FastSpec: Scalable Generation and Detection of Spectre Gadgets Using Neural Embeddings

M. Caner Tol, Koray Yurtseven, Berk Gulmezoglu, and Berk Sunar

Worcester Polytechnic Institute, Worcester, MA, USA

Abstract

Several techniques have been proposed to detect vulnerable Spectre gadgets in widely deployed commercial software. Unfortunately, detection techniques proposed so far rely on hand-written rules which fall short in covering subtle variations of known Spectre gadgets as well as demand a huge amount of time to analyze each conditional branch in software. Since it requires arduous effort to craft new gadgets manually, the evaluations of detection mechanisms are based only on a handful of these gadgets.

In this work, we employ deep learning techniques for automated generation and detection of Spectre gadgets. We first create a diverse set of Spectre-V1 gadgets by introducing perturbations to the known gadgets. Using mutational fuzzing, we produce a data set with more than 1 million Spectre-V1 gadgets which is the largest Spectre gadget data set built to date. Next, we conduct the first empirical usability study of Generative Adversarial Networks (GANs) for creating assembly code without any human interaction. We introduce SpectreGAN which leverages masking implementation of GANs for both learning the gadget structures and generating new gadgets. This provides the first scalable solution to extend the variety of Spectre gadgets.

Finally, we propose FastSpec which builds a classifier with the generated Spectre gadgets based on the novel high dimensional Neural Embedding technique BERT. For case studies, we demonstrate that FastSpec discovers potential gadgets in OpenSSL libraries and Phoronix benchmarks. Further, FastSpec offers much greater flexibility and much faster classification compared to what is offered by the existing tools. Therefore FastSpec can be used for gadget detection in large-scale projects.

1 Introduction

A new era of microarchitectural attacks began with newly discovered Spectre [26] and Meltdown [32] attacks which may be exploited to exfiltrate confidential information through subtle channels during speculative and out-of-order executions. Spectre attacks target vulnerable code patterns called gadgets, which leak information during speculatively executed instructions. While the initial variants of Spectre [26] exploit conditional and indirect branches, Koruyeh et al. [27] proposes another Spectre variant obtained by poisoning the entries in Return-Stack-Buffers (RSBs). Moreover, new Spectre-type attacks [7, 27] are implemented against the SGX environment and even remotely over the network [54]. These attacks show the applicability of Spectre attacks in the wild.

Unfortunately, chip vendors try to patch the leakages one-by-one with microcode updates rather than fixing the flaws by changing their hardware designs. Therefore, developers rely on automated malware analysis tools to eliminate mistakenly placed Spectre gadgets in their programs. The proposed detection tools mostly implement taint analysis [66] and symbolic execution [15,65] to identify potential gadgets in benign applications. However, the methods proposed so far are have two shortcomings: (1) the scarcity of Spectre gadgets prevents the comprehensive evaluation of the tools, (2) the scanning time increases drastically with increasing binary file sizes. Thus, there is a need for a robust and fast analysis tool that can automatically discover potential Spectre gadgets in large-scale commercial software.

Natural Language Processing (NLP) techniques are applied to automate challenging tasks in natural language and text processing [48]. Later, NLP techniques have been applied for security as well, such as in network traffic [49] and vulnerability analysis [50]. Such applications leverage word [39] or paragraph [30] embedding techniques to learn the vector representations of the text. The success of these techniques heavily depend on the large data sets which ease to train scalable and robust NLP models. However, for Spectre, for instance, the number of available gadgets is only 15, which makes it crucial to create new Spectre gadgets before building an NLP-based detection tool.

Generative Adversarial Networks (GANs) [14] are a type of generative models, which aim to produce new examples by learning the distribution of training instances in an adversarial setting. Since adversarial learning makes GANs more robust

and applicable in real-world scenarios, GANs have become quite popular in recent years with applications ranging from generating images [42, 68] to text-to-image translation [51] etc. While the early applications of GANs focused on computer vision, implementing the same techniques in Natural Language Processing (NLP) tasks poses a challenge due to the lack of continuous space in text. To overcome this obstacle, various mathematical GAN-based techniques have been proposed to achieve better success in generating human-like sentences [12, 16]. However, it is still unclear whether GANs can be used to create application-specific code snippets for use in the context of computer security. Additionally, each computer language has a different structure, semantics, and other features that make it more difficult to generate meaningful snippets for a specific application.

Neural vector embeddings [30, 39] which are used to obtain the vector representations of words, have proven extremely useful in NLP applications. Such embedding techniques also enable one to perform vector operations in high dimensional space while preserving the meaningful relations between similar words. Typically, supervised techniques apply word embedding tools as an initial step to obtain the vector embedding of each token, and then, build a supervised model on top. For instance, BERT [9] was proposed by the Google AI team to learn the relations between different words in a sentence by applying a self-attention mechanism [63]. BERT has exhibited superior performance compared to previous techniques [38, 57] when combined with bi-directional learning. Furthermore, the attention mechanism improves GPU utilization while learning long sequences more efficiently. Recently, BERT-like architectures are shown to be capable of modeling high-level programming languages [13, 29]. However, it is still unclear whether it will be effective to model a low-level programming language, such as Assembly language, and help in building more robust malware detection tools which is the goal of this paper.

Our Contributions. Our contributions consist of two parts. First, we focus on increasing the number and diversity of Spectre gadgets with mutational fuzzing. We start with 15 examples [25] and produce 1 million gadgets by introducing various perturbations to the existing gadgets. Then, we propose SpectreGAN, which learns the distribution of 1 million Spectre gadgets to generate new gadgets with higher accuracy. The generated gadgets are evaluated in terms of both semantic and microarchitectural aspects to verify their diversity. The gadgets that are not detected by detection tools are introduced as novel gadgets.

In the second part, we introduce FastSpec which is a high dimensional neural embedding derived from BERT, and use the embedding to obtain a highly accurate and fast classifier for Spectre gadgets. We train FastSpec with generated gadgets and achieve 0.99 F-1 score in the test phase. Further, we apply FastSpec on the OpenSSL libraries and the Phoronix benchmarks to show that FastSpec is capable of detecting the hidden gadgets while significantly decreasing the analysis time compared to *oo7* and *Spectector* in large-scale software.

In summary,

- We extend 15 base Spectre examples to 1 million gadgets via mutational fuzzing,
- We propose SpectreGAN which leverages conditional GANs to create new Spectre gadgets by learning the distribution of existing Spectre gadgets in a scalable way,
- We show that both mutational fuzzing and SpectreGAN create diverse and novel gadgets which are not detected by *oo7* and *Spectector* tools,
- We introduce FastSpec which is based on supervised neural word embeddings to identify the potential gadgets in benign applications orders of magnitude faster than rule-based methods.

Outline The paper is organized as follows: First, the background on transient execution attacks and NLP are given in Section 2. Then, the related work is given in Section 3. Next, we introduce both fuzzing-based and SpectreGAN generation techniques in Section 4. A new Transformer-based detection tool namely, FastSpec is proposed in Section 5. Finally, we conclude the paper with discussions in Section 6 and conclusion in Section 7.

2 Background

2.1 Transient Execution Attacks

In order to keep the pipeline occupied at all times, modern CPUs have sophisticated microarchitectural optimizations to predict the control flow and data dependencies, where some instructions can be executed ahead of time in the transient domain. However, the control-flow predictions are not 100% accurate, which might cause to execute some instructions wrongly. These instructions cause pipeline flush once they are detected and their results are never committed. Interestingly, microarchitectural optimizations make it possible to leak secrets. The critical time period before the flush happens is commonly referred to the transient domain.

There are two classes of attacks in the transient domain [5]. The first one is called Meltdown-type attacks [4, 32, 52, 56, 61, 62] which exploit delayed permission checks and lazy pipeline flush in the re-order buffer. The other class is Spectre-type attacks [19, 24, 26, 27, 34] that exploit the speculative execution. As most Meltdown-type attacks are fixed in most microarchtiectures and the Spectre-type attacks are still applicable to a wide range of targets [5], i.e. Intel, AMD, and ARM CPUs, we will focus our attention to Spectre-V1 attacks.

Some researchers proposed new designs requiring change in the silicon level [23, 28, 70] while others proposed software solutions to mitigate transient execution attacks [45, 60]. Although these mitigations are effective against Spectre-type

attacks, majority of them are not being used because either they degrade the performance drastically [6] or the current hardware has no support. Hence, Spectre-type attacks are not completely resolved yet and finding an efficient countermeasure is still an open problem.

2.1.1 Spectre

As a typical software consists of branches and instruction/data dependencies, modern CPUs have components for predicting the outcome of conditional branches to execute the instructions speculatively. These components are called branch prediction units (BPU) which use a history table and other components to make predictions on branch outcomes.

```
1 void victim_function(size_t x){
2   if(x < size)
3     temp &= array2[array1[x] * 512];
4 }
```

Listing 1: Spectre-V1 C Code

In Spectre attacks, a user fills the history table with malicious entries such that the BPU makes a misprediction. Then, the CPU executes a set of instructions speculatively. As a result of misprediction, the sensitive data can be leaked through microarchitectural components for instance by encoding the secret to the cache lines to establish a covert channel. For example, in the Spectre gadget in Listing 1, the 2^{nd} line checks whether the user input `x` is in the bound of `array1`. In a normal execution environment, if the condition is satisfied, the program retrieves x^{th} element of `array1` and a multiple of the retrieved value (512) is used as an index to access the data in `array2`. However, under some conditions, the `size` variable might not be present in the cache. In such occurrences, instead of waiting for `size` to be available, the CPU executes the next instructions speculatively to eliminate unnecessary stalls in the pipeline. When `size` becomes available, the CPU checks whether it made a correct prediction or not. If the prediction was wrong, the CPU rolls back and executes the correct path. Although the results of speculatively executed instructions are not observable in architectural components, the access to the `array2` leaves a footprint in the cache, which makes it possible to leak the data through side-channel analysis.

There are a number of known Spectre variants: Spectre-V1 (bounds check bypass), Spectre-V2 (branch target injection), Spectre-RSB [27, 34] (return stack buffer speculation), and Spectre-V4 [19] (speculative store bypass). We focus on Spectre-V1 as our primary goal since there is no effective solution other than inserting fence instructions after conditional branches. On the other hand, Spectre-V2 is mitigated by flushing BTB across context switches. Moreover, unlike Spectre-V2 gadgets, to stop exploit in Spectre-V1, the vulnerable code segment needs to be found. Thus, a detection tool for finding gadgets is needed.

2.2 Natural Language Processing

2.2.1 seq2seq Architecture

A major drawback of DNNs is the lack of mapping sequences to sequences. To solve this problem, a new approach called seq2seq [57] was introduced to learn sequence-to-sequence relations. The seq2seq architecture consists of encoder and decoder units. Both units leverage multi-layer Long Short Term Memory (LSTM) structures where the encoder produces a fixed dimension encoder vector. The encoder vector represents the information learned from the input sequence. Then, the decoder unit is fed with the encoder vector to predict the mapping sequence of the input sequence. After the end of the sequence token is produced by the decoder, the prediction phase stops. The seq2seq structure is commonly used in chatbot engine [46] since sequences with different lengths can be mapped to each other.

2.2.2 Generative Adversarial Networks

A specialized method of training generative models was proposed by Goodfellow et al. [14] which is called generative adversarial networks (GANs). The generative models are trained with a separate discriminator model under an adversarial setting. In [14], the training of generative model is defined as,

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[log\ D(x)] + \mathbb{E}_{z\sim p_z(z)}[log(1-D(G(z)))]. \quad (1)$$

In Equation 1, the generator G and the discriminator D are trained in such a way that D, as a regular binary classifier, tries to maximize its confidence $D(x)$ on real data x, while minimizing $D(G(z))$ on generated samples by the G. At the same time, G tries to maximize the confidence of discriminator $D(G(z))$ on generated samples $G(z)$ and minimize $D(x)$ where x is the real data.

In previous works, GANs are commonly used for learning the probability distribution of a data set p_{data} and generating new data samples from the same distribution on a continuous space, *e.g.*, images [1, 20, 43, 47], audio signals [11].

2.2.3 MaskGAN

MaskGAN [12] is a type of conditional GAN technique to establish a good performance out of traditional GANs. MaskGAN is based on seq2seq architecture with an attention mechanism. Attention mechanism improves the performance of the fixed-length encoder vector since LSTMs cause information loss in long sequences. Each time a prediction is made by the decoder unit, a part of the input sequence is used instead of the encoder vector. Hence, each token in the input sequence has a different weight on the decoder output. The main difference of MaskGAN from other GAN-based

text generation techniques is the token masking approach which helps learning the missing texts in a sequence. For this purpose, some tokens are masked that are conditioned on the surrounding context. This technique increases the chance of generating longer and more meaningful sequences out of GANs.

2.2.4 Transformer and BERT

Although recurrent models with attention mechanisms learn the representations of long sequences, attention-only models, namely *Transformer* architectures [63], are shown to be highly effective in terms of computational complexity and performance on long-range dependencies. Similar to *seq2seq* architecture, *Transformer* architecture consists of encoder-decoder model. The main difference of *Transformer* is that recurrent models are not used in encoder or decoder units. Instead, the encoder unit is composed of L hidden layers where each layer has a multi-head self-attention mechanism with A attention heads and a fully connected feed-forward network. The input embedding vectors are fed into the multi-head attention and the output of the encoder stack is formed by feed-forward network which takes the output of the attention sub-layer. The decoder unit also has L hidden layers, and it has the same sub-layers with encoder. In addition to one multi-head attention unit and one feed-forward network, the decoder unit has an extra multi-head attention layer that processes the encoder stack output. In order to process the information in the sequence order, positional embeddings are used with token embeddings where both embedding vectors have a size of H.

Keeping the same Transformer architecture, Devlin et al. [9] introduced a new language representation model called BERT (Bidirectional Encoder Representations from Transformers) which surpasses the state-of-the-art scores on language representation learning. BERT is designed to pre-train the token representation vectors of deep bidirectional Transformers. For the detailed description of the architecture, we refer the readers to [9, 63]. The heavy part of the training is handled by processing unlabeled data in an unsupervised manner. The unsupervised phase is called *pre-training* which consists of masked language model training and next sentence prediction procedures. The supervised phase is referred to as *fine-tuning* where the model representations are further trained with labeled data for a text classification task. Both phases are further explained in detail for Spectre gadget detection model in Section 5.

3 Related Work

3.1 Spectre attacks and detectors

Spectre Variations and Covert Channels In the first Spectre study [26], two variants were introduced. While Spectre-V1 exploits the conditional branch prediction mechanism when a bound check is present, Spectre-V2 manipulates the indirect branch predictions to leak the secret. Next, researchers discovered new variants of Spectre-based attacks. For instance, a variant of Spectre focuses on poisoning Return-Stack-Buffer (RSB) entries with the desired malicious return addresses [27, 34]. Another variant of Spectre called "Speculative Store Bypass" [19] takes the advantage of memory disambiguator's prediction to create leakage. Traditionally, secrets are leaked through cache timing differences. Then, researchers showed that there are also other covert channels to measure the time difference: namely using network latency [54], port contention [3], or control flow hijack attack based on return-oriented programming [36] to leak secret data.

Defenses against Spectre There are various detection methods for speculative execution attacks. Taint analysis is used in *oo7* [66] software tool to detect leakages. As an alternative way, the taint analysis is implemented in the hardware context to stop the speculative execution for secret dependent data [53, 71]. The second method relies on symbolic execution analysis. Spectector [15] symbolically executes the programs where the conditional branches are treated as mispredicted. Furthermore, SpecuSym [18] and KleeSpectre [65] aim to model cache usage with symbolic execution to detect speculative interference which is based on Klee symbolic execution engine. Following a different approach, Speculator [35] collects performance counter values to detect mispredicted branches and speculative execution domain. Finally, Specfuzz [44] uses a fuzzing strategy to analyze the control flow paths which are most likely vulnerable against speculative execution attacks.

3.2 Binary Analysis with Embedding

Binary analysis is one of the methods to analyze the security of a program. The analysis can be performed dynamically [40] by observing the binary code running in the system. Alternatively, the binary can also be analyzed statically [55]. NLP techniques have been applied to binary analysis in recent years. Mostly, the studies leverage the aforementioned techniques to embed Assembly instructions and registers into a vector space. The most common usage of NLP in binary analysis is to find the similarities between files. Asm2Vec [10] leverages a modified version of PV-DM model to solve the obfuscation and optimization issues in a clone search. Zuo et al. [73] and Redmond et al. [50] solve the binary similarity problem by NLP techniques when the same file is compiled in different architectures. SAFE [37] proposes a combination of skip-gram and RNN self-attention model to learn the embeddings of the functions from binary files to find the similarities.

3.3 GAN-based Text Generation

The first applications of GANs were mostly applied to computer vision to create new images such as human faces [21,22], photo blending [69], video generation [64], and so on. However, text generation is a more challenging task since it is more difficult to evaluate the performance of the outputs. An application [31] of GANs is in the dialogue generation, where adversarial learning and reinforcement are applied together. SeqGAN [72] introduces gradient policy update with Monte Carlo search. LeakGAN [17] implements a modified policy gradient method to increase the usage of word-based features in the adversarial learning. RelGAN [41] applies Gumbel-Softmax relaxation for training GANs as an alternative method to gradient policy update. SentiGAN [67] proposes multiple generators to focus on several sentiment labels with one multi-class generator. However, to the best of our knowledge, the literature lacks GANs which are applied to the Assembly code generation. To fill this literature gap, we propose SpectreGAN in Section 4.2.

4 Gadget Generation

We propose both mutational fuzzing and GAN-based gadget generation techniques to create novel and diverse gadgets. In the following sections, details of both techniques and the diversity analysis of the gadgets are given:

4.1 Gadget Generation via Fuzzing

We begin with fuzzing techniques to extend the base gadgets to create a large data set consists of a million Spectre gadgets in four steps.

Algorithm 1: Gadget generation using mutational fuzzing

Input: An Assembly function A, a set of instructions $\mathbb{I}_b$ and sets of registers $\mathbb{R}_b$ for different sizes of b
Output: A mutated Assembly function A'

$\mathbb{G} := \mathbb{R}_b \mapsto \mathbb{I}_b$
$A' = A$
$MaxOffset = length(A)$
for *1:Diversity* **do**
  for *Offset=1:MaxOffset* **do**
    for *1:Offset* **do**
      $i_b \leftarrow random(\mathbb{I})$
      $r_b \leftarrow random(\mathbb{R}_b|\mathbb{G})$
      $l \leftarrow random(0 : length(A'))$
      $Insert(\{i_b|r_b\}, A', l)$
    end
    Test boundary check(A')
    Test Spectre leakage(A')
  end
end

- **Step 1: Initial Data Set** There are 15 Spectre-V1 gadgets written in C by Kocher [25] and 2 modified examples introduced by *Spectector* [15]. For each example, a separate attacker code is written to leak the whole secret data completely in a reasonable time.

- **Step 2: Compiler variants and optimization levels** Since our target data set is in the Assembly code format, each Spectre gadget written in C is compiled into x86 Assembly code functions by using different compilers. We compiled each example with *gcc*, *clang* and *icc* compilers using *-o0* and *-o2* optimization flags. Therefore, we obtain 6 different Assembly functions from each C function with AT&T syntax.

- **Step 3: Mutational fuzzing based generation** We generated new samples with an approach inspired from mutation-based fuzzing technique [58] as introduced in Algorithm 1. Our mutation operator is the insertion of random Assembly instructions with random operands. For an Assembly function A with length L, we create a mutated assembly function A'. We set a limit on the number of generated samples per assembly function A for each *Offset* value, which is denoted as *Diversity*. We choose a random instruction i_b from the instruction set $\mathbb{I}$ and depending on the instruction format of i_b, we choose random operands r_b which are compatible with the instruction in terms of bit size, b. After proper instruction-operand selection, we choose a random position l in A' and insert $\{i_b|r_b\}$ into that location. We repeat the insertion process until we reach the *Offset* value. The randomly inserted instruction and register list is given in Appendix, Table 3.

- **Step 4: Verification of generated gadgets** Finally, A' is tested whether it still has the array boundary-check for a given user input and still leaks the secret through speculative execution or not. Since a random instruction is inserted in a random location, it is likely to introduce an instruction which alters the flags whose value is checked by the original conditional jump. Thus, the secret may be leaked as the flags are broken. To overcome broken-flag issue, we perform boundary-check test by giving out-of-bound values to A'. If the gadget still leaks the secret, we exclude the candidate gadget from our data set since the gadget is no longer a Spectre-V1 gadget.

At the end of fuzzing-based generation, we obtained a data set of almost 1.1 million Spectre gadgets[1]. The overall success rate of fuzzing technique is around 5% out of compiled gadgets. The generated gadgets are used to train SpectreGAN in the next section.

[1]The entire data set and fuzzing/SpectreGAN generation codes will be available at https://github.com/vernamlab/FastSpec

4.2 SpectreGAN: Assembly Code Generation with GANs

We introduce SpectreGAN which learns the fuzzing generated gadgets in an unsupervised way and generates new Spectre-V1 variants from existing examples in assembly language. The purpose of SpectreGAN is to develop an intelligent way of creating assembly functions instead of randomly inserting instructions and operands. Hence, the low gadget generation rate of fuzzing technique can be improved further with GANs.

We build SpectreGAN based on the MaskGAN model with 1.1 million examples generated in Section 4. Since MaskGAN is originally designed for text generation, we modify the MaskGAN architecture to train SpectreGAN on Assembly language. Finally, we evaluate the performance of SpectreGAN and discuss challenges in assembly code generation.

4.2.1 SpectreGAN Architecture

As all the GAN frameworks, SpectreGAN has a generator model which learns and generates x86 Assembly functions, and a discriminator model which gives feedback to the generator model by classifying the generated samples as real or fake as depicted in Figure 1.

Generator The generator model consists of encoder-decoder architecture (seq2seq) [57] which is composed of two-layer stacked LSTM units. Firstly, the input assembly functions are converted to a sequence of tokens $T' = \{x'_1, ..., x'_N\}$ where each token represents an instruction, register, parenthesis, comma, intermediate value or label. SpectreGAN is conditionally trained with each sequence of tokens where a masking vector $m = (m_1, ..., m_N)$ with elements $m_t \in \{0, 1\}$ is generated. The masking rate of m is determined as $r_m = \frac{1}{N}\sum_{t=1}^{N} m_t$. $m(T')$ is the modified sequence where x'_t is replaced with `<MASK>` token for the corresponding positions of $m_t = 1$. Both T' and $m(T')$ are converted into the lists of vectors $T = \{x_1, ..., x_N\}$ and $m(T)$ by a lookup in a randomly initialized embedding matrix of size $V \times H$, where V and H are the vocabulary size and embedding vector dimension, respectively. In order to learn the masked tokens, T and $m(T)$ are fed into the encoder LSTM units of the generator model. Each encoder unit outputs a hidden state $\overline{h}_s$ which is also given as an input to the next encoder unit. The last encoder unit (e^6_G in Figure 1) produces the final hidden state which encapsulates the information learned from all assembly tokens.

The decoder state is initialized with the final hidden state of the encoder and the decoder LSTM units are fed with $m(T)$ at each iteration. To calculate the hidden state $\widetilde{h}_t$ of each decoder unit, the attention mechanism output and the current state of the decoder h_t are combined. The attention mechanism reduces the information bottleneck between encoder and decoder and eases the training [2] on long token sequences in assembly function data set. The attention mechanism is implemented exactly same for both generator and discriminator model which is illustrated in the discriminator part in Figure 1. The alignment score vector a_t is calculated as:

$$a_t(s) = \frac{e^{h_t^\top \overline{h}_s}}{\sum_{s'=1}^{N} e^{h_t^\top \overline{h}_{s'}}}, \tag{2}$$

where a_t describes the weights of $\overline{h}_s$, for a token x'_t at time step t, where $h_t^\top \overline{h}_s$ is the score value between the token x'_t and T'. This forces decoder to consider the relation between each instruction, register, label and other tokens before generating a new token. The context vector c_t is calculated as the weighted sum of $\overline{h}_s$ as follows:

$$c_t = \sum_{s'=1}^{N} a_t(s)\overline{h}_{s'}. \tag{3}$$

For a context vector, c_t, the final attention-based hidden state, $\widetilde{h}_t$, is obtained by a fully connected layer with hyperbolic tangent activation function,

$$\widetilde{h}_t = tanh(W_c[c_t; h_t]), \tag{4}$$

where $[c_t; h_t]$ is the concatenation of c_t and h_t with the trainable weights W_c. The output list of tokens $\widetilde{T} = (\widetilde{x}_1, ..., \widetilde{x}_N)$ is then generated by filling the masked positions for $m(T')$ where $m_t = 1$. The probability distribution $p(y_t|y_{1:t-1}, x_t)$ is calculated as,

$$p(y_t|y_{1:t-1}, x_t) = \frac{e^{W_s \widetilde{h}_t}}{\sum e^{W_s \widetilde{h}_t}}, \tag{5}$$

where y_t is the output token and attention-based hidden state $\widetilde{h}_t$ is fed into the softmax layer which is represented by the green boxes in Figure 1. It is important to note that the softmax layer is modified to introduce a randomness at the output of the decoder by a sampling operation. The predicted token is selected based on the probability distribution of vocabulary, *i.e.* if a token has a probability of 0.3, it will be selected with a 30% chance. This prevents the selection of the token with the highest probability every time. Hence, at each run the predicted token would be different which increases the diversity in the generated gadgets.

Discriminator The discriminator model has a very similar architecture with the generator model. The encoder and decoder units in the discriminator model are again two-layer stacked LSTM units. The embedding vectors $m(T)$ of tokens $m(T')$, where we replace x'_t with `<MASK>` when $m_t = 1$, are fed into the encoder. The hidden vector encodings $\overline{h}_s$ and the final state of the encoder are given to the decoder.

The LSTM units in the decoder are initialized with the final hidden state of the encoder and $\overline{h}_s$ is given to the attention

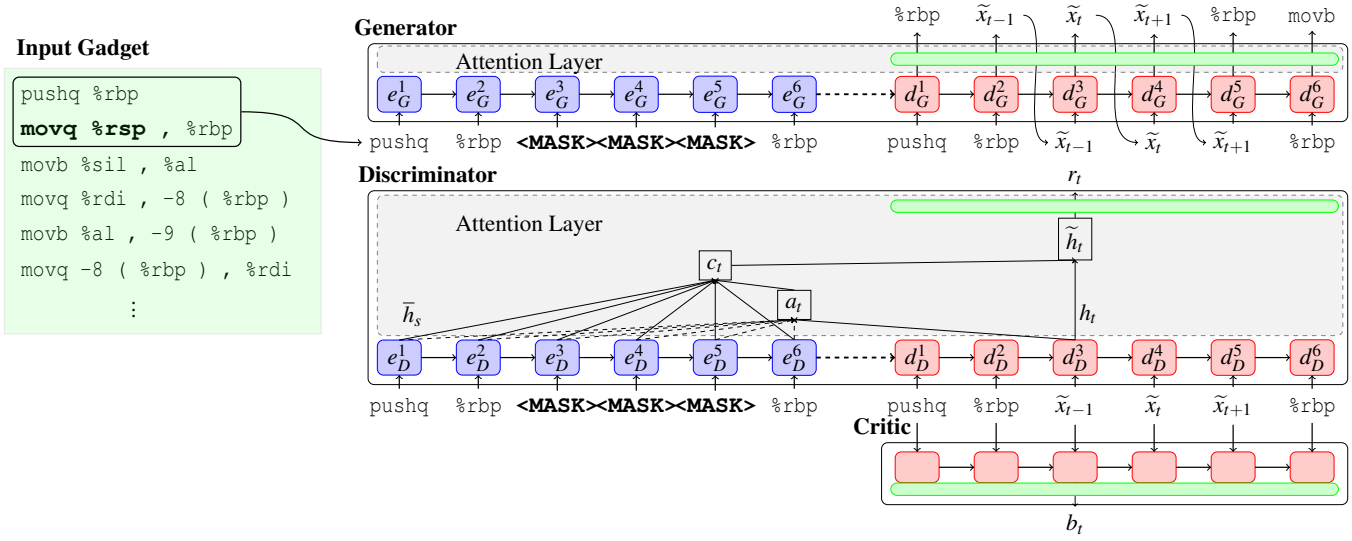

Figure 1: SpectreGAN architecture. Blue and red boxes represent the encoder and decoder LSTM units respectively. Green boxes represent the softmax layers. The listed assembly function (AT&T format) on the left is fed to the models after the tokenization process. The critic model and the decoder part of the discriminator get the same sequence of instructions in the adversarial training.

layer. The list of tokens $\widetilde{T}$ which represents the generated assembly function by the generator model is fed into the decoder LSTM unit with *teacher forcing*. The previous calculations for $a_t(s)$, c_t and $\widetilde{h}_t$ stated in Equation 2, 3, and 4 are valid for the attention layer in the discriminator model as well. The attention-based state value $\widetilde{h}_t$ is fed through the softmax layer which outputs only one value at each time step t,

$$p_D(\widetilde{x}_t = x_t^{real}|\widetilde{T}) = \frac{e^{W_s \widetilde{h}_t}}{\sum e^{W_s \widetilde{h}_t}}, \tag{6}$$

which is the probability of being a real target token x_t^{real}.

SpectreGAN has one more model apart from the generator and the discriminator models, which is called critic model and it has only one two-layer stacked LSTM unit. The critic model is initialized with zero states and gets the same input $\widetilde{T}$ with the decoder. The output of LSTM unit at each time step t is given to the softmax layer and we obtain

$$p_C(\widetilde{x}_t = x_t^{real}|\widetilde{T}) = \frac{e^{W_b h_t}}{\sum e^{W_b h_t}}, \tag{7}$$

which is an estimated version of p_D. The purpose of introducing a critic model for probability estimation will be explained in Section 4.2.2.

4.2.2 Training

The training procedure consists of two main phases namely, pre-training and adversarial training.

Pre-training phase The generator model is first trained with maximum likelihood estimation. The real token sequence T' and masked version $m(T')$ are fed into the encoder of the generator model. In the pre-training, only the real token sequence T' is fed into the decoder using *teacher forcing*. The training maximizes the log-probability of generated tokens, $\widetilde{x}_t$ given the real tokens, x'_t, where $m_t = 1$. Therefore, the pre-training objective is

$$\frac{1}{N}\sum_{t=1}^{N} \log p(m(\widetilde{x}_t)|m(x'_t)), \tag{8}$$

where $p(m(\widetilde{x}_t)|m(x'_t))$ is calculated only for the masked positions. The masked pre-training objective ensures that the model is trained for a *Cloze* task [59].

Adversarial training phase The second phase is the adversarial training where the generator and the discriminator are trained with the GAN framework. Since the generator model has a sampling operation from the probability distribution stated in Equation 5, the overall GAN framework is not differentiable. We utilize the policy gradients to train the generator model as described in the previous works [12, 72].

The reward r_t for a generated token $\widetilde{x}_t$ is calculated as the logarithm of $p_D(\widetilde{x}_t = x_t^{real}|\widetilde{T})$. The aim of the generator model is to maximize the total discounted rewards $R_t = m(\sum_{s=t}^{N} \gamma^s r_s)$ for the fake samples, where γ is the discount factor. Therefore, for each token, the generator is updated with the gradient in Equation 9 using the REINFORCE algorithm, where

$b_t = \log p_C(\widetilde{x}_t = x_t^{real}|\widetilde{T})$ is the baseline rewards by the critic model. Subtracting b_t from R_t helps reducing the variance of the gradient [12].

$$\nabla_\theta \mathbb{E}_G[R_t] = (R_t - b_t)\nabla_\theta \log G_\theta(\tilde{x}_t) \tag{9}$$

To train the discriminator model, both real sequence T and fake sequence $\widetilde{T}$ are fed into the discriminator. Then, the model parameters are updated such that $\log p_D(\widetilde{x}_t = x_t^{real}|\widetilde{T})$ is minimized and $\log p_D(x_t = x_t^{real}|T)$ is maximized using maximum log-likelihood estimation.

4.2.3 Tokenization and Training Parameters

Firstly, we pre-process the fuzzing generated data set in order to convert the assembly functions into sequences of tokens, $T' = (x'_1, ..., x'_N)$. We keep commas, parenthesis, immediate values, labels, instruction and register names as separate tokens. To decrease the complexity, we reduce the vocabulary size of the tokens and simplify the labels in each function so that the total number of different labels is minimum. The tokenization process converts the instruction `"movq (%rax), %rdx"` into the list `["movq", "(", "%rax", ")", ",", "%rdx"]` where each element of the list is a token x'_t. Hence, each token list $T' = \{x'_1, ..., x'_N\}$ represents an assembly function in the data set.

Masking vector has two different roles in the training. While a random masking vector $m = (m_1, ..., m_N)$ is initialized for the pre-training, we generate m as a contiguous block with a random starting position in the adversarial training. In both training phases, the mask for the first token is always selected as $m_1 = 0$, meaning that the first token given to the model is always real. The masking rate, r_m determines the ratio of masked tokens in an assembly function whose effect on code generation is analyzed further in Section 4.2.4.

SpectreGAN is configured with the embedding vector size of $d = 64$, generator learning rate of $\eta_G = 5 \times 10^{-4}$, discriminator learning rate of $\eta_D = 5 \times 10^{-3}$, critic learning rate of $\eta_C = 5 \times 10^{-7}$ and discount rate of $\gamma = 0.89$ based on the MaskGAN implementation [2]. We select the sequences with a maximum length of 250 tokens, which then build the vocabulary with a size of $V = 419$. We separate 10% of the data set for model validation. SpectreGAN is trained with a batch size of 100 on NVIDIA GeForce GTX 1080 Ti until the validation perplexity converges in Figure 2. The pre-training lasts about 50 hours while the adversarial training phase takes around 30 hours.

4.2.4 Evaluation

SpectreGAN is based on learning masked tokens with the surrounding tokens. The masking rate is not a fixed value which is determined based on the context. Since SpectreGAN is the first study to train on Assembly functions, the choice of masking rate is of utmost importance to generate high quality gadgets. Typically, NLP-based generation techniques are evaluated with their associated perplexity score which is the indicator of how well a token is predicted by the model. Hence, we evaluate the performance of SpectreGAN with various masking sizes and their perplexity scores. In Figure 2, the perplexity converges with the increasing number of training steps, which means the tokens are predicted with a higher accuracy towards the end of the training. SpectreGAN achieves lower perplexity with higher masking rates which indicates that higher masking rates are more preferable for SpectreGAN.

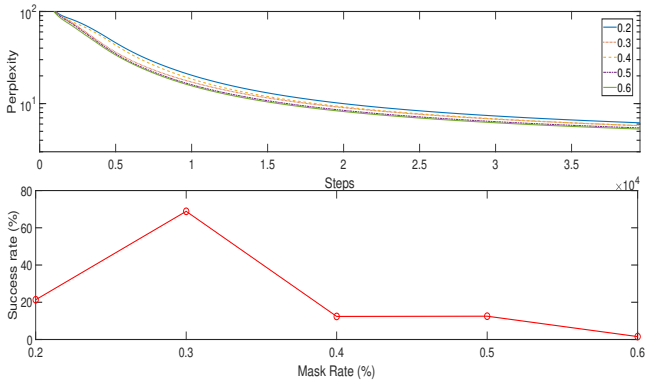

Figure 2: (Above) The validation perplexity decreases at each training step and converges for all r_m. (Below) Spectre gadget success rates are evaluated when different masking rates are used to train SpectreGAN. Spectre gadget success rate shows the percentage of gadgets out of compiled functions.

Even though the higher masking rates yield lower perplexity and assembly functions of high quality in terms of token probabilities, our purpose is to create functions which behave as Spectre gadgets. Therefore, as a second test, we generated 100,000 gadgets for 5 different masking rates. Next, we compiled our gadgets with gcc compiler, and then tested them with all the attacker codes to verify their secret leakage. When SpectreGAN is trained with a masking rate of 0.3, the success rate of gadgets increases up to 72%. Interestingly, the success rate drops for other masking rates, which also demonstrates the importance of masking rate choice. In total, 70,000 gadgets are generated with a masking rate of 0.3 to evaluate the performance of SpectreGAN in terms of the gadget diversity in Section 4.3.

To illustrate an example of the generated samples, we fed the gadget in Listing 2 to SpectreGAN and generated a new gadget in Listing 3. We demonstrate that SpectreGAN is capable of generating realistic assembly code snippets by inserting, removing, or replacing the instructions, registers, and labels. In the Listing 3, the lines that start with the instructions written with red color are generated by SpectreGAN and they correspond to the masked portion of Spectre-V1 gadget given in Listing 2.

[2] https://github.com/tensorflow/models/tree/master/research/maskgan

```
victim_function:
 .cfi_startproc
movl    size(%rip),%eax
cmpq    %rdi,%rax
jbe     .L0
leaq    array1(%rip),%rax
movzbl  (%rdi,%rax),%eax
ror     $1,%rsi
shlq    $9,%rax
leaq    array2(%rip),%rcx
movss   %xmm8,%xmm4
movb    (%rax,%rcx),%al
andb    %al,temp(%rip)
movd    %xmm1,%r14d
test    %r15,%rcx
sbbl    %r13d,%r9d
.L0:
retq
cmovll  %r8d,%r10d
.cfi_endproc
```

Listing 2: Input SpectreV1 gadget

```
victim_function:
 .cfi_startproc
movl    size(%rip),%eax
cmpq    %rdi,%rax
jbe     .L0
leaq    array1(%rip),%rax
movzbl  (%rdi,%rax),%eax
ror     $1,%rsi
shlq    $9,%rax
movb    array2(%rdi),%al
andb    %al,temp(%rip)
.L1:
andb    %r13b,%al
movb    array2(%rax),%al
andb    %al,temp(%rip)
sbbl    %r13d,%r9d
.L0:
retq
cmovll  %r8d,%r10d
.cfi_endproc
```

Listing 3: Generated gadget by SpectreGAN

4.3 Diversity Analysis of Generated Gadgets

Mutational fuzzing and SpectreGAN generated approximately 1.2 million gadgets in total. Since the gadgets are derived from existing examples, it is crucial to analyze their diversity. For this purpose, we randomly select 10,000 samples from fuzzing and SpectreGAN generated gadgets, and then we evaluate the gadget quality with syntactic and microarchitectural analysis as well as their detection rates in oo7 [66] and Spectector [15] tools.

4.3.1 Syntactic Analysis

The quality of generated texts is mostly evaluated by analyzing the number of unique n-grams. For instance, perplexity and BLEU concepts are calculated based on the probabilistic occurrences of n-grams in a sequence. However, these scores are obtained during the training phase, which makes it impossible to evaluate fuzzing generated gadgets. Hence, to determine the diversity of the generated gadgets, we analyze the unique n-grams introduced by fuzzing and SpectreGAN methods.

The number of unique n-grams in newly generated gadgets are compared with the 15 base examples in Table 1. The unique n-grams are calculated as follows: First, unique n-grams produced by fuzzing are identified and stored in a list. Then, the unique n-grams in SpectreGAN gadgets which are not present in fuzzing generated gadgets are noted. Therefore, the unique n-grams generated by SpectreGAN in Table 1 represent the number of n-grams introduced by SpectreGAN excluding fuzzing generated n-grams. In total, the number of unique bigrams (2-grams) is increased from 2,069 to 22,910 to more than a 10-fold increase. While new instructions added by fuzzing improve the diversity in the gadgets, SpectreGAN contributes to the gadget diversity by introducing perturbations. Since the number of instructions increase drastically compared to base gadgets, the unique 5-grams are increased to almost 2 million from 4747 5-grams.

Table 1: Table shows the number of unique n-grams for base gadgets and generated gadgets by fuzzing and SpectreGAN. In the last column the total number of unique n-grams and the factor of increase are given.

	Base	**Fuzzing**	**SpectreGAN**	**Total**
2-grams	2069	15,448	7,462	22,910 (×11)
3-grams	3349	181,606	91,851	273,457 (×82)
4-grams	4161	639,608	460,317	1,099,925 (×264)
5-grams	4747	998,279	921,519	1,919,798 (×404)

4.3.2 Microarchitectural Analysis

The purpose of gadget generation is to introduce various instructions and operands to create diverse gadgets, which also affect the microarchitectural characteristics of the gadgets. However, it is challenging to examine the effects of instructions in the transient domain since they are not visible in the architectural state. After we carefully analyzed the performance counters for the Haswell architecture, we determined that two counters namely, $UOPS\_ISSUED:ANY$ and $UOPS\_RETIRED:ANY$ give an idea to what extent the speculative window is altered. $UOPS\_ISSUED:ANY$ counter is incremented every time a μop is issued which counts both speculative and non-speculative μops. On the other hand, $UOPS\_RETIRED:ANY$ counter only counts the executed and committed μops which automatically excludes speculatively executed μops.

The distribution of generated gadgets and base gadgets are given in Figure 3. The gadget quality improves when the difference between issued and retired μops increases since high difference means the gadgets contain more instructions in the speculative domain. Hence, if a gadget is close to the x-axis and far from the y-axis, the gadget diversity increases in parallel. Especially, gadgets with more instructions in the speculative domain tend to bypass detectors with a higher chance.

It is more likely to obtain more diverse gadgets with fuzzing during which instructions are randomly added. On the other hand, SpectreGAN learns the essential structure of the fuzzing generated gadgets, which yields almost the same number of samples close to x-axis in Figure 3. Moreover, the advantage of SpectreGAN is to automate the creation of gadgets with high accuracy (%72) compared to fuzzing (%5).

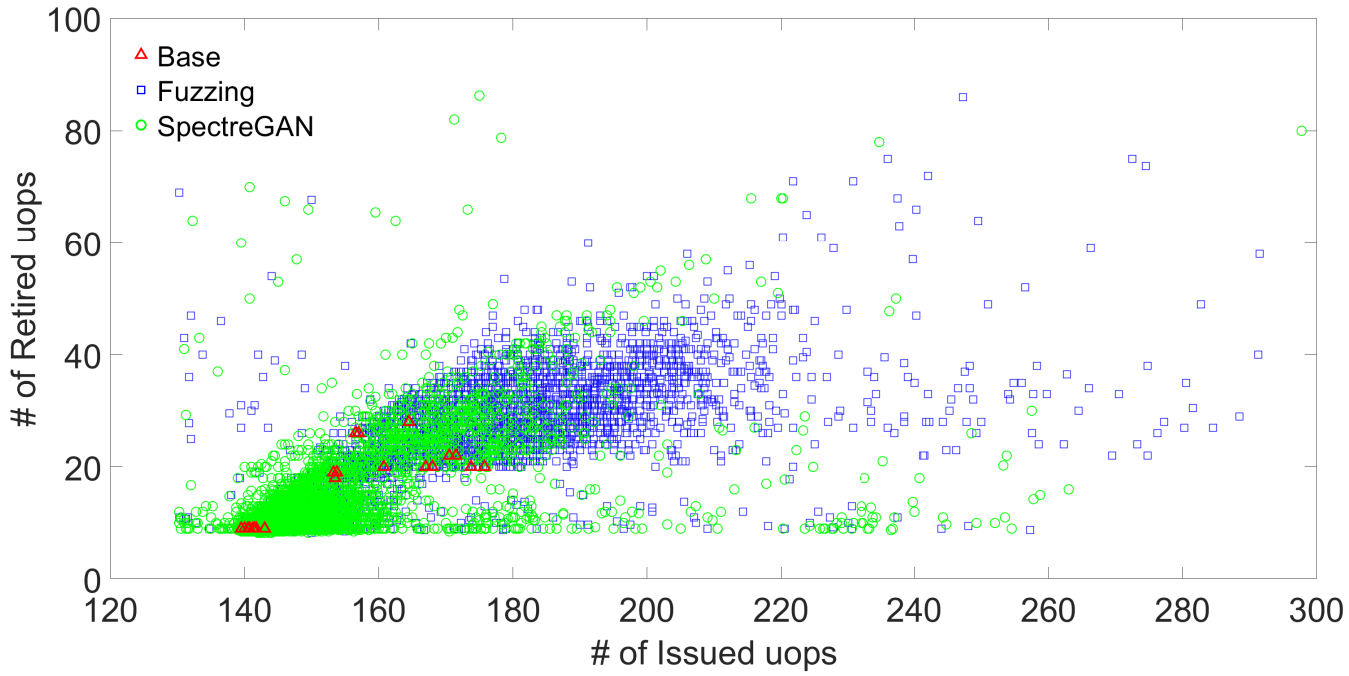

Figure 3: The distribution of base (red), fuzzing generated (blue) and SpectreGAN generated (green) gadgets is given for 2 μops counters. Both SpectreGAN and fuzzing techniques generate diverse set of gadgets for the Haswell architecture.

4.3.3 Detection Analysis

Even though the microarchitectural and syntactic analyses show that fuzzing and SpectreGAN can produce diverse set of gadgets, we aim to enable comprehensive evaluation of detection tools as well as determine the most interesting gadgets in our data set. For this reason, the generated gadgets are fed into the Spectector [15] and oo7 [66] tools to analyze the novelty of the gadgets.

oo7 The *oo7* tool leverages taint analysis to detect Spectre-V1 gadgets. It is based on the Binary Analysis Platform (BAP) [8] which forwards taint propagation along all possible paths after a conditional branch is encountered. *oo7* [3] is built on a set of hand-written rules which cover the existing examples by Kocher [25]. Although our data set size is 1.2 million, we have selected 100,000 samples from each gadget example uniformly random due to the immense time consumption of *oo7* (150 hours for 100K gadgets) which achieves 94% detection rate.

Interestingly, specific gadget types from both fuzzing and SpectreGAN are not caught by *oo7*. When a gadget contains *cmov* or *xchg* or *set* instruction and its variants, it is not identified as a Spectre gadget. Hence, we introduce these new gadgets as new Spectre-V1 gadgets that are listed in Listing 4 and Listing 5. The corresponding assembly files generated from fuzzing and SpectreGAN are given in Appendix A.

[3] https://gitlab.com/igoto/spectre-detector

```
1 void victim_function(size_t x){
2   if(global_condition)
3     x = 0;
4   if(x < size)
5     temp &= array2[array1[x] * 512];
6 }
```

Listing 4: `CMOV` gadget: An example Spectre gadget in C format. When it is compiled with *gcc-7.5* -o2 optimization level, `CMOVcc` gadget bypasses *oo7* tool. The generated assembly version is given in Appendix A.

```
1 size_t prev = 0xff;
2 void victim_function(size_t x) {
3   if (prev < size)
4     temp &= array2[array1[prev] * 512];
5   prev = x;
6 }
```

Listing 5: `XCHG` gadget: When a past value controlled by the attacker is used in the Spectre gadget, *oo7* cannot detect the XCHG gadget

Spectector *Spectector* [15] makes use of symbolic execution technique to detect the potential Spectre-V1 gadgets. For each Assembly file, *Spectector* is adjusted to track 25 symbolic paths of at most 5000 instructions each, with a global timeout of 30 minutes. The remaining parameters are kept as default.

When we analyze the generated gadgets with *Spectector* 23.75% of the gadgets are not detected by *Spectector*. We observed that 96% of the undetected gadgets contain unsupported instruction/register which is the indicator of an implementation issue in *Spectector*. On the other hand, the remaining 1% of the gadgets is more crucial to determine the novelty of the gadgets. After we examined the undetected gadgets, we observed that if the gadgets include either sfence/mfence/lfence or 8-bit registers (%al, %bl, %cl, %dl), they are likely to bypass *Spectector*. We introduce the corresponding novel gadgets and their code snippets in Appendix A.

5 FastSpec: Fast Gadget Detection Using BERT

In an assembly function representation model, the main challenge is to obtain the representation vectors, namely embedding vectors, for each token in a function. Since the skip-gram and RNN-based training models are surpassed by the attention-only models in sentence classification tasks, we introduce FastSpec which applies a lightweight version of BERT.

5.1 Training Procedures

We adopt the same training procedures with BERT on assembly functions, which are called *pre-training* and *fine-tuning*.

5.1.1 Pre-training

The first procedure is *pre-training* which includes two unsupervised tasks. The first task follows a similar approach to MaskGAN by masking a portion of tokens in an assembly function. Differently, the mask positions are selected from 15% of the training sequence and the selected positions are masked and replaced with `<MASK>` token with 0.80 probability, replaced with a random token with 0.10 probability or kept as the same token with 0.10 probability. While the masked tokens are predicted based on the context of other tokens in the sequence, the context vectors are obtained by the multi-head self-attention mechanism.

The second task is the next sentence prediction where the previous sentence is given as an input. Since our assembly code data has no paragraph structure where the separate long sequences follow each other, each assembly function is split into pieces with a maximum token size of 50. For the next sentence prediction task, we add `<CLS>` to each piece. For each piece of function, the following piece is given with the label `IsNext`, and a random piece of function is given with label `NotNext`. FastSpec is trained with the self-supervised approach.

At the end of the *pre-training* procedure, each token is represented by an embedding vector with a size of H. Since it is not possible to visualize the high dimensional embedding vectors, we leverage the t-SNE algorithm [33] which maps the embedding vectors to a three-dimensional space as shown in Figure 4. We illustrate that the embedding vectors for similar tokens are close to each other in three dimensional space as this outcome shows that the embedding vectors are learned efficiently. In Figure 4, the registers with different sizes, floating point instructions, control flow instructions, shift/rotate instructions, set instructions, and MMX instructions/registers are accumulated in separate clusters. The separation among different type of tokens enables to achieve a higher success rate in Spectre gadget detection phase.

5.1.2 Fine-tuning

The second procedure is called *fine-tuning* which corresponds to a supervised sequence classification in FastSpec. This phase enables FastSpec to learn the conceptual differences between Spectre gadgets and general purpose functions through labeled pieces. The pieces that are created for the pre-training phase are merged into a single sequence with a maximum size of 250 tokens. The disassembled object files, which have more than 250 tokens, split into separate sequences. Each sequence is represented by a single `<CLS>` token at the beginning. The benign disassembled files are labeled with `0` and the gadget samples are labeled with `1` for the supervised classification. Then, the embedding vectors of the corresponding `<CLS>` token and position embedding vectors for the first position are summed up. Finally, the resulting vector is fed into the softmax layer which is fine-tuned with supervised training. The output probabilities of the softmax layer are the predictions on the assembly code sequence.

5.2 Training Details and Evaluation

We combine the assembly data set that was generated in Section 4 and the disassembled Linux libraries to train FastSpec. Although it is possible that Linux libraries contain Spectre-V1 gadgets, we assume that the total number of hidden Spectre gadgets are negligible comparing the total size of the data set. Therefore, the model treat those gadgets as noise, which have no effect on the performance of FastSpec. In total, a data set of 107 million lines of assembly code is collected which consists of 370 million tokens after the pre-processing. We separate 80% of the data set for training and validation, and the remaining 20% is used for FastSpec evaluation. While the same pre-processing phase in Section 4.2.3 is implemented, we further merge similar types of tokens to decrease the total vocabulary size. We replace all labels, immediate values and out-of-vocabulary tokens with `<label>`, `<imm>` and `<UNK>`, respectively. After the pre-processing, the vocabulary size is reduced to 960.

We set the number of Transformer blocks as $L = 3$, the hidden size as $H = 64$, and the number of self-attention heads as $A = 2$. We train FastSpec on NVIDIA Titan XP GPU. The *pre-*

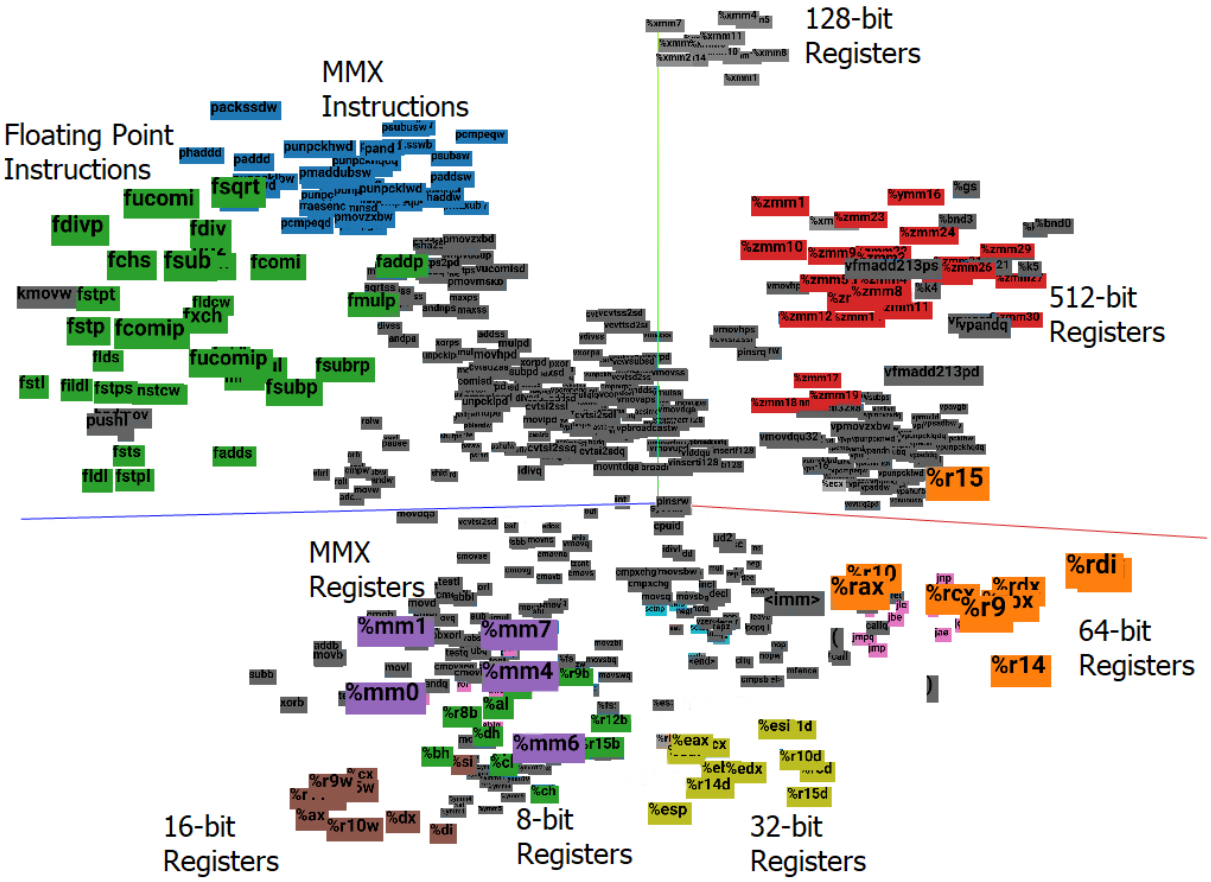

Figure 4: 3-D visualization for the distribution of instructions and registers after t-SNE is applied to embedding vectors. Similar instructions and registers have the same colors. The unrelated instructions are separated from each other in the three-dimensional space after the pre-training.

training phase takes approximately 6 hours with a sequence length of 50. We further train the positional embeddings for 1 hour with a sequence length of 250. The fine-tuning takes only 20 minutes on the pre-trained model to achieve classifying all types of samples in the test data set correctly. Note that the training time is less than previous NLP techniques in the literature since BERT [9] leverages GPU parallelization, significantly. The analysis duration is measured on Intel Xeon CPU E5-2637 v2 @3.50GHz.

In the evaluation of FastSpec, we obtained 1.3 million true positives and 110 false positives (99.9% precision rate) in the test data set which demonstrates the high performance of FastSpec. We assume that the false positives are Spectre-like gadgets in Linux libraries, which needs to be explored deeply in the future work. Moreover, we only have 55 false negatives (99.9% recall rate) which yields 0.99 F-1 score on the test data set.

In the next section, we show that FastSpec achieves high performance and extremely fast gadget detection without needing any GPU acceleration since FastSpec is built on a lightweight BERT implementation.

5.3 Case Study: Phoronix Test Suite Analysis

The performance comparison between FastSpec and other detection tools is evaluated on the Phoronix Test Suite v5.2.1. The selected benign files have source codes since it is required to obtain the assembly files for *Spectector* tool. The assembly files are generated by compiling the source C codes with the *gcc* compiler. On the other hand, the binary files are generated at the time of the test installation, therefore, there is no further processing required before testing the binary files in *oo7*. For FastSpec, the disassembled binary files are given as an input. Note that since the larger benchmarks take more time to be analyzed by *oo7*, we preferred small size files to make the comparison with *Spectector* and FastSpec easier.

Timing The overall timing results for various benchmarks are given in Table 2. The analysis time of *oo7* and *Spectector* increases drastically with the number of conditional branches since the tools analyze both paths after a conditional branch is encountered. On the other hand, FastSpec analysis time increases linearly with the binary size. We observed that the

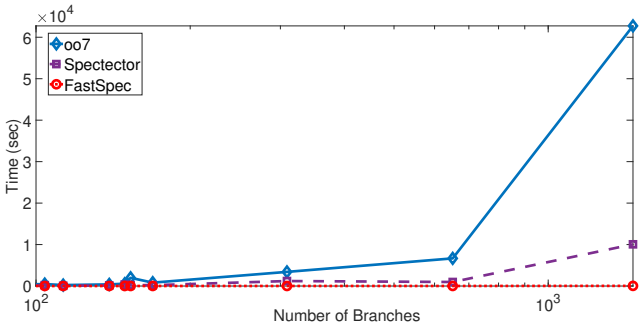

Figure 5: The processing time of FastSpec is independent of the number of branches whereas for Spectector and oo7 the analysis time increases drastically.

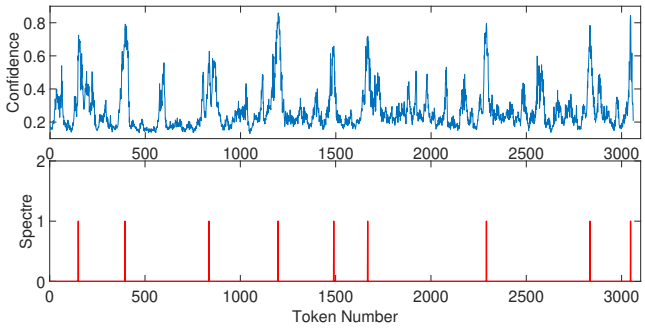

Figure 6: OpenSSL crypto library (libcrypto−lib−ex_data) analysis by FastSpec. The blue line represents the confidence ratio for a window of 50 tokens. When the confidence exceeds 0.6, the window is classified as Spectre gadget as the red line shows.

pre-processing phase takes the major portion in the analysis time of FastSpec while the inference time is in the order of microseconds.

The effect of the increasing number of branches on time consumption is clear in Crafty and Clomp benchmarks in Table 2. Even though Crafty benchmark has only 10,796 branches, *oo7* and *Spectector* analyze the file in more than **10 days** (the analysis process is terminated after 10 days) and **2 days**, respectively. In Figure 5, we show that both tools are not sufficiently scalable to be used in real-world applications, especially when the files contain thousands of conditional branches. Especially *oo7* shows an exponential behavior because of the forced execution approach which executes every possible path of the conditional branches. In contrast, FastSpec analyzed the same Crafty benchmark under 6 minutes which is a significant improvement over rule-based tools.

Note that Byte benchmark has a higher number of branches than most of the remaining benchmarks. However, it consists of multiple files that need to be tested separately which takes less time to analyze in total. Consequently, FastSpec is faster than *oo7* and *Spectector* 455 times and 75 times on average, respectively.

Number of Gadgets The number of gadgets found by the tools varies significantly. While *oo7* and FastSpec report each Spectre gadget in a binary file, *Spectector* outputs whether a function contains a Spectre gadget or not. Hence, if a control or data leakage is found in a function, it is reported as a vulnerable function. Thus, we give the number of vulnerable functions instead of the number of gadgets in Table 2 for *Spectector*. FastSpec is more likely to detect gadgets that are not discovered by *oo7* and *Spectector* such as in Xsbench and Stream benchmarks. On the other hand, FastSpec detects the most number of gadgets compared to other tools when a confidence rate threshold of 0.6 is applied for the gadget classification.If two gadgets are closer to each other than the window size, the gadgets are counted as one gadget. Moreover, the confidence rate threshold of FastSpec can be increased to detect the Spectre gadgets with higher probabilities. FastSpec is flexible since a confidence rate is given for each Spectre gadget, which can be tuned based on the developers' need.

5.4 Case Study: OpenSSL Analysis

We use FastSpec to analyze OpenSSL crypto libraries for any Spectre-V1 gadgets. We focus on OpenSSL 1.1.1g, as it is a popular in commercial software. We selected general purpose libraries which are used in many crypto functions whose list is given in Table 4 in the Appendix. In total, 109 object files are extracted from libraries to analyze with FastSpec.

First, we apply the same pre-processing procedures as explained in Section 5.2 to obtain the tokens. The total number of tokens is 203,055 while the analysis time is around 17 minutes. An resule of a sample gadget detection scan on a crypto library is given in Figure 6 where the potential gadgets are identified with FastSpec. The confidence rate increases with the potential gadgets which mostly consist of conditional jump instructions. The confidence threshold is chosen as 0.6 as in the Phoronix benchmark case. In total, we discovered 379 potential Spectre gadgets which may be subject to exploitation. In contrast, these gadgets are not always exploitable since the third-party user input is not feasible. Therefore, we claim that these gadgets can be further analyzed with rule-based detection techniques for leakage detection.

6 Discussion and Limitations

Scalability: Although we limit the scope of this paper to generating and detecting the Spectre-V1 gadgets on x86 assembly code, the use of SpectreGAN and FastSpec can always be extended to other architectures and applications with only mild effort. Furthermore, specially designed architectures are not needed when pre-trained embedding representations are used [9]. Therefore, the pre-trained FastSpec model can be used for any other vulnerability detection, cross-architecture code migration, binary clone detection, and many other assembly-level tasks.

Table 2: Comparison of *oo7* [66], Spectector [15] and FastSpec on the Phoronix Test Suite. The last column shows that FastSpec is on average 455 times faster than *oo7* and 75 times faster than *Spectector*.

Program	Binary Size (KB)	#Cond. Branches	#Functions	oo7		Spectector		FastSpec	
				#Detected	Time (sec)	#Detected	Time (sec)	#Detected	Time (sec)
Byte	183.5	363	30	116	400	3	115	**70**	**14**
Cachebench	27.7	149	18	0	556	5	360	**24**	**5**
Clomp	79.4	1464	63	0	17.5 hours	18	10057	**30**	**35**
Crafty	594.8	10796	189	342	>10 days	73	18 hours	**798**	**315**
C-ray	27.2	139	9	23	395	5	153	**9**	**8**
E-bizzy	18.5	104	11	0	467	5	206	**8**	**3**
Mbw	13.2	70	4	0	145	2	34	**9**	**2**
M-queens	13.4	51	2	8	136	2	24	**9**	**2**
Postmark	38.0	309	38	37	3409	14	1202	**30**	**10**
Stream	22.0	113	2	0	231	0	63	**17**	**4**
Tio-test	36.1	169	16	0	813	4	201	**9**	**9**
Tscp	40.8	651	7	0	6667	2	972	**42**	**12**
T-test	13.7	47	5	0	99	3	36	**11**	**3**
Xsbench	27.9	153	81	8	1985	0	249	**19**	**7**

Challenges in Assembly Code Generation The challenges faced in the regular text generation with GANs [12,72] also exist in assembly code generation. One of the challenges is *mode collapse* in the generator models. Although training the model and generating the gadgets with masking help reducing mode collapse, we observed that our generator model still generates some tokens or patterns of tokens repetitively, which reduces the quality of the generated samples, compilation, and real gadget generation rates.

In regular text generation, even if the position of a token changes in a sequence, the meaning of the sequence may change while it would be still somewhat acceptable. However, if the position of a token in an assembly function changes, it may result in a compilation error because of the incorrect syntax. Even if the generated assembly function has the correct assembly syntax, the function behavior may be completely different from the expected one due to the positions of a few instructions and registers.

Window Size: Since Transformer architecture has no utilization of recurrent modeling as RNNs do, the maximum sequence length is needed to be set before the training procedures. Therefore, the sliding window size can be set to at most the maximum sequence length. On the other hand, our experiments show that lower window sizes compared to maximum sequence length detect more code snippets that have a higher possibility of being a Spectre gadget and provide fine-grain information on the sequence. The further analysis on window size selection is given in Appendix B.

7 Conclusion

This work for the first time proposed NLP inspired approaches for Spectre gadget generation and detection. First, we extended our gadget corpus to 1.1 million samples with a mutational fuzzing technique. We introduced SpectreGAN tool that achieves a high success rate in creating new Spectre gadgets by automatically learning the structure of gadgets in assembly language. SpectreGAN overcomes the difficulties of training a large model for assembly language, which is a completely different domain compared to natural language. We demonstrate that 72% of the compiled code snippets behave as a Spectre gadget which is a huge improvement over fuzzing based generation. Furthermore, we show that our generated gadgets span the speculative domain by introducing new instructions and their perturbations which yield diverse and novel gadgets. The most interesting gadgets are also introduced as new examples of Spectre-V1 gadgets. Finally, we propose FastSpec which is based on BERT style neural embedding to detect the hidden Spectre gadgets. We demonstrate that for large binary files, FastSpec is 2 to 3 orders of magnitude faster than *oo7* and *Spectector* while it still detects more gadgets. We also demonstrate the scalability of FastSpec on OpenSSL libraries to detect potential gadgets.

Acknowledgments

We are grateful to *oo7*'s [66] author Ivan Gotovchits, for helping us running the *oo7* tool accurately. This work is supported by the National Science Foundation, under grant CNS-1814406.

References

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Re-*

search, pages 214–223, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[3] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.

[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. extended classification tree at https://transient.fail/.

[6] Chandler Carruth. Rfc: Speculative load hardening (a spectre variant 1 mitigation), 2018.

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[8] T. Avgerinos D. Brumley, I. Jager and E. J. Schwartz. Bap: A binary analysis platform, 2011.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[11] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In *ICLR*, 2019.

[12] William Fedus, Ian J. Goodfellow, and Andrew M. Dai. Maskgan: Better text generation via filling in the _____. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[15] Marco Guarnieri, Boris Köpf, Jose F. Morales, Jan Reineke, and Andres Sanchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[16] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777, 2017.

[17] Jiaxian Guo, Sidi Lu, Han Cai, Weinan Zhang, Yong Yu, and Jun Wang. Long text generation via adversarial training with leaked information. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[18] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. Specusym: Speculative symbolic execution for cache timing leak detection. *arXiv preprint arXiv:1911.00507*, 2019.

[19] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[20] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[21] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[22] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[23] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation, 2018.

[24] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses, 2018.

[25] Paul Kocher. "spectre mitigations in microsoft's c/c++ compiler", 2018.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[27] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[28] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating spectre attacks using cfi informed speculation, 2019.

[29] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[30] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[31] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. Adversarial learning for neural dialogue generation. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2157–2169, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[33] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[34] Giorgi Maisuradze and Christian Rossow. ret2spec. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jan 2018.

[35] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 747–761, 2019.

[36] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, Wil Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. *arXiv preprint arXiv:2003.05503*, 2020.

[37] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[38] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 3111–3119, Red Hook, NY, USA, 2013. Curran Associates Inc.

[40] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[41] Weili Nie, Nina Narodytska, and Ankit Patel. Relgan: Relational generative adversarial networks for text generation. 2018.

[42] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2642–2651. JMLR. org, 2017.

[43] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier GANs. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[44] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. *arXiv preprint arXiv:1905.10311*, 2019.

[45] A. Pardoe. Spectre mitigations in msvc, Jan. 2018.

[46] Minghui Qiu, Feng-Lin Li, Siyu Wang, Xing Gao, Yan Chen, Weipeng Zhao, Haiqing Chen, Jun Huang, and Wei Chu. Alime chat: A sequence to sequence and rerank based chatbot engine. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 498–503, 2017.

[47] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[48] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[49] Benjamin J Radford, Bartley D Richardson, and Shawn E Davis. Sequence aggregation rules for anomaly detection in computer network traffic. *arXiv preprint arXiv:1805.03735*, 2018.

[50] Kimberly Redmond, Lannan Luo, and Qiang Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. *arXiv preprint arXiv:1812.09652*, 2018.

[51] Scott Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. *arXiv preprint arXiv:1605.05396*, 2016.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[53] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. Context: Leakage-free transient execution. *arXiv preprint arXiv:1905.09100*, 2019.

[54] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299. Springer, 2019.

[55] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008.

[56] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018.

[57] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[58] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[59] Wilson L. Taylor. "cloze procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433, 1953.

[60] P. Turner. "retpoline: a software construct for preventing branch-target-injection.", 2018.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, August 2018.

[62] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[64] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics. In *Advances in neural information processing systems*, pages 613–621, 2016.

[65] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *arXiv preprint arXiv:1909.00647*, 2019.

[66] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *arXiv preprint arXiv:1807.05843*, 2018.

[67] Ke Wang and Xiaojun Wan. Sentigan: Generating sentimental texts via mixture adversarial networks. In *IJCAI*, pages 4446–4452, 2018.

[68] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807, 2018.

[69] Huikai Wu, Shuai Zheng, Junge Zhang, and Kaiqi Huang. Gp-gan: Towards realistic high-resolution image blending. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2487–2495, 2019.

[70] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 428–441. IEEE Press, 2018.

[71] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019.

[72] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 2852–2858. AAAI Press, 2017.

[73] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*, 2018.

A Gadget Examples from *oo7* and *Spectector* Tests

```
victim_function:
    xchg    %rdi, %r13
    cmpl    %esp, %esp
    movl    array1_size(%rip), %eax
    shr     $1,   %r11
    cmpq    %rdi, %rax
    jbe     .LBB1_1
    addq    %r13, %r11
    leaq    array1(%rip), %rax
    movzbl  (%rdi,%rax), %edi
    jmp     leakByteNoinlineFunction
.LBB1_1:
    retq
leakByteNoinlineFunction:
    movl    %edi, %eax
    shlq    $9, %rax
    leaq    array2(%rip), %rcx
    movb    (%rax,%rcx), %al
    andb    %al, temp(%rip)
    retq
```

Listing 6: While generating gadgets with mutational fuzzing technique, this code is generated by our algorithm from Kocher's example 3 (using clang-6.0 with 02 optimization). The xchg %rdi, %r13 instruction stores the content of %rdi register to %r13 register. In the next iterations of leaking the secret, the stored value is being accessed and as a result the *oo7* is fooled

```
victim_function:
    movl    size(%rip), %eax
    cmpq    %rax, %rdi
    jae     .B1.2
    movzbl  array1(%rdi), %eax
    shlq    $9, %rax
    xorb    %al, %al
    movb    array2(%rax), %dl
    andb    %dl, temp(%rip)
.B1.2:
    ret
```

Listing 7: *xorb %al, %al* is added to 1^{st} example in Kocher examples [25]. *Spectector* is no longer able to detect the leakage due to the zeroing %al register.

```
victim_function:
.LFB23:
    movl    global_condition(%rip), %eax
    testl   %eax, %eax
    movl    $0, %eax
    cmovne  %rax, %rdi
    movslq  array1_size(%rip), %rax
    cmpq    %rdi, %rax
    jbe     .L1
    leaq    array1(%rip), %rax
    leaq    array2(%rip), %rdx
    movzbl  (%rax,%rdi), %eax
    sall    $12, %eax
    cltq
    movzbl  (%rdx,%rax), %eax
    andb    %al, temp(%rip)
.L1:
    rep ret
```

Listing 8: When the C code in Listing 4 compiled with certain optimizations (gcc 7-4 with O2 enabled) the Assembly code contains CMOV instruction which fools *oo7*

```
victim_function:
    seta    %sil
    cmpl    $0, (%rsi)
    je      .LBB0_2
    addl    %r15d, %r12d
    sarq    $1,    %r11
    addb    %sil, %r15b
    movzbl  array1(%rdi), %eax
    ja      .L1324337
    testw   %r10w, %ax
    shlq    $12, %rax
    nop
    movb    array2(%rax), %al
.L1324337:
    andb    %al, temp(%rip)
.LBB0_2:
    retq
```

Listing 9: While generating gadgets with mutational fuzzing technique, this code is generated by our algorithm from Kocher's example 9 (using clang-6.0 with 02 optimization). The seta %sil instruction sets the lowest 8-bit of %rsi register based on a condition which results in fooling the *oo7*

B Fine-tuning Sliding Window

We analyze the performance of FastSpec with synthetic data to obtain the optimal value for the window size. We insert a known Spectre code with a size of 89 tokens to a random benchmark code given in Listing 1. In Figure 7, FastSpec detects a simple Spectre gadget with a 0.9 confidence rate for a contiguous interval of 108 tokens. This result shows that FastSpec can detect a hidden gadget with a sliding window size of 50, even though the gadget length is larger than the window size. The window size is set to 50 in the remaining experiments.

When FastSpec is implemented with a window size of 50 to analyze Cachebench benchmark, 24 potential Spectre gadgets are discovered in Figure 7. The threshold of 0.6 is chosen as the threshold to distinguish the benign and Spectre gadgets.

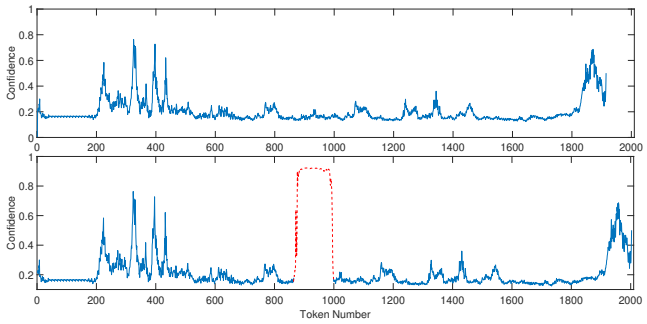

Figure 7: (Above) The confidence rates of sliding windows with token size 50 is shown for a benign assembly function. (Below) The inserted Spectre-V1 gadget is detected with 90% confidence rate in the same function.

C Instructions and registers inserted randomly in the fuzzing technique

Table 3: Instructions and registers inserted randomly in the fuzzing technique.

Instructions					
add	cmovll	jns	movzbl	ror	subl
addb	cmp	js	movzwl	sall	subq
addl	cmpb	lea	mul	salq	test
addpd	cmpl	leal	nop	sarq	testb
addq	cmpq	leaq	not	sar	testl
andb	imul	lock	notq	sal	testq
andl	incq	mov	or	sbbl	testw
andq	ja	movapd	orl	sbbq	xchg
call	jae	movaps	orq	seta	xor
callq	jbe	movb	pop	setae	xorb
cmova	je	movd	popq	sete	xorl
cmovaeq	jg	movdqa	prefetcht0	shll	xorq
cmovbe	jle	movl	prefetcht1	shlq	lfence
cmovbq	jmp	movq	push	shr	sfence
cmovl	jmpq	movslq	pushq	sub	mfence
cmovle	jne	movss	rol	subb	
Registers					
rax	eax	ax	al	xmm0	ymm0
rbx	ebx	bx	bl	xmm1	ymm1
rcx	ecx	cx	cl	xmm2	ymm2
rdx	edx	dx	dl	xmm3	ymm3
rsp	esp	sp	spl	xmm4	ymm4
rbp	ebp	bp	bpl	xmm5	ymm5
rsi	esi	si	sil	xmm6	ymm6
rdi	edi	di	dil	xmm7	ymm7
r8	r8d	r8w	r8b	xmm8	ymm8
r9	r9d	r9w	r9b	xmm9	ymm9
r10	r10d	r10w	r10b	xmm10	ymm10
r11	r11d	r11w	r11b	xmm11	ymm11
r12	r12d	r12w	r12b	xmm12	ymm12
r13	r13d	r13w	r13b	xmm13	ymm13
r14	r14d	r14w	r14b	xmm14	ymm14
r15	r15d	r15w	r15b	xmm15	ymm15

Table 4: OpenSSL Analysis

Program	Binary Size (KB)	#Detected
libcrypto – lib – mem_sec.o	23	21
libcrypto – shlib – mem_sec.o	23	21
tls13secretstest – bin – packet.o	17	15
libcrypto – lib – ex_data.o	12	8
libcrypto – shlib – ex_data.o	12	8
libfips – lib – ex_data.o	12	8
liblegacy – lib – ex_data.o	12	8
libcrypto – lib – o_str.o	7.5	7
libcrypto – lib – params.o	13	7
libcrypto – lib – provider_core.o	26	7
libcrypto – lib – sparse_array.o	5.1	7
libcrypto – shlib – o_str.o	7.5	7
libcrypto – shlib – params.o	13	7
libcrypto – shlib – provider_core.o	26	7
libcrypto – shlib – sparse_array.o	5.1	7
libfips – lib – o_str.o	7.5	7
libfips – lib – params.o	13	7
libfips – lib – sparse_array.o	5.1	7
liblegacy – lib – o_str.o	7.5	7
liblegacy – lib – params.o	13	7
liblegacy – lib – sparse_array.o	5.1	7
libcrypto – lib – core_namemap.o	11	6
libcrypto – shlib – core_namemap.o	11	6
libcrypto – lib – context.o	8.3	5
libcrypto – lib – self_test_core.o	6.4	5
libcrypto – shlib – context.o	8.3	5
libcrypto – shlib – self_test_core.o	6.4	5
libfips – lib – context.o	6.1	5
libfips – lib – core_namemap.o	8.9	5
libfips – lib – self_test_core.o	6.2	5
liblegacy – lib – context.o	8.3	5
libcrypto – lib – asn1_dsa.o	4.8	4
libcrypto – lib – o_time.o	3.1	4
libcrypto – lib – packet.o	11	4
libcrypto – lib – param_build.o	16	4
libcrypto – shlib – asn1_dsa.o	4.8	4
libcrypto – shlib – o_time.o	3.1	4
libcrypto – shlib – packet.o	11	4
libcrypto – shlib – param_build.o	16	4
libfips – lib – asn1_dsa.o	4.8	4
libfips – lib – packet.o	11	4
libfips – lib – param_build.o	16	4
libfips – lib – provider_core.o	20	4
liblegacy – lib – asn1_dsa.o	4.8	4
liblegacy – lib – packet.o	11	4
liblegacy – lib – param_build.o	16	4
libssl – lib – packet.o	11	4
libssl – shlib – packet.o	11	4
libcrypto – lib – der_writer.o	6.2	3
libcrypto – lib – initthread.o	8.1	3
libcrypto – lib – mem.o	4.1	3
libcrypto – shlib – der_writer.o	6.2	3
libcrypto – shlib – initthread.o	8.1	3
libcrypto – shlib – mem.o	4.1	3
libfips – lib – der_writer.o	6.2	3
liblegacy – lib – der_writer.o	6.2	3
liblegacy – lib – initthread.o	8.1	3
libcrypto – lib – core_algorithm.o	2.3	2
libcrypto – lib – cryptlib.o	4.0	2
libcrypto – lib – param_build_set.o	3.9	2
libcrypto – lib – provider_conf.o	6.0	2
libcrypto – lib – threads_pthread.o	4.9	2
libcrypto – lib – x86_64cpuid.o	3.5	2
libcrypto – shlib – core_algorithm.o	2.3	2
libcrypto – shlib – cryptlib.o	4.0	2
libcrypto – shlib – param_build_set.o	3.9	2
libcrypto – shlib – provider_conf.o	6.0	2
libcrypto – shlib – threads_pthread.o	4.9	2
libcrypto – shlib – x86_64cpuid.o	3.5	2
libfips – lib – core_algorithm.o	2.3	2
libfips – lib – cryptlib.o	4.0	2
libfips – lib – param_build_set.o	3.9	2
libfips – lib – x86_64cpuid.o	3.5	2
liblegacy – lib – cryptlib.o	4.0	2
liblegacy – lib – param_build_set.o	3.9	2
liblegacy – lib – threads_pthread.o	4.9	2
liblegacy – lib – x86_64cpuid.o	3.5	2
libcrypto – lib – params_from_text.o	5.2	1
libcrypto – shlib – params_from_text.o	5.2	1
libfips – lib – params_from_text.o	5.2	1
libfips – lib – threads_pthread.o	4.3	1
liblegacy – lib – params_from_text.o	5.2	1
libcrypto – lib – bsearch.o	1.7	0
libcrypto – lib – core_fetch.o	3.6	0
libcrypto – lib – cpt_err.o	3.2	0
libcrypto – lib – ctype.o	2.3	0
libcrypto – lib – cversion.o	4.2	0
libcrypto – lib – info.o	4.1	0
libcrypto – lib – init.o	13	0
libcrypto – lib – o_dir.o	2.4	0
libcrypto – lib – provider.o	4.1	0
libcrypto – lib – trace.o	5.4	0
libcrypto – lib – uid.o	1.5	0
libcrypto – shlib – bsearch.o	1.7	0
libcrypto – shlib – core_fetch.o	3.6	0
libcrypto – shlib – cpt_err.o	3.2	0
libcrypto – shlib – ctype.o	2.3	0
libcrypto – shlib – cversion.o	4.2	0
libcrypto – shlib – info.o	4.1	0
libcrypto – shlib – o_dir.o	2.4	0
libcrypto – shlib – provider.o	4.1	0
libcrypto – shlib – trace.o	5.4	0
libcrypto – shlib – uid.o	1.5	0
libfips – lib – bsearch.o	1.7	0
libfips – lib – core_fetch.o	3.6	0
libfips – lib – ctype.o	2.3	0
libfips – lib – initthread.o	4.3	0
liblegacy – lib – bsearch.o	1.7	0
liblegacy – lib – ctype.o	2.3	0
Total	**777.2**	**379**