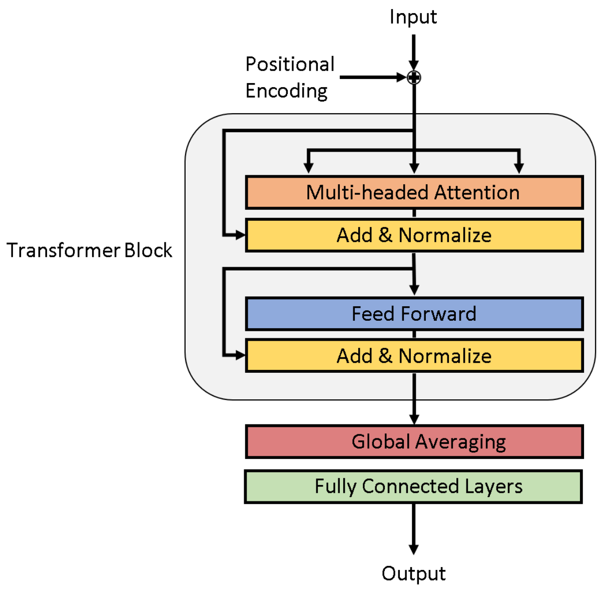


Figure 9.3: Architecture of TransGANs-COD2EM with one transformer block.

The original paper of TransGANs [26], makes use of the concept of attention for generative purposes. In specifics, the input for the model was random Gaussian noise converted into an 8x8xC image, where C is the dimensional embedding, using a multiple-layer perception. Then the image when through several layers of grid transformer blocks. Each grid transformer block performed grid-self attention on sections of an image to generate an upscaled or descaled version of the image section by section, thus requiring less memory.

TransGANs-COD2EM takes the concept of using attention techniques from the original TransGANs. However, the transformer block only performs standard multi-headed self-attention rather than grid-attention. Additionally, the transformer block was altered as the feed-forward process is done with one-dimensional convolution layers to account for one-dimensional time-series data. Furthermore, no upscale or downscale was performed. The basic structure of the TransGANs-COD2EM is illustrated in Figure 9.3. The reader should note that several transformer blocks could be added one after another.

TransGANs-COD2EM used for experiments contains one transformer block. This is due to resource requirements and higher amounts of transformer blocks that take more time to train. Additionally, more blocks showed minimal improvements in the initial experiments.

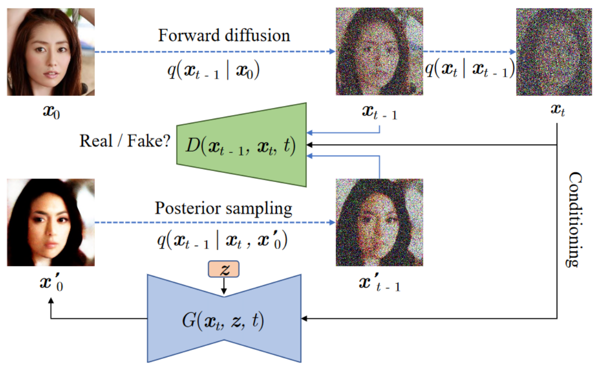


Figure 9.4: Process of DiffGANs for image generation from the original paper [3].

9.3.4 DiffGANs-COD2EM:

Denoise Diffusion GANs [3] or DiffGANs for short, was first introduced as a model to fix the issues with three generative models, namely GANs, denoising diffusion, and variational autoencoders (VAE). Each of these models has issues that the other does not. Denoise Diffusion suffered from being able to do fast sampling, GANs have model convergence and diversity issues, and AVE suffers when generating high-quality samples. To develop a model that could solve these issues, the structures of Denoise Diffusion and GANs were combined.

Denoise Diffusion, Section 2.2.2, slowly generates less noisy samples in subsequent steps. This concept is implemented in DiffGANs but used the GANs model to generate samples, rather than a mean function approximator. DiffGANs [3] go through a forward diffusion process of generating noised samples by applying Gaussian noise in steps. Then, in a process similar to Conditional GANs, Section 2.2.4, these steps are pasted into the generator with an additional conditioning identifier for which the diffusion step is being past in. The generated images with the conditional identifier are passed to the discriminator. The discriminator receives pairs of real and fake samples for each diffusion step and determines the viability of each. The goal of the generator is to fool the discriminator into thinking the fake samples produced are real for all diffusion steps. This process is illustrated in Figure 9.4.

DiffGANs-COD2EM implements the sampling process as DiffGANs. A slight adjustment of adding sequence tokens was applied to be able to generate EM signals of various instructions. Overall, the process of DiffGANs-COD2EM generator is $G(n,t,d) \rightarrow \hat{s}$ with d being the diffusion step identifier, and $D(s,t,d) \rightarrow \{0,\infty\}$ for the discriminator.

DiffGANs-COD2EM used for experiments accounted for four diffusion steps. The noise per step was given using a signal sample with Signal-to-Noise Ratio of ten and increasing by ten for the following steps, with the final step being the pure captured signal for the instruction.

Chapter 10

Evaluation of Generating EMs through GANs

The following chapter contains a detailed description of the experiments for evaluating the proposed GANs framework for generating EM signals. Evaluations target primarily the accuracy of the anomaly detection task.

10.1 Experiment One: Generating Full EM Signals

As an initial test, validation of the use of GANs for synthetic EM generation was performed in generating the EM signal of the program *Alpha* given in Section 9.1. However, under real scenarios, capturing the EM signals for every sequence pair of instructions is nearly impossible. As such, we considered five cases in which the model has been trained with 100%, 90%, 80%, 70%, and 60% of the instructions that comprise the *Alpha*. These tests were done in an effort to simulate the case where we aim to generate signals of a program whose instruction sequences are partially unmodeled. Furthermore, each test was evaluated using ResGANs-COD2EM, CycleGANs-COD2EM, TransGANs-COD2EM, and DiffGANs-COD2EM as described in Chapter 8 to identify which model performs the best.

In order to generate the full EM signals, a process of generating each instruction, cutting excess information, and appending is performed as defined in Section 8.2. The evaluation was performed by estimating the similarity between the generated and the real captured signals of *Alpha* using the method defined in Section 9.2 and averaging the result. The evaluation method will provide a general similarity between the 500 generated signals and the 500 real signals. Note that as the model can generate potentially infinite samples, sometimes the results could be lower or higher. However, the results will remain generally in the same area (± 0.005).

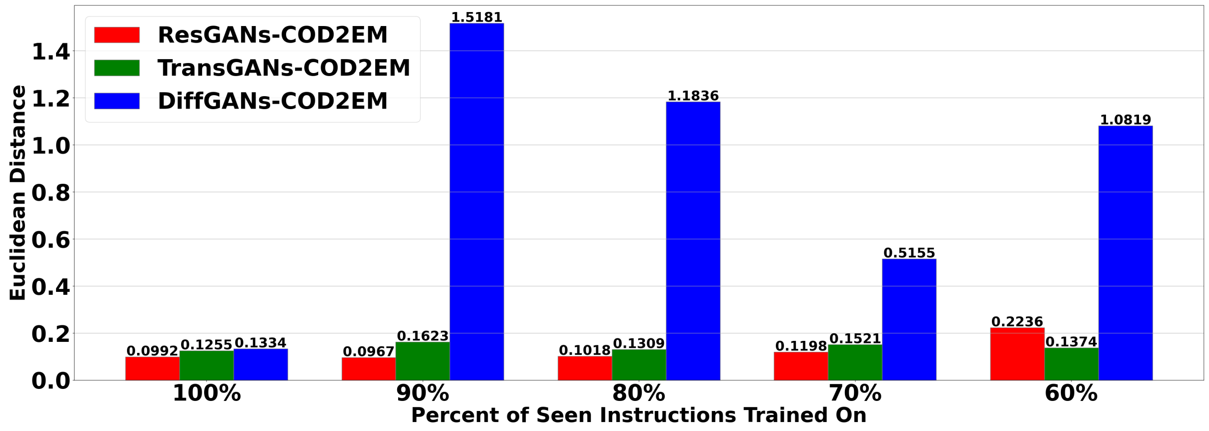


Figure 10.1: Average Euclidean distance between various GANs-CTEM models generating the full EM signal when training on 100%, 90%, 80%, 70%, and 60% of seen instructions of the program *Alpha*.

10.1.1 Results

The results of generating the full EM signal are provided in Figure 10.1. Furthermore, samples of the real and generated signals are provided in Figure 10.2.

The similarity metric employed is Euclidean distance, therefore, the lower the average, the closer to the real EMs. The reader should notice that all models performed well when trained on 100% of the seen instructions, with ResGANs-COD2EM achieving the best result at 0.0992 average distance. In fact, ResGANs-COD2EM provides the poorest accuracy in the majority of tests, achieving 0.0967, 0.1010, and 0.1198 when training on 90%, 80%, and 70% respectively. This provides evidence that the ResGANs-COD2EM is the best overall model for code-to-EM signal translation. However, with 60% training on seen instructions of *Alpha*, ResGANs-COD2EM achieved a distance of 0.2236. In comparison, TransGANs-COD2EM recorded an average distance of 0.1376 at 60%. Indicating that TransGANs-COD2EM can help maintain higher similarity when training on fewer instructions. DiffGANs-COD2EM did not perform well when training on less than 100% of seen instructions in program *Alpha*, obtaining high average distances in the order of 0.5155 or above. This indicates that the particular model cannot accurately and consistently generate signals of instructions it has not been specifically trained on.

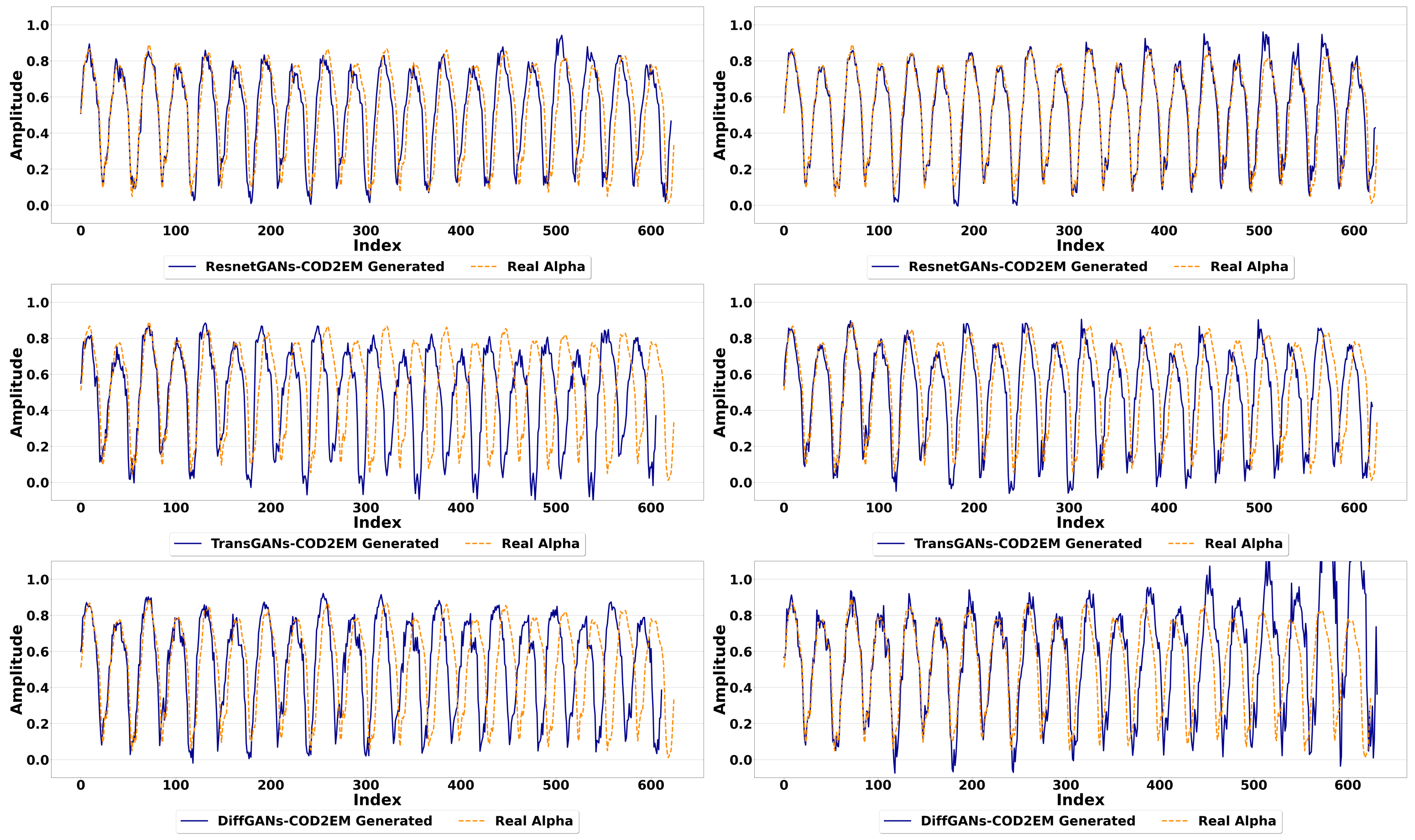


Figure 10.2: Samples of the generated signals compared to the real signal of *Alpha*. Rows are organized from top to bottom as samples from ResnetGANs-COD2EM, TransGANs-COD2EM, and DiffGANs-COD2EM. Columns indicate the training percentage, ordered by 100% and 60% respectively.

10.2 Experiment Two: Generating EM Peaks

A second experiment was performed in a similar fashion, with the exception that the models are configured to generate only the peaks of the signals per cycle. This was done to improve the results of the initial experiment.

The EM peaks are the main indication of the general operation being performed at a given time frame. As such, many techniques, such as anomaly detection, can perform by only utilizing the EM peaks. Furthermore, having the model generated two samples for the high peaks rather than the full EM signal for an instruction focus the model to accurately account for these values. Thus, improving results for the peak values and the detection capabilities.

The dataset used for evaluation was the peaks of the real signals of *Alpha*. Generating a full EM of peaks was done similarly to the method used in the previous experiment, excluding the need to remove additional information as only the peaks are generated. The evaluation was done using the same calculation of similarity defined previously.

10.2.1 Results

Results of generating the EM peaks are provided in Figure 10.3. Additionally, samples of the generated peaks are given in Figure 10.4.

All results for every model, the use of the peaks features instead of the entire raw signal, drastically improved the fidelity of the generated signal. Furthermore, the influence of training on fewer instructions for the generated signal is easier to see. ResGANs-COD2EM obtains the lowest average distance results across all models. Specifically, ResGANs-COD2EM achieved 0.0591, 0.0650, 0.0681, 0.0865, and 0.1016 when training on 100%, 90%, 80%, 70%, and 60% of instructions seen in *Alpha*. For the majority of the experiment, TransGANs-COD2EM performs the second best achieving scores of 0.0822 to 0.1538. DiffGANs-COD2EM still provides the highest results when training on less than 100 percent of seen instructions in *Alpha*.

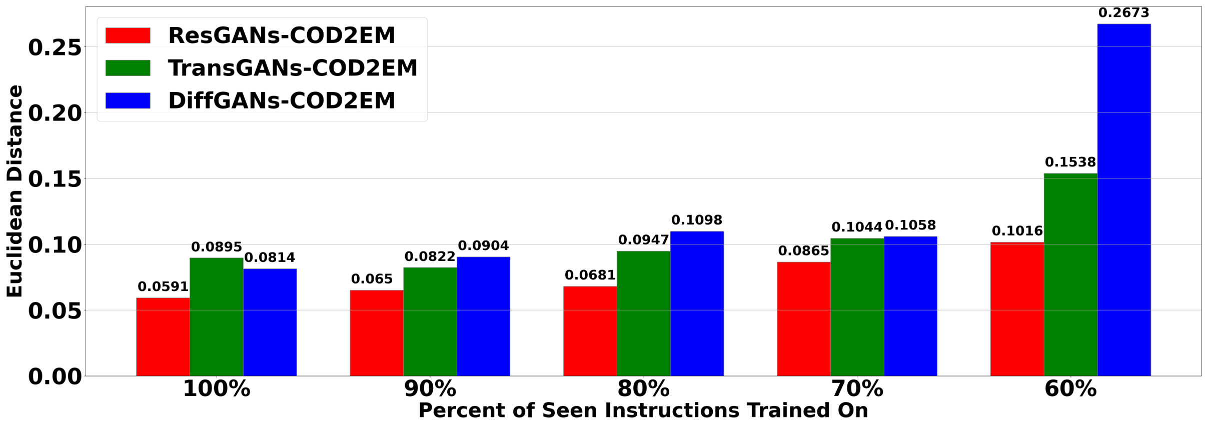


Figure 10.3: Average Euclidean distance between various GANs-CTEM models generating EM peaks when training on 100%, 90%, 80%, 70%, and 60% of seen instructions of program *Alpha*.

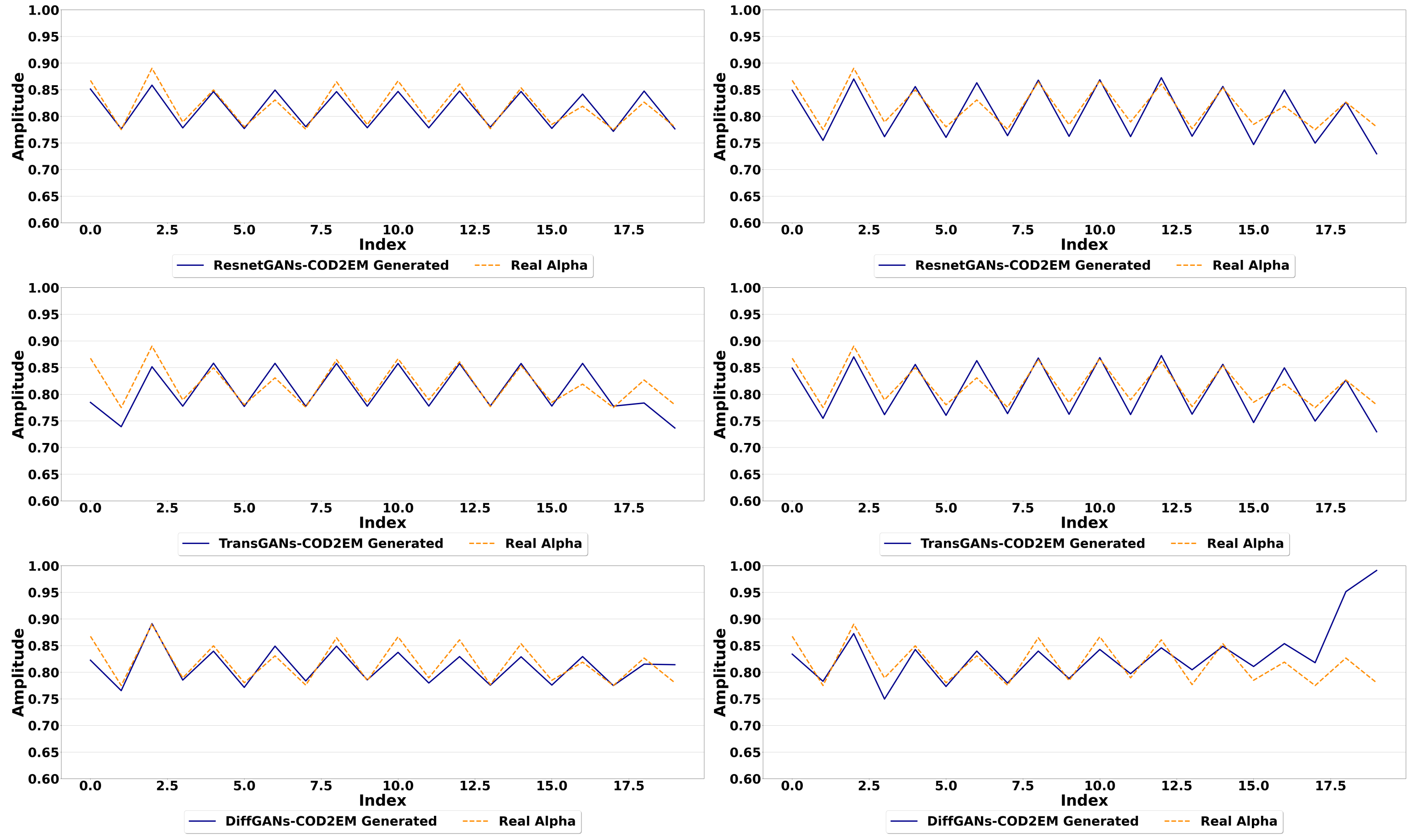


Figure 10.4: Samples of the generated peaks compared to the real signal of *Alpha*. Rows are organized from top to bottom as samples from ResnetGANs-COD2EM, TransGANs-COD2EM, and DiffGANs-COD2EM. Columns indicate the training percentage, ordered by 100% and 60% respectively.

10.3 Experiment Three: Training Time

The focus lies in the time required to train the models on 1000 samples for ten instructions per epoch. This serves as a general indication of the resource requirements between each model.

Results were obtained when running models on an NVidia RTX 3080. Models used are the versions indicated in Section 9. Each model generally takes 30 to 50 epochs on average to train the model to generate accurate signals. As such, the amount of time per epoch serves as an indication between the models as to which is more efficient given available resources.

Model	**Time Per Epoch**
ResGANs-COD2EM	620s
TransGANs-COD2EM	1800s
DiffGANs-COD2EM	830s

(a) Training Time To Generate Full Signals

Model	**Time Per Epoch**
ResGANs-COD2EM	420s
TransGANs-COD2EM	1120s
DiffGANs-COD2EM	640s

(b) Training Time To Generate Peaks

Table 10.1: Average time in seconds to train for one epoch across different models. Times are provided when training to generate the full EM signal (left) and only the peaks (right).

10.3.1 Results

The time per epoch for each model is provided in Table 7.1. As expected, the time to train on peaks takes less time than training on full EM signals of instructions. Additionally, ResGANs-COD2EM takes the least amount of time to train, taking approximately 620 seconds and 420 seconds per epoch to train when utilizing full signals and only the peaks respectively. TransGANs-COD2EM unsurprisingly takes the most amount of time. On average, TransGANs-COD2EM takes 1800 seconds per epoch to train on full EM signals. This time is reduced to 1120 seconds when training on peaks. DiffGANs-COD2EM is also slower than ResGANs-COD2EM requiring 830 and 640 seconds when training on full signal and only peaks respectively.

TransGANs-COD2EM requires more time because it needs to obtain the key, and value dictionaries and pass them to each query. Each query is determined utilizing a potentially large dictionary. As such, when using Transformer blocks with one GPU, this can be very slow. Note, however, that with multiple

GPUs, one can speed up the process by assigning each to estimate the different queries in parallel. For this reason, we can claim that this approach is much more scalable than other models with similar amounts of GPUs. Additionally, more transformer blocks could be added to improve the accuracy of generated samples; however, this then becomes a balancing act between performance time and accuracy.

It should be noted that DiffGANs-COD2EM is also expected to be slower than ResGANs-COD2EM. This is because, despite not having a resource-heavy Resnet layer, the model essentially contains additional values to train and account for. With the given input of the number diffusion step s, the amount of training is increased to t x s, where t is the number of training samples. The use of four training steps increases the time to train beyond ResGANs-COD2EM. The use of only two training steps would be faster, and would probably sacrifice the accuracy of generating samples.

10.4 Experiment Four: Acceptable Range of Distribution

To identify if the synthetic signals are valuable for the task of anomaly detection, an additional experiment was performed to identify the range of acceptable similarity. Obviously, the generated signals are expected to be less similar to real captured signals. Nevertheless, it is expected that the generated signals should still be more similar to the real EMs of the program than other programs or (malicious) modifications of these programs.

To calculate the similarity, the same method used in the previous two experiments was used. To obtain a baseline, 500 samples of *Alpha* were compared against 500 other samples of *Alpha*. Additionally, *Alpha* was compared against two malicious programs that inject two different amounts of instructions. Those are the malicious programs in Section 9.1. Note that in order to achieve good anomaly detection, the similarity score of the generated signals must clearly be separated and below the similarity for the malicious examples.

10.4.1 Results

The similarity results are provided in Figure 10.5. As shown, there is a clear separation between the comparison to other samples of *Alpha* and the two malicious versions. The range of the similarity distances of *Alpha* to other signals of the same program is roughly 0.015 to 0.035. For the malicious

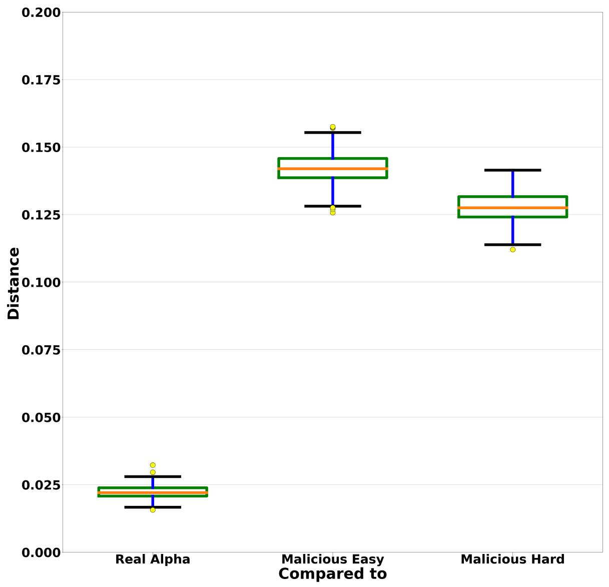


Figure 10.5: Similarity between the real EM signals of *Alpha* compared to itself, program with four injections (*Malicious Easy*), and a program with two injections (*Malicious Hard*)

cases, similarity distances are around 0.125 to 0.155 and 0.110 to 0.140 when compared to *Malicious Easy* and *Malicious Hard* respectively. As such, generative models that obtained a result of averaged similarity below 0.100, should obtain high accuracy when used for detecting programs outside the normal distribution of *Alpha*.

10.5 Experiment Five: Anomaly Detection Results

To further evaluate the proposed framework for generating EM signals. We performed anomaly detection using the best model when under perfect training conditions. In further detail, we used ResGANs-COD2EM, which trained on 100% of the instructions seen in *Alpha*, to generate signals of *Alpha* for training an Anomaly Detection method.

The dataset used for this experiment is the same as in the previous one, utilizing samples of *Alpha*, *Malicious Easy*, and *Malicious Hard*. Furthermore, we used the same anomaly detection method outlined in Chapter 4. To be more detailed, the training dataset contains 500 samples of the generated signals, the testing set contains 50 samples of real captured *Alpha*, and 50 samples of the malicious case. Results were analyzed using 10-fold cross-validation evaluating with a threshold of 0 to 1 skipping by 0.001.

Evaluation was done on the same metrics as in Chapter 6, those being the average AUC, accuracy, and F1 scores among the folds and the ROC graphs.

Table 10.2: Anomaly detection results.

Data	**Training**	**Test-Normal**	**Test-Anomaly**	**Scores (Avg.)**		
				AUC	**ACC**	**F1**
Full Signal	Synthetic Alpha ResGANs-COD2EM	Real Alpha Signals	Malicious (Easy)	0.993	0.986	0.986
			Malicious (Hard)	0.981	0.945	0.946
Peaks Only	Synthetic Alpha ResGANs-COD2EM	Real Alpha Signals	Malicious (Easy)	1.000	1.000	1.000
			Malicious (Hard)	1.000	1.000	1.000

10.5.1 Results

Results of the experiment are provided in Table 10.2 and the ROC graphs in Figure 10.6. The results of the experiment overall achieved near-perfect accuracy, with the best threshold found on average is 0.75 for when using full signals and 0.8 when using peaks. The best threshold can vary slightly from folds due to samples used for creating the baseline. The use of fully generated signals using ResGANs-COD2EM achieved an AUC of 0.993 and 0.981 when testing against samples of *Malicious Easy* and *Malicious Hard* respectively. The ACC and F1 scores, were 0.986 and 0.986 respectively when testing against *Malicious Easy*, and 0.945 and 0.946 respectively when testing against *Malicious Hard*. These results improved 100% when generating, comparing, and testing using only the peaks.

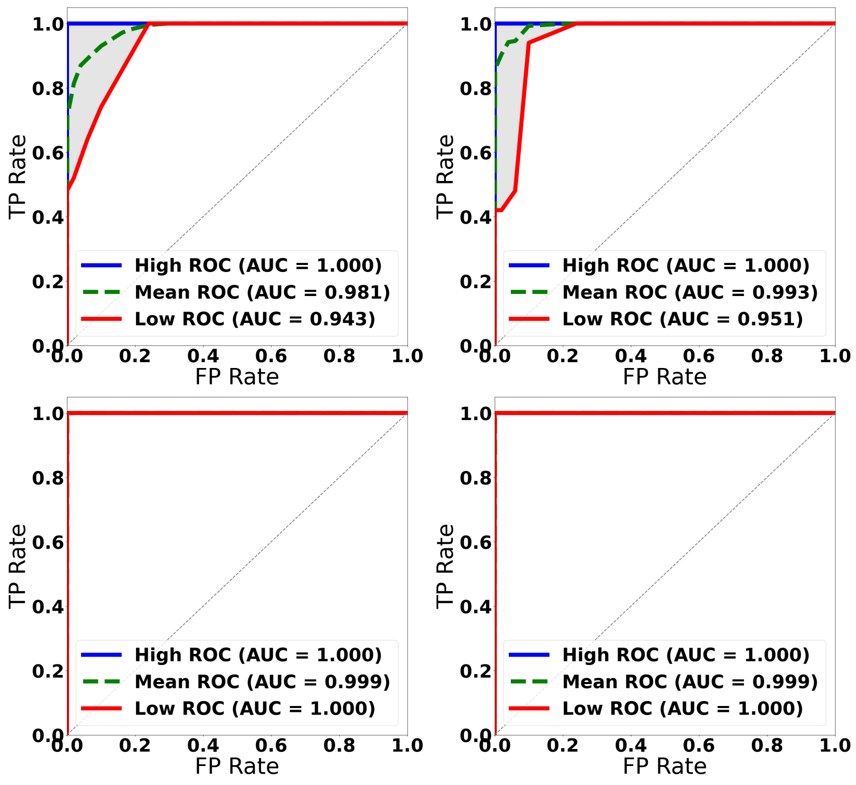


Figure 10.6: ROC graphs for the anomaly detection experiment. In the upper row, are the results obtained when generating the full EM signals for training. In the lower row, the results obtained when generation only the peak values for training.

Chapter 11

Discussion

Given the results from the prior chapter, there are GANs models that can generate accurate EM signals that can be used for anomaly detection purposes. Furthermore, the defined framework of GANs-COD2EM works well for binary code-to-EM signal translation.

11.1 Issues with CycleGANs

The reader probably noticed that CycleGANs was not included in the experiments in the prior chapter. This was due to CycleGAN's inability to accurately determine the binary code given the EM signal.

EM signals can provide general knowledge of the performed operation at any given point, however, the executed instruction cannot be derived given the signal. This is because of various factors, including the influence of prior instructions and small fluctuation in the amplitude, Section 2.1. Most importantly, from empirical observations, some of the EMs of certain instructions appear to be very similar. As such, the second generator and discriminator in the CycleGANs that was tasked to convert EM signals to binary code, *could not decipher which instruction is taking place given similar EMs*. Consequently, CycleGANs could not generate EM signals given code due to *requiring accurate feedback from both generators and both discriminators*.

11.2 Comparison Among the GAN Models

We provide the experimental results with three GANs-COD2EM models in Chapter 10. Out of the three models, ResGANs-COD2EM outperformed the other two models. When generating only the peaks and full EM signals, ResGANs-COD2EM provided the closest similarity to the real captured EM signals of program *Alpha*. This is probably due to the nature of Resnet, reapplying features, and helping the GANs model avoid convergence issues as indicated in Chapter 9. Additionally, perfect results were obtained when accounting for and generating only the peaks of the EM signal. Furthermore, ResGANs-COD2EM took the least amount of time to train, indicating that it requires less computational power. TransGANs performed well when comparing the similarity between the real and generated EMs, in some cases only slightly worse than ResGANs-COD2EM. However, TransGANs took the longest to train. DiffGANs-COD2EM can achieve good results when trained on all sequences that it is trying to generate. However, generating the EMs for unseen instruction sequences results in low accuracy around the corresponding locations in the generated EM signal.

Chapter 12

Related Work

12.1 Side-Channel Analysis

Side-channel analysis refers to the process of examining and extracting information from data that get involuntarily emitted from devices. Some examples include analyzing the audio signals, thermal transmission, or power consumption produced from a device during normal operation. Such analysis can be used to identify internal processes and executions. Once obtained, the information can be used for offensive or defensive purposes.

Offensive: Research works on side-channel analysis for offensive purposes include the use of power consumption [43] to obtain secret keys used from devices, identifying the keystrokes based on sound [44], or using the thermal information from a 3D printer to indirectly reconstructing the 3D objects being printed. For more information, [45] provides an overview of various side-channel analysis attacks. It should be noted that attacks using side-channel analysis are outside the main scope and focus of this paper. Instead, this paper focuses on the defensive uses of side-channel analysis, specifically used for anomaly detection.

Defensive: A family of novel defense systems is based upon the analysis of side-channels that get emitted constantly and involuntarily by various components [46], [47] of devices. Compared to the traditional network intrusion detection systems (N-IDS), such approaches may detect compromises and the execution of malicious code, even if the malware never produces any network footprint or if it remains in an installed-but-dormant state.

In the past, alternative types of side channels have been considered for defensive purposes, including the thermal emission profiles [48] or acoustic signals [7] of devices during their usual operational cycles. However, the current dominant methods of side-channel-based anomaly detection rely on the analysis of *power-consumption patterns* [8], [11]. This is primarily due to the ease of data collection and the robustness of this modality against environmental noise. Nevertheless, electromagnetic (EM) based approaches offer a comparative advantage since the signals themselves can be captured and analyzed in a completely non-intrusive fashion, i.e., no installation of software in the monitored device is assumed. Additionally, in contrast to power consumption signals, the EM spectrum offers high bandwidth and can be sampled at higher rates [9], [12].

12.2 EM Side-Channel Analysis

EM side-Channel analysis evaluates the electromagnetic emissions from a given device. These EM emissions can be generated from various components, such as the network controller chips, video displays, sensors, and actuators [49]. Out of all the components that could be analyzed, arguably the one that can convey the most valuable information is the processor. Consequently, major information on the general process being performed on a CPU can be derived given the EM signals. For more information on the morphology of the EM signals from the CPU, please refer to Section 2.1.

Similar to other forms of side-channel analysis, EM signals can be used to retrieve private information. In fact, EM signals can be used to also retrieve secret keys from a device [50], [51]. Additionally, Sayakkara et al. [49] provide a method to use EM side-channel analysis to identify cryptographic algorithms running on high-end IoT devices. Furthermore, authors of [52] state that EM emissions can be used to break cryptographic implementations while bypassing countermeasures against other types of side-channel attacks. However, more important to this research is EM side-channel analysis which may be conducted with the aim to detect attacks and/or malware infections.

In most cases, EM side-channel analysis used for defensive purposes utilizes anomaly detection techniques. Works that relate to EM-based anomaly detection include [17], which tests the limits of EM-based approaches by demonstrating the ability to identify the control flow of a given program, and showcase how it can be used to identify anomalous behaviors. Another example is given in [12], which presents a methodology for contactless security monitoring for programmable logic controllers (PLC), to ensure control flow integrity. The researchers behind the IDEA EM-based IDS [53] conducted their EM-based anomaly detection analysis fully in the time domain. The EM emanations from an uncompromised device are used to create a baseline *dictionary*. During the monitoring stage, the EM signal is split into windows that are then matched against *words* in that dictionary. The signal is then reconstructed using the matched words, and it is compared with the monitoring signal. Furthermore, EM-based anomaly detection has been able to identify anomalies, even in small portions of the program. In fact, EM-based anomaly detection tools have proven to be successful for the detection of extensive [13], [14], or even minimal modifications, say, down to the injection of a few instructions (at the assembly level) [15], [16]. Unfortunately, EM-based anomaly detection methods face challenges of feasibility in real settings.

12.2.1 Challenges of EM-based Anomaly Detection

EM-based anomaly detection methods have not been widely used in real settings due to three main challenges. First, many prior works required the use of expensive equipment to obtain clear EM signals. Second, not many works address the issue of environmental noise. An issue that is ever present in all real settings such as industrial environments. Finally, and perhaps the most restricting, is that the traditional methods assume high-fidelity signal capturing that can only be achieved by a human expert. This process is known to be time-consuming, error-prone, and requires a trained professional.

Less Expensive Equipment: One challenge that has been addressed in other works is the need to use less expensive equipment. Boggs et al. [46] demonstrate the efficiency of EM-based anomaly detection systems using commercial off-the-shelf (COTS) hardware. Furthermore, they showcase the feasibility of such approaches being applied to a wide range of critical infrastructure devices. We further provided evidence of this by using inexpensive equipment for our experiments.

Environmental Noise: Environmental noise is a major factor. For example, many IoT devices produce EM emissions. This causes unwanted noise in real practice, such as in industrial settings, which decreases the signal-to-noise ratio, resulting in unclear signals. As such, methods to denoise these signals are required. Some prior works that address this issue are [15] and Miller's et all [16], which provides a novel for removing environmental noise based on SVD. However, while the use of SVD performs well, methods such as Denoise Diffusion may provide better results and is a focus for future works.

Large Amount of Manual Capturing: The methods proposed in all the prior works in this section are based on a traditional framework that performs fingerprinting of benign cases by manually capturing EM signals. This method requires a trained professional to capture the benign cases every time an update is applied to the base program. Furthermore, fingerprinting of every possible execution path is necessary. This tactic is known to be *error-prone*, *time-consuming*, and *cost expensive*. This major issue is the main challenge towards providing scalable EM-based anomaly detection and is the main reasoning behind the need for the work in Chapters 4 and 8.

12.3 Generative Processing

Generative processing is the operation of generating new data that is close to the real samples. This is usually done by converting random noise into data close to a training samples distribution. Some of the most popular generative models include Continuous Normalizing Flows (CNFs) [54], Variational Autoencoders (AVEs) [55], Transformers [1], Denoise Diffusion [2], and Generative Adversarial Networks (GANs) [22]. A subcategory of generative processors of particular note to this work is generative translators.

12.3.1 Generative Translators

Generative translators are models and methods to convert from one data type, class, or object to another. Furthermore, Generative translation has been gaining popularity in the past decade and is a main topic for deep learning models. While there are many others, some of the most popular categories of generative translators are text-to-speech, text-to-image, and image-to-image translation, all of which can be done using a GANs model.

Text-to-Speech: Text-to-speech is the process of transforming the written text of a given language into audio waves that represent human speech. Some uses of such a model include teaching new readers, helping people who have trouble speaking, and communicating with people in another language.

There are many works in the area of text-to-speech translation. Early works include *WaveNet* [56], a neural network model that estimates the joint probability of a waveform as a product of conditional probabilities through the use of dilated causal convolutional layers. Another text-to-speech translator is *Tacotron2* [57], which is composed of a recurrent sequence-to-sequence feature prediction network and a modified *WaveNet* model acting as a vocoder to synthesize time-domain waveforms from spectrograms. More recently, [20], Li et al. introduce a transformer-based text-to-speech model that outperforms many prior methods, including *Wavenet* and *Tacotron2*. Further details into transformer models are provided in Chapter 2.

GANs models have further improved text-to-speech translation accuracy. Some GANs models for text-to-speech include *TransGANs* [26] (a GAN that incorporates multi-headed attention from transformer models), and *DiffGANs* [25] (a GAN applying the concept of diffusion step denoising from diffusion networks). Both *TransGANs* and *DiffGANs models* are detailed in Section 9.3.

It should be noted that text-to-speech is similar to the goal of code-to-EM signal translation as they both convert text to signal.

Text-to-Image: Text-to-image translation is the operation of generating images from descriptive sentences or text. The generated images are usually in a three-dimensional format with each axis corresponding to the red, blue, and green color values for each pixel or square box area in an image.

One of the first works in text-to-image translation is *AlignDraw* [58], which extended the DRAW model to condition image captions. The *AlignDraw* model uses recurrent neural networks to apply attention at several steps, with each step generating an image by accounting for a noised version of a training image and the textual input. Another popular model for text-to-image translation is presented by Ramesh et al. [59], which autoregressively models text and image tokens as a single stream of data. Ramesh's proposed model generates images from the text by first compressing the training sample image using a discrete variational autoencoder. Second, the text and image tokens are retrieved and used to train an autoregressive transformer to model the joint distribution over the tokens.

GANs have also had success in text-to-image translation. Reed et al. [4] utilize a text-conditional convolutional GANs architecture given in Figure 12.1. This architecture takes random noise samples appended with a sentence that has gone through a text encoder as input for each pixel. The input passes through the generator that contains convolutional layers, up-sampling into desired resolution. The discriminator down-samples and determines if real or fake based on the value of the pixel and the encoded text. This model was further improved in *AttnGANs* [5], by incorporating a deep attentional multimodel similarity model (DAMSM), which applies attention layers from transformers and discriminators at several resolution steps. The architecture of *AttnGANs* is given in Figure 12.2. More descriptively, each attention model automatically retrieves the conditions, or most relevant word vectors, for the generating different sub-regions of the images, the DAMSM provides the fine-grained image-text matching loss for the generative network.

Image-to-Image: Image-to-image converts one image into another that is close, but uniquely

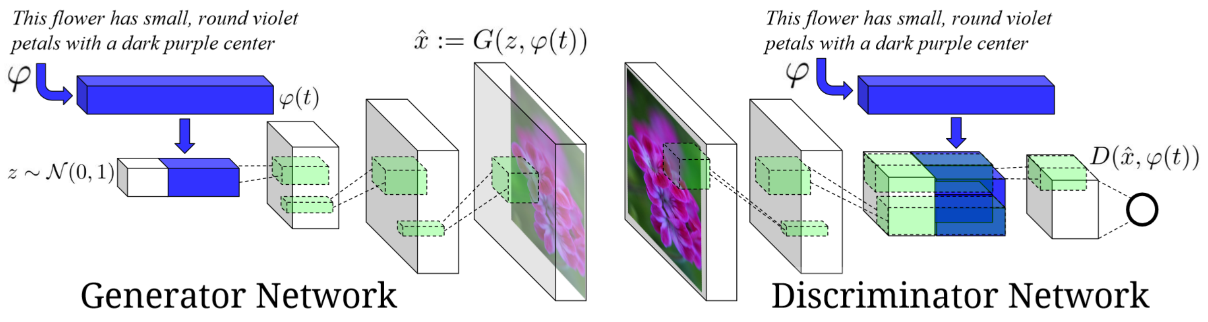

This flower has small, round violet petals with a dark purple center
φ
$\varphi(t)$
$z \sim \mathcal{N}(0,1)$
$\hat{x} := G(z, \varphi(t))$
Generator Network
This flower has small, round violet petals with a dark purple center
φ
$D(\hat{x}, \varphi(t))$
Discriminator Network

Figure 12.1: Text-conditional convolutional GAN architecture. Text encoding $\varphi(t)$ is used by both generator and discriminator. Image from the original work in [4].

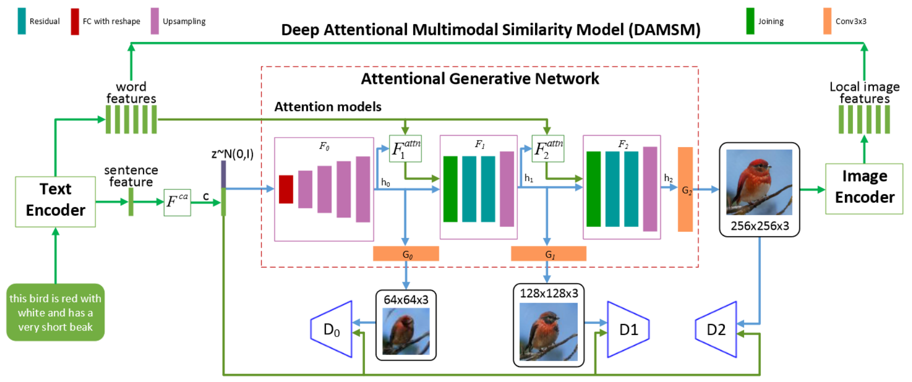


Figure 12.2: The AttnGANs architecture. Image from the original work in [5].

different. Such uses of this process include changing the style of an image (such as realism to Van Gogh), characteristics (person's hair, eye color, or facial shape), or creating realistic images based on sketches.

Some generative models used for image-to-image are variational autoencoders and transformers. *TransGaGa* [60] utilizes both models to apply the geometry and appearance of the image to one another. This is done by first creating two autoencoders, one used to obtain the geometry of the image and another to transform the geometry back to the base image. Next, two transformers are created, one that transforms one part of the geometry of one style of image to the other, and one transformer to convert the appearance (or the encoded output when converting from the geometry to image) to the other style. Using the information from the transformers combined with those from the autoencoders, the model is able to generate images translated in another style. Another model that can be used for image-to-image translation is denoised diffusion [61], which trains to apply noise to input images and then denoise using a trained diffusion model that was trained on denoising images in the targeted style.

GANs are one of the most dominant generative process models for image-to-image translations. Isola et al. *PatchGANs* [23] introduces one of the first GANs models for image-to-image translation. *PatchGANs* utilizes a U-Net architecture [62] and incorporates skip connections, which concatenates the same level channels when down sampling to those when up sampling, with a discriminator architecture that penalizes the structure at the scale of patches. Another GANs model for image-to-image translation is from the first paper on *CycleGANs* [63]. *CycleGANs* was introduced as a method to perform unpaired (no direct one-to-one reference in the training set) image-to-image translation. *CycleGANs* are further explained in Section 2.2.4.1. Another GANs model of note is Karras et al. *StyleGANs* [6]. The main difference between *StyleGANs* and other GANs networks is the architecture of the generator as illustrated in Figure 12.3. *StyleGANs* incorporate a mapping network to apply individual styles at separate individual sections by controlling the generator through adaptive instance normalization at each convolutional layer. The concept of applying styles incrementally is of particular interest for future works as this could be used to implement slight adjustments to generated EM signals to account for probe position and different devices.

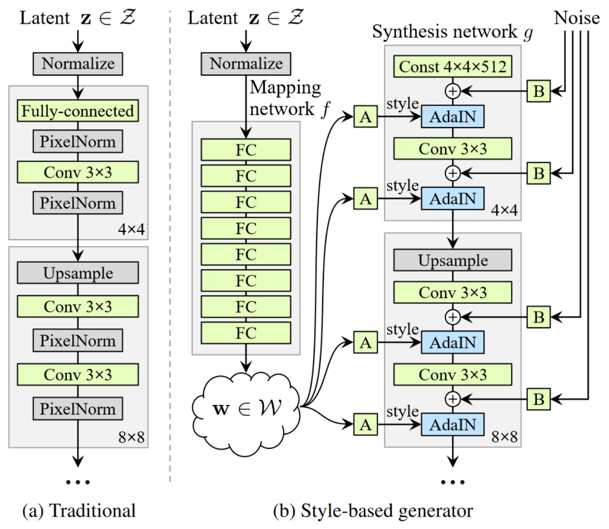


Figure 12.3: Comparison between the generator architecture of traditional GANs (left) and StyleGANs (right). Image from the original work in [6].

12.4 EM Simulators

Few works exist in the field of generating EM signals (EM simulators). Furthermore, to the best of our knowledge, no EM simulator exists that is optimized for the purpose of EM-based anomaly detection. However, three works exist in the area of EM simulation. Two works attempt to generate EM signals based on creating an algorithm comprised of information from the entire physical makeup of the CPU, and one work just considers the bit-flips performed during run-time.

12.4.1 ConvEM:

Kumar et al. [50] presents a computational platform for EM side-channel attack analysis using commercial electronic design automation tools to extract current waveforms and a custom EM simulator (*ConvEM*) to emit them. The authors' main focus is to identify the vulnerabilities of integrated circuits (ICs) to EM side channel attacks aimed to decipher AES encryption keys.

The authors rely on an expensive computational algorithm to simulate EMs at a given point by understanding the physical makeup of the device. The method of ConvEM is based on observations of the EM signal given different interactions on the wires of the chip. Some parameters to estimate the EM signals is the probe positions near the surface of a chip, the density of points on the surface of the interconnect network (wire density), and the direction of the signal from the source port to the observer point. The authors note that, while their algorithms work for their small-scale experiment, a more powerful algorithm needs to be adopted for EM simulations.

Issues with using *ConvEM* for anomaly detection: ConvEM is based on simulating the EM signal through knowledge of all physical components of the circuit board, which is difficult and time-consuming in practice to gather given an entire program. Furthermore, in order to utilize such a model for anomaly detection purposes would require a reevaluation of information for every execution branch and update, making the use of this method infeasible for fingerprinting purposes.

12.4.2 EMSim2023

Ma et al. [64] introduce another EM simulator (*EMSim2023*) developed for the purpose of analyzing possible EM leakage during the early design phase of ICs. The process of generating EM signals is done using an expanded algorithm from *ConvEM*, taking into account sub-regions of wires rather than the whole makeup of the ICs at once.

According to the authors, the logic cell and parasitic network characterize the digital complementary metal oxide-semiconductor-based ICs. The logic cell network is devised of transistors on the silicon substrate providing combinatorial and sequential logic functionality. The logic cell's pins include input/output pins and power-supply pins that transfer the logical signals in and out of the logic cells through interconnected wires. The power-supply pins provide the positive and negative supply voltage for logic cells along with the VDD and GND power grids. The power grid forms a parasitic network when combined with the interconnect wires. The metal wires carrying time-varying current emits the EM signals. Thus, the parasitic network excited by the logic cell network is responsible for the circuit's EMs.

Using this information, the authors generate the EM emanations based on the sub-regions of wires. Information required for the *EMSim2023* model includes the amount, the length and width, the distance from the center to the measured point, and the current density in the wire at a given time for all sub-regions.

Issues with using *EMSim2023* for anomaly detection: While *EMSim2023* may be acceptable for the purposes of identifying EM leakage early in the design phase of an integrated circuit, this method requires many parameters of the device's circuitry and distances. Furthermore, the work is focused on the amount and effects an ICs design has on EM signals rather than what can be derived and used for fingerprinting purposes.

12.4.3 EMSim2020

One mechanism for EM simulation is presented in [65] called *EMSim2020*. In the paper, Sehatbakhs et al. propose a cycle-by-cycle method to synthetically generate EM signals for embedded devices by analyzing the CPU architecture. The cycle-by-cycle was chosen to address pipeline effects, such as micro-architectural events namely stalls, mispredictions, and cache misses. To account for the pipeline effects, the authors choose to analyze and generate signals based on their bit-flips rather than individual instructions.

The authors note that EM side-channel signals are created due to bit-flips at the transistor level. As such, all transistors and metal layer components contribute to the EM signal, but modeling all transistors and on-chip wires are practically infeasible. Instead, the approach focuses on individual instructions and operations to attribute the average behavior of the EM emission, the authors modeled the cycle-by-cycle effect on the processor's hardware. To be more specific, they modeled micro-architectural components as independent sources of the EM emissions and grouped these units in each pipeline stage as an individual sources. This was done mainly because of their findings that each instruction has a different footprint in each cycle and the side-channel generated at each cycle is a combination of these activities in all stages.

The authors' overall process is done by determining the effects of what they call the instruction-dependent actives (caused by activities of micro-architectural units such as the register files) and data-dependent activities (created from bit-flips on the data bus, address-bus, and any other registers). This was done by removing all other sources and analyzing the signal amplitude for individual sources one-by-one. The authors go on to use the identified signals generated by the individual sources and combine them using linear regression to create a simulated EM signal.

Issues with using *EMSim2020* for anomaly detection: The overall purpose of *EMSim2020* was to develop an EM simulator to identify possible emission leakage, rather than anomaly detection which is our goal. Furthermore, this process requires significant manual labor as they must observe, analyze, and model the peculiarities of certain CPU architectures. Additionally, *EMSim2020* is not transferable to other CPU architectures, and as such the method must be redone for every device.

Chapter 13

Conclusion

In conclusion, generating EM signals given the ASM code can significantly reduce reliance on a human expert for capturing signals. As such, the major problem that traditional EM-based anomaly detection methods have with respect to their scalability, can be solved using generative models. In this work, we defined and evaluated a novel framework for scalable anomaly detection. This framework includes a technique for gathering samples from a library of instruction sequence building blocks, a GANs architecture for code-to-EM signal translation, and an anomaly detection method that accounts for multiple baselines. From our experiments, we evaluate several GANs models. The best model found (ResGANs-COD2EM) shows that it can be used for generating EM signals for anomaly detection with near-to-perfect results when generating both the full and peak values. Additionally, we indicated that ResGANs-COD2EM requires minimal processing power, being trained using one GPU while generating high similarity EMs to real captured EM observations of the same binary code.

13.1 Future Work

While we have successfully identified a possible solution to the problem of scalability for traditional EM-based anomaly detection, there are still several challenges to address before a fully EM-based anomaly detection method can be feasibly used in realistic scenarios. Currently, we have several planned works towards this goal.

Large Scale Experimentation: Larger-scale experimentation is planned and the results will be provided in a future research paper. This larger-scale experimentation will be performed on a program that is at least ten times that of *Alpha*, the program we used in our small-scale experiment. Additionally, the generative model will be trained on a library containing a reasonable amount of building blocks corresponding to the common sequences to generate the EM of any program in a targeted device.

Library of the Most Common Sequences: We plan to release the library containing the EM signals of common sequences that we will use in the large-scale experiments to the public. As such, we encourage others in the field to adjust and make their own models in an effort to further advance the feasibility of EM-based anomaly detection.

Denoising Captured EM Signals: Currently, limited work has been provided for denoising EM signals. My Initial effort is presented in [15], which illustrates the use of SVD for denoising EM signals. This method was further improved in [16], where a formula is provided to calculate the optimal parameters for SVD given a signal's signal-to-noise ratio. To further remove the environmental noise, I plan to evaluate the use of NN models such as Denoise Diffussion [2] or variational autoencoders [55] for removing noise on captured EM signals.

Transferability: Transferability is a main subject that hasn't been addressed much in EM-based anomaly detection. Traditional methods do not allow for this and require recapturing based on the probe position, device, and binary code. A method we are looking into that will allow higher transferability of EM-based anomaly detection is the inclusion of further translating the EM signals. In further detail, we plan to readjust or alter the generated or existing EM signals based on the EM-probe location and the device model. If a generative model can account for these factors, then a single model and library can be provided for numerous devices and equipment. This removes the need to train a generative model for each probe position and embedded device. We plan on achieving this by utilizing a similar model to StyleGANs [6], or by including another model after generating the EM signals.

Advanced CPU Events: A subject that we have not addressed, due to the rarity in basic single process embedded devices, is the micro-architectural events. These events include pipeline stall, cache miss, and prediction that can cause a slight skew in the EM signal. However, such events are not abnormal nor malicious. As one of the last planned works, we plan to account for such events that can randomly take place by the CPU.

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[2] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6840–6851, 2020.

[3] Z. Xiao, K. Kreis, and A. Vahdat, "Tackling the generative learning trilemma with denoising diffusion gans," *arXiv preprint arXiv:2112.07804*, 2021.

[4] S. E. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative adversarial text to image synthesis." in *ICML*, ser. JMLR Workshop and Conference Proceedings, M.-F. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 1060–1069.

[5] T. Xu, P. Zhang, Q. Huang, H. Zhang, Z. Gan, X. Huang, and X. He, "Attngan: Fine-grained text to image generation with attentional generative adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 1316–1324.

[6] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4396–4405.

[7] S. D. D. Anton, A. P. Lohfink, and H. D. Schotten, "Discussing the feasibility of acoustic sensors for side channel-aided industrial intrusion detection: An essay," in *Proceedings of the Third Central European Cybersecurity Conference*, 2019, pp. 1–4.

[8] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu, "On code execution tracking via power side-channel," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1019–1031.

[9] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, “Eddie: Em-based detection of deviations in program execution,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 333–346.

[10] N. Sehatbakhsh, M. Alam, A. Nazari, A. Zajic, and M. Prvulovic, “Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devices,” in *2018 IEEE international symposium on hardware oriented security and trust (HOST)*. IEEE, 2018, pp. 1–8.

[11] S. Wei, A. Aysu, M. Orshansky, A. Gerstlauer, and M. Tiwari, “Using power-anomalies to counter evasive micro-architectural attacks in embedded systems,” in *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019, pp. 111–120.

[12] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu, “Watch me, but don’t touch me! contactless control flow monitoring via electromagnetic emanations,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1095–1108.

[13] C. Kolias, D. Barbará, C. Rieger, and J. Ulrich, “Em fingerprints: Towards identifying unauthorized hardware substitutions in the supply chain jungle,” in *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2020, pp. 144–151.

[14] C. Kolias, R. Borrelli, D. Barbara, and A. Stavrou, “Malware detection in critical infrastructures using the electromagnetic emissions of plcs,” *Transactions*, vol. 121, no. 1, pp. 519–522, 2019.

[15] K. Vedros, G. M. Makrakis, C. Kolias, M. Xian, D. Barbara, and C. Rieger, “On the limits of em based detection of control logic injection attacks in noisy environments,” in *2021 Resilience Week (RWS)*. IEEE, 2021, pp. 1–9.

[16] E. Miller, “Detecting code injections in noisy environments through em signal analysis and svd denoising,” Master’s thesis, University of Idaho, 2022. [Online]. Available: https://www.proquest.com/dissertations-theses/detecting-code-injections-noisy-environments/docview/2674042935/se-2

[17] H. A. Khan, M. Alam, A. Zajic, and M. Prvulovic, “Detailed tracking of program control flow using analog side-channel signals: a promise for iot malware detection and a threat for many cryptographic implementations,” in *Cyber Sensing 2018*, vol. 10630. International Society for Optics and Photonics, 2018, p. 1063005.

[18] Microchip, “AVR Instruction Set Manual,” http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf, accessed 2022-09-30.

[19] L. Juvela, B. Bollepalli, J. Yamagishi, and P. Alku, “Waveform generation for text-to-speech synthesis using pitch-synchronous multi-scale generative adversarial networks,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2019, pp. 6915–6919.

[20] N. Li, S. Liu, Y. Liu, S. Zhao, and M. Liu, “Neural speech synthesis with transformer network,” in *AAAI Conference on Artificial Intelligence*, 2019.

[21] S. Frolov, T. Hinz, F. Raue, J. Hees, and A. Dengel, “Adversarial text-to-image synthesis: A review,” *Neural Networks*, vol. 144, pp. 187–209, 2021.

[22] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014.

[23] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1125–1134.

[24] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired image-to-image translation using cycle-consistent adversarial networks,” in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[25] S. Liu, D. Su, and D. Yu, “Diffgan-tts: High-fidelity and efficient text-to-speech with denoising diffusion gans,” *arXiv preprint arXiv:2201.11972*, 2022.

[26] Y. Jiang, S. Chang, and Z. Wang, "Transgan: Two pure transformers can make one strong gan, and that can scale up," *Advances in Neural Information Processing Systems*, vol. 34, pp. 14 745–14 758, 2021.

[27] S. A. Barnett, "Convergence problems with generative adversarial networks (gans)," *arXiv preprint arXiv:1806.11382*, 2018.

[28] Z. Zhang, M. Li, and J. Yu, "On the convergence and mode collapse of gan," in *SIGGRAPH Asia 2018 Technical Briefs*, ser. SA '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3283254.3283282

[29] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[30] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[31] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 639–650.

[32] K. A. Vedros, G. M. Makrakis, C. Kolias, R. C. Ivans, and C. Rieger, "Towards scalable em-based anomaly detection for embedded devices through synthetic fingerprinting," *arXiv preprint arXiv:2302.02324*, 2023.

[33] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, 2011.

[34] D. Barbará, C. Domeniconi, and J. Rogers, "Detecting outliers using transduction and statistical testing," in *Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Philadelphia, 2006, pp. 55–64.

[35] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, “Lof: identifying density-based local outliers,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 93–104.

[36] M. Binkowski, J. Donahue, S. Dieleman, A. Clark, E. Elsen, N. Casagrande, L. C. Cobo, and K. Simonyan, “High fidelity speech synthesis with adversarial networks,” *ArXiv*, vol. abs/1909.11646, 2020.

[37] T. Karras, T. Aila, S. Laine, and J. Lehtinen, “Progressive growing of gans for improved quality, stability, and variation,” *arXiv preprint arXiv:1710.10196*, 2017.

[38] C. Ge, I. Y.-H. Gu, A. S. Jakola, and J. Yang, “Enlarged training dataset by pairwise gans for molecular-based brain tumor classification,” *IEEE access*, vol. 8, pp. 22 560–22 570, 2020.

[39] F. H. K. d. S. Tanaka and C. Aranha, “Data augmentation using gans,” *arXiv preprint arXiv:1904.09135*, 2019.

[40] M. Wang, H. Li, and F. Li, “Generative adversarial network based on resnet for conditional image restoration,” *arXiv preprint arXiv:1707.04881*, 2017.

[41] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[42] D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun, and M. Paluri, “A closer look at spatiotemporal convolutions for action recognition,” in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2018, pp. 6450–6459.

[43] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Investigations of power analysis attacks on smartcards.” *Smartcard*, vol. 99, pp. 151–161, 1999.

[44] G. de Souza Faria and H. Y. Kim, “Differential audio analysis: a new side-channel attack on pin pads,” *International Journal of Information Security*, vol. 18, pp. 73–84, 2019.

[45] T.-H. Le, C. Canovas, and J. Clédiere, “An overview of side channel analysis attacks,” in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, 2008, pp. 33–43.

[46] N. Boggs, J. C. Chau, and A. Cui, "Utilizing electromagnetic emanations for out-of-band detection of unknown attack code in a programmable logic controller," in *Cyber Sensing 2018*, vol. 10630. International Society for Optics and Photonics, 2018, p. 106300D.

[47] R. A. Riley, J. T. Graham, R. M. Fuller, R. O. Baldwin, and A. Fisher, "A new way to detect cyberattacks: extracting changes in register values from radio-frequency side channels," *IEEE Signal Processing Magazine*, vol. 36, no. 2, pp. 49–58, 2019.

[48] M. A. Islam, S. Ren, and A. Wierman, "Exploiting a thermal side channel for power attacks in multi-tenant data centers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1079–1094.

[49] A. Sayakkara, N.-A. Le-Khac, and M. Scanlon, "Leveraging electromagnetic side-channel analysis for the investigation of iot devices," *Digital Investigation*, vol. 29, pp. S94–S103, 2019.

[50] A. Kumar, C. Scarborough, A. Yilmaz, and M. Orshansky, "Efficient simulation of em side-channel attack resilience," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 123–130.

[51] A. Moradi and T. Schneider, "Improved side-channel analysis attacks on xilinx bitstream encryption of 5, 6, and 7 series," in *Constructive Side-Channel Analysis and Secure Design: 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*. Springer, 2016, pp. 71–87.

[52] D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi, "The em side-channel(s):attacks and assessment methodologies," 12 2008.

[53] H. A. Khan, N. Sehatbakhsh, L. N. Nguyen, R. L. Callan, A. Yeredor, M. Prvulovic, and A. Zajić, "Idea: Intrusion detection through electromagnetic-signal analysis for critical embedded and cyber-physical systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 3, pp. 1150–1163, 2019.

[54] D. Rezende and S. Mohamed, "Variational inference with normalizing flows," in *International conference on machine learning*. PMLR, 2015, pp. 1530–1538.

[55] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” *arXiv preprint arXiv:1312.6114*, 2013.

[56] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “Wavenet: A generative model for raw audio,” *arXiv preprint arXiv:1609.03499*, 2016.

[57] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan *et al.*, “Natural tts synthesis by conditioning wavenet on mel spectrogram predictions,” in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2018, pp. 4779–4783.

[58] E. Mansimov, E. Parisotto, J. L. Ba, and R. Salakhutdinov, “Generating images from captions with attention,” *arXiv preprint arXiv:1511.02793*, 2015.

[59] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, “Zero-shot text-to-image generation,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 8821–8831.

[60] W. Wu, K. Cao, C. Li, C. Qian, and C. C. Loy, “Transgaga: Geometry-aware unsupervised image-to-image translation,” in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 8012–8021.

[61] H. Sasaki, C. G. Willcocks, and T. P. Breckon, “Unit-ddpm: Unpaired image translation with denoising diffusion probabilistic models,” *arXiv preprint arXiv:2104.05358*, 2021.

[62] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*. Springer, 2015, pp. 234–241.

[63] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired image-to-image translation using cycle-consistent adversarial networks,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2242–2251.

[64] H. Ma, M. Panoff, J. He, Y. Zhao, and Y. Jin, “Emsim: A fast layout level electromagnetic emanation simulation framework for high accuracy pre-silicon verification,” *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1365–1379, 2023.

[65] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic, “Emsim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 71–85.