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

M. Caner Tol, *Worcester Polytechnic Institute*, *mtol@wpi.edu*
Berk Gulmezoglu, *Iowa State University*, *bgulmez@iastate.edu*
Koray Yurtseven, *Worcester Polytechnic Institute*, *kcyurtseven@wpi.edu*
Berk Sunar, *Worcester Polytechnic Institute*, *sunar@wpi.edu*

***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. Moreover, detection tool evaluations are based only on a handful of these gadgets, as it requires arduous effort to craft new gadgets manually.**

In this work, we employ both fuzzing and deep learning techniques to automate the 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) in the context of assembly code generation 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 a novel high dimensional Neural Embeddings technique (BERT). For the case studies, we demonstrate that FastSpec discovers potential gadgets with a high success rate in OpenSSL libraries and Phoronix benchmarks. Further, FastSpec offers much greater flexibility and time-related performance gain compared to the existing tools and therefore can be used for gadget detection in large-scale software.

***Index Terms*—Spectre, Generative Adversarial Networks, Neural Embeddings, Vulnerability Analysis**

1. Introduction

The new era of microarchitectural attacks began with newly discovered Spectre [1] and Meltdown [2] attacks, which may be exploited to exfiltrate confidential information through microarchitectural channels during speculative and out-of-order executions. The Spectre attacks target vulnerable code patterns called gadgets, which leak information during speculatively executed instructions. While the initial variants of Spectre [1] exploit conditional and indirect branches, Koruyeh et al. [3] propose another Spectre variant by poisoning the entries in Return-Stack-Buffers (RSBs). Moreover, new Spectre-type attacks [3], [4] are implemented against the SGX environment and even remotely over the network [5]. 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 [6] and symbolic execution [7], [8] to identify potential gadgets in benign applications. However, the methods proposed so far are associated with two shortcomings: (1) the low number of Spectre gadgets prevents the comprehensive evaluation of the tools, (2) time consumption exponentially increases when the binary files become larger. 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 natural language and text processing tasks [9]. Later, NLP techniques have been used in the security domain, such as network traffic [10] and vulnerability analysis [11]. Such applications leverage word [12] or paragraph [13] embedding techniques to learn the vector representations of the text. The success of these techniques heavily depends on the large data sets, which ease training scalable and robust NLP models. However, for Spectre, for instance, the number of available gadgets is only 15, making 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 [15], [16] to text-to-image translation [17], etc. While the early applications of GANs focused on computer vision, implementing the same techniques in NLP tasks poses a challenge due to the lack of continuous space in the text. Various mathematical GAN-based techniques have been proposed to achieve better success in generating human-like sentences to overcome this obstacle [18], [19]. However, it is still unclear whether GANs can be implemented in the context of computer security to create application-specific code snippets. 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 [12], [13] 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 [20] was proposed by the Google AI team, which learns the relations between different words in a sentence by applying a self-attention mechanism [21]. BERT has exhibited superior performance compared to previous techniques [22], [23] 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 [24], [25]. However, it is still unclear whether it will be effective to model a low-level programming language, such as Assembly language, and help build more robust malware detection tools, which is the goal of this paper.

Our Contributions Our contributions are twofold. First, we increase the diversity of Spectre gadgets with the mutational fuzzing technique. We start with 15 examples [26] and produce 1 million gadgets by introducing various instructions and operands to the existing gadgets. Then, we propose a GAN-based tool, namely, SpectreGAN, which learns the distribution of 1 million Spectre gadgets to generate new gadgets with high accuracy. The generated gadgets are evaluated from both semantic and microarchitectural aspects to verify their diversity and quality. Furthermore, we introduce novel gadgets that are not detected by state-of-the-art detection tools.

In the second part, we introduce FastSpec, a high dimensional neural embedding based detection technique derived from BERT, to obtain a highly accurate and fast classifier for Spectre gadgets. We train FastSpec with generated gadgets and achieve a 0.998 Area Under the Curve (AUC) score for OpenSSL libraries in the test phase. Further, we apply FastSpec on Phoronix benchmark tests to show that FastSpec outperforms taint analysis-based and symbolic execution-based detection tools as well as significantly decreases the analysis time.

In summary,

- We extend 15 base Spectre examples to 1 million gadgets by applying a mutational fuzzing technique,
- 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 and limitations 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, causing them 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 period before the flush is commonly referred to as the transient domain.

There are two classes of attacks in the transient domain [27]. The first one is called Meltdown-type attacks [2], [28]–[32] which exploit delayed permission checks and lazy pipeline flush in the re-order buffer. The other class is Spectre-type attacks [1], [3], [33]–[35] that exploit the speculative execution. As most Meltdown-type attacks are fixed in latest microarchitectures and Spectre-type attacks are still applicable to a wide range of targets [27], i.e., Intel, AMD, and ARM CPUs, we focus on Spectre-V1 attacks in this study.

Some researchers proposed new designs requiring a change in the silicon level [36], [37] while others proposed software solutions to mitigate transient execution attacks [38], [39]. Although these mitigations are effective against Spectre-type attacks, most of them are not used because of the drastic performance degradation [40]or the lack of support in the current hardware. Hence, Spectre-type attacks are not entirely resolved yet, and finding an efficient countermeasure is still an open problem.

2.1.1. Spectre. Since a typical software consists of branches and instruction/data dependencies, modern CPUs have components for predicting conditional branches' outcomes 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, 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, making it possible to leak the data through side-channel analysis.

2.1.2. Program Analysis Techniques. There are two main program analysis techniques that are commonly used to detect Spectre gadgets.

Taint Analysis: Taint analysis tracks outside user-controlled variables that possibly leak any secret data. If the tainted variables are consumed by a new variable in the program flow, the latter is also tainted in the information flow. This technique is commonly used in vulnerability detection [41], malware analysis , [42], [43] and web applications [44], [45] where user input misuses are highly likely. Similarly, in Spectre gadgets, the secret dependent operations after conditional branches are potential secret leakage sources. In particular, when the branch decision depends on the user input, the secret is subject to be revealed in the speculative execution state. In order to detect the Spectre-V1 based leakage in benign programs, the taint analysis technique is used in *oo7*.

Symbolic Execution: Symbolic execution is a technique to analyze the program with symbolic inputs. Each path of the conditional branch is executed symbolically to determine the values, resulting in unexpected bugs. The symbolic execution is applied to detect potential information leakage in benign applications. For instance, *Spectector* [8] aims to identify the memory and control leaks by supplying symbolic inputs to target functions. While the symbolic execution provides a good understanding of underlying bugs for different input values, it is challenging to apply for large-scale projects due to high resource demand.

2.2. Natural Language Processing

2.2.1. seq2seq Architecture. Sequence to sequence mapping is a challenging process since the text data set has no numeric values. First, the text data is converted to numeric values with embedding methods [12], [46]. Then, a DNN model is trained with vector representations of the text.

A new approach called seq2seq [22] was introduced to model 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 input sequence's mapping 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 [47] 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] called generative adversarial networks (GANs). The generative models are trained with a separate discriminator model under an adversarial setting. In [14], the training of the generative model is defined as,

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

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.

MaskGAN [18] 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 which improves the performance of the fixed-length encoder vectors. 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 to learn the missing texts in a sequence. For this purpose, some tokens are masked that is conditioned on the surrounding context. This technique increases the chance of generating longer and more meaningful sequences out of GANs.

2.2.3. Transformer and BERT. Although recurrent models with attention mechanisms learn the representations of long sequences, attention-only models, namely *Transformer* architectures [21], are shown to be highly effective in terms of computational complexity and performance on long-range dependencies. Similar to *seq2seq* architecture, *the Transformer* architecture consists of an 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 a 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 as the 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. 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. [20] 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 a detailed description of the architecture, we refer the readers to [20], [21]. 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 [1], 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 [3], [35]. Another variant of Spectre called Speculative Store Bypass [33] takes advantage of the 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 [5], port contention [48], or control flow hijack attack based on return-oriented programming [49] to leak secret data.

Defenses against Spectre There are various detection methods for speculative execution attacks. Taint analysis is used in *oo7* [6] 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 [50], [51]. The second method relies on symbolic execution analysis. Spectector [8] symbolically executes the programs where the conditional branches are treated as mispredicted. Furthermore, SpecuSym [52] and KleeSpectre [7] 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 [53] collects performance counter values to detect mispredicted branches and speculative execution domain. Finally, Specfuzz [54] leverages a fuzzing strategy to test functions with diverse set of inputs. Then, the tool analyzes the control flow paths and determines the most likely vulnerable code snippets 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 [55] by observing the binary code running in the system. Alternatively, the binary can also be analyzed statically [56]. 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 the binary analysis is to find the similarities between files. Asm2Vec [57] leverages a modified version of the PV-DM model to solve the obfuscation and optimization issues in a clone search. Zuo et al. [58] and Redmond et al. [11] solve the binary similarity problem by NLP techniques when the same file is compiled in different architectures. SAFE [59] proposes a combination of skip-gram and RNN self-attention models 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 [60], [61], photo blending [62], video generation [63], 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 [64] of GANs is in the dialogue generation, where adversarial learning and reinforcement are applied together. SeqGAN [65] introduces gradient policy update with Monte Carlo search. LeakGAN [66] implements a modified policy gradient method to increase the usage of word-based features in adversarial learning. RelGAN [67] applies Gumbel-Softmax relaxation for training GANs as an alternative method to gradient policy update. SentiGAN [68] 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 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 an extensive data set consists of a million Spectre gadgets in four steps.

- **Step 1: Initial Data Set** There are 15 Spectre-V1 gadgets written in C by Kocher [26] and two modified examples introduced by *Spectector* [8]. For each example, a different attacker code is written to leak the entire secret data completely in a reasonable time.
- **Step 2: Compiler variants and optimization levels** Since our target data set is in assembly code format, each Spectre gadget written in C is compiled into x86 assembly functions 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.

Algorithm 1: Gadget generation using mutational fuzzing

Input: An Assembly function A, a set of instructions I_b and sets of registers R_b for different sizes of b

Output: A mutated Assembly function A'

```
G := R_b 7→ I_b
A' = A
MaxOffset = length(A)
for 1:Diversity do
    for Offset=1:MaxOffset do
        for 1:Offset do
            i_b ← random(I)
            r_b ← random(R_b|G)
            l ← random(0 : length(A'))
            Insert({i_b|r_b}, A', l)
        end
        Test boundary check(A')
        Test Spectre leakage(A')
    end
end
```

- **Step 3: Mutational fuzzing based generation**
  We generated new samples with an approach inspired by mutation-based fuzzing technique [69] 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, denoted as *Diversity*. We choose a random instruction i_b from the instruction set 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 are given in Appendix A.2.
- **Step 4: Verification of the generated gadgets**
  Finally, A' is tested whether it is still a Spectre-V1 gadget or not. There are two verification tests that are applied to the generated functions.
  The first verification test is applied to make sure that the function still has the proper array boundary-check for given user inputs. Since random instructions are inserted in random locations in the gadget, a new instruction may alter the flags whose value is checked by the original conditional jump. Once the flags are broken, the secret may be leaked without any speculative execution. To test this case, the PoC Spectre-V1 attacker code [1] is modified to supply only out-of-bounds inputs to A', which prevents mistraining the branch predictor. If the secret bytes in the PoC code are still leaked, we conclude that the candidate gadget is broken and exclude it from the pool.
  If a generated function A' passes from the first test, we apply the PoC Spectre-V1 attack to the gadget and exclude it if it does not leak the secret data through speculative execution. Additionally, the verification code is modified based on Kocher's examples since each example gadget leaks the secret in a different way. For instance, 4^{th} example shifts the user input by 1, which affects the leakage mapping in the cache. Therefore, we modified the PoC code to compile it with the generated gadgets together to leak the whole secret. This process is repeated for each example in Kocher's gadget dataset [26], which yields 16 different verification codes. The secret in the gadgets is only decoded via implementing the Flush+Reload technique. Other microarchitectural side-channels are not in the scope of the verification phase.
  Other Spectre variants such as SmotherSpectre [48] and NetSpectre [5] are not in our scope. Hence, the generated gadgets that potentially include SmotherSpectre and NetSpectre variants are not verified with other side-channel attacks. Our verification procedure only guarantees that the extracted gadgets leak secret information through cache side-channel attacks. The verification method can be adjusted to other Spectre variants, which is explained further in Section 6.

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

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 assembly language samples. The purpose of SpectreGAN is to develop an intelligent way of creating assembly functions instead of randomly inserting instructions and operands. Hence, the low success rate of gadget generation in the 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. SpectreGAN has a generator model that learns and generates x86 assembly functions and a discriminator model that 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) [22] which is composed of two-layer stacked LSTM units. Firstly, the input assembly functions are converted to a sequence of tokens $T^0 = \{x^0_1, ..., x^0_N\}$ 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 $r_m = \frac{1}{N}\sum_{t=1}^{N} m_t$. $m(T^0)$ is the

1. The attacker codes for each example, the entire data set, SpectreGAN, and FastSpec code are available at https://github.com/vernamlab/FastSpec

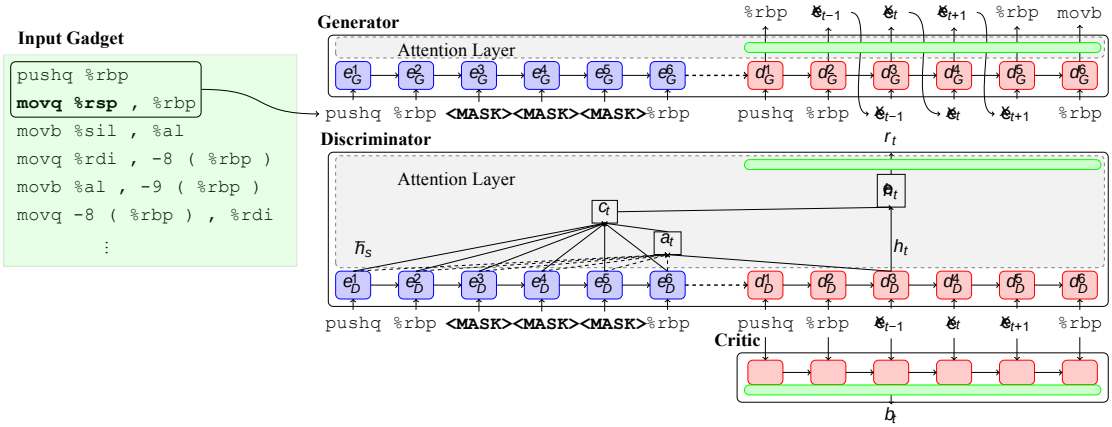

Figure 1: SpectreGAN architecture. Blue and red boxes represent the encoder and decoder LSTM units, respectively. Green boxes represent the softmax layers. The listed assembly function (AT&T format) on the left is fed to the models after the tokenization process. The critic model and the decoder part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

modified sequence where x_t^0 is replaced with <MASK> token for the corresponding positions of $m_t = 1$. Both T^0 and $m(T^0)$ 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 $\bar{h}_s$ which is also given as an input to the next encoder unit. The last encoder unit (e_G^6 in Figure 1) produces the final hidden state which encapsulates the information learned from all assembly tokens.

The decoder state is initialized with the encoder's final hidden state, and the decoder LSTM units are fed with $m(T)$ at each iteration. To calculate the hidden state $\tilde{h}_t$ 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 [70] 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:

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

where a_t describes the weights of $\bar{h}_s$, for a token x_t^0 at time step t, where $h_t^\top \bar{h}_s$ is the score value between the token x_t^0 and T^0. 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 $\bar{h}_s$ as follows:

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

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

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

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 $\hat{T} = (\hat{x}_1, ..., \hat{x}_N)$ is then generated by filling the masked positions for $m(T^0)$ where $m_t = 1$. The probability distribution $p(y_t | y_{1:t-1}, x_t)$ is calculated as,

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

where y_t is the output token and attention-based hidden state $\tilde{h}_t$ 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 to 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 $m(T^0)$, where we replace x_t^0 with <MASK> when $m_t = 1$, are fed into the encoder. The hidden vector encodings $\bar{h}_s$ and the encoder's final state are given to the decoder.

The LSTM units in the decoder are initialized with the final hidden state of the encoder and $\bar{h}_s$ is given to the attention layer. The list of tokens $\hat{T}$ 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 $\tilde{h}_t$ stated in Equation 2, 3, and 4 are valid for the attention layer in the discriminator model as well. The attention-based state value $\tilde{h}_t$ is fed through the softmax layer which outputs only one value at each time step t,

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

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 the 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 $\hat{x}$ with the decoder. The output of the LSTM unit at each time step t is given to the softmax layer, and we obtain

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

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^0 and masked version $m(T^0)$ are fed into the generator model's encoder. Only the real token sequence T^0 is fed into the decoder using *teacher forcing* in the pre-training. The training maximizes the log-probability of generated tokens, $\hat{x}_t$ given the real tokens, x_t^0, where $m_t = 1$. Therefore, the pre-training objective is

$$\frac{1}{N} \sum_{t=1}^{N} \log p(m(\hat{x}_t) | m(x_t^0)), \quad (8)$$

where $p(m(\hat{x}_t)|m(x_t^0))$ is calculated only for the masked positions. The masked pre-training objective ensures that the model is trained for a *Cloze* task [71].

Adversarial training phase. The second phase is 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 [18], [65].

The reward r_t for a generated token $\hat{x}_t$ is calculated as the logarithm of $p_D(\hat{x}_t = x_t^{real} | \hat{x})$. 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 $b_t = \log p_C(\hat{x}_t = x_t^{real} | \hat{x})$ is the baseline rewards by the critic model. Subtracting b_t from R_t helps reducing the variance of the gradient [18].

$$\nabla_\theta E_G[R_t] = (R_t - b_t) \nabla_\theta \log G_\theta(x_t) \quad (9)$$

To train the discriminator model, both real sequence T and fake sequence $\hat{x}$ are fed into the discriminator. Then, the model parameters are updated such that $\log p_D(\hat{x}_t = x_t^{real} | \hat{x})$ 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 to convert the assembly functions into sequences of tokens, $T^0 = (x_1^0, ..., x_N^0)$. We keep commas, parenthesis, immediate values, labels, instruction and register names as separate tokens. To decrease the complexity, we reduce the tokens' vocabulary size 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^0. Hence, each token list $T^0 = \{x_1^0, ..., x_N^0\}$ represents an assembly function in the data set.

The 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 first token's mask 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 [18]. We select the sequences with a maximum length of 250 tokens, building 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 masking rate choice is of utmost importance to generate high-quality gadgets. Typically, NLP-based generation techniques are evaluated with their associated perplexity score, which indicates how well the model predicts a token. 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.

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 the *GCC* compiler and then tested them with all the attacker code to verify their secret leakage. When SpectreGAN is trained with a masking rate of 0.3, the success rate of gadgets increases by up to 72%. Interestingly, the success rate drops for other masking rates, demonstrating 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 gadget diversity in Section 4.3.

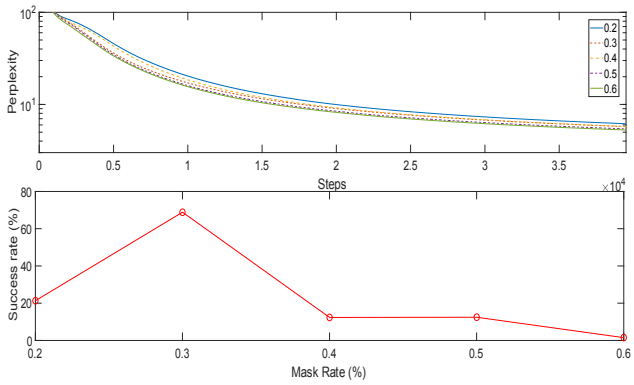

Figure 2: (Above) The validation perplexity decreases at each training step and converges for all r_m. (Below) Spectre gadget success rates are evaluated when different masking rates are used to train SpectreGAN. Spectre gadget success rate shows the percentage of gadgets out of compiled functions.

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.

```
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 Spectre-V1 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 and Quality Analysis of Generated Gadgets

In total, 1.2 million gadgets are generated by the mutational fuzzing technique and SpectreGAN. Since the gadgets are derived from existing examples, it is crucial to analyze their diversity and quality. The diversity is measured by syntactic analysis, e.g., counting the number of unique n-grams in gadgets. For the quality metric, we monitor performance counters while the gadgets are executed. 5000 gadgets are randomly selected from each gadget generation technique to perform syntactic and microarchitectural analysis. Furthermore, novel gadgets that are not detected by *oo7* [6] and *Spectector* [8] tools are given to show that our gadget generation techniques produce meaningful Spectre-V1 gadgets.

4.3.1. Syntactic Analysis. In NLP applications, the diversity of the generated texts is evaluated by counting the number of unique n-grams. The most common metrics for the text diversity are perplexity and BLEU scores that are calculated based on the probabilistic occurrences of n-grams in a sequence. The higher number of n-grams indicates that an NLP model learns the data set distribution efficiently and produces new sequences with high diversity. However, both scores are obtained during the training phase; thus, making it impossible to evaluate the fuzzing generated gadgets since there is no training phase. Instead, we conduct diversity analysis by counting the unique n-grams introduced by fuzzing and SpectreGAN methods after all the gadgets are generated.

The number of unique n-grams in generated gadgets is compared with 17 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, additional unique n-grams introduced by SpectreGAN 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.

TABLE 1: Table shows the number of unique n-grams for base gadgets and generated gadgets by fuzzing and SpectreGAN methods. In the last column the total number of unique n-grams are given as well as the increase factor that improves with the increasing n-grams.

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

In total, the number of unique bigrams (2-grams) is increased to 22,910 from 2,069, which is more than 10 times raise. While new instructions and registers added by fuzzing improve the gadgets' diversity, SpectreGAN contributes to the gadget diversity by producing unique perturbations. Since the instruction diversity increases drastically compared to base gadgets, the unique 5-grams reach up to almost 2 million, 400 times higher than the base gadgets. The results show that both fuzzing and SpectreGAN span the diversity in the generated gadgets. High diversity in the gadget data set also results in microarchitectural behavior diversity as well as new Spectre-V1 gadgets that were not previously considered during the design process of previous detection mechanisms.

4.3.2. Microarchitectural Analysis. Another purpose of gadget generation is to introduce new instructions and operands to create high-quality gadgets. To assess the quality of the gadgets, we analyze gadgets' microarchitectural characteristics. The first challenge is to examine the effects of instructions in the transient domain since they are not visible in the architectural state. After carefully analyzing the performance counters for Haswell architecture,

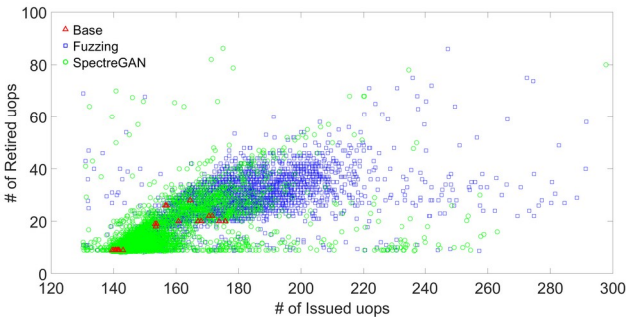

Figure 3: The distribution of base (red-triangle), fuzzing generated (blue-square) and SpectreGAN generated (green-circle) gadgets is given for issued and retired μops counters. Both SpectreGAN and fuzzing techniques generate diverse set of gadgets in Haswell architecture.

we determined that two counters, namely, *uops_issued* : *any* and *uops_retired* : *any* give an insight into gadgets' microarchitectural behavior. *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 performance counter distribution of generated gadgets and base gadgets are given in Figure 3. The gadget quality is measured by the number of instructions in the transient domain after a gadget passes the verification step. The exploitable gadgets in the commercial software have many instructions that are speculatively executed until the secret is leaked. If our detection tool in Section 5 is only trained with simple gadgets from Kocher's examples, the success rate would be low in large-scale software binaries. Moreover, the gadgets that are detected in the case studies are very similar to the generated gadgets which have more instructions in the transient domain. A similar observation is also shared in [72], where the authors claim that Spectre gadgets have up to 150 instructions between the conditional branch and speculative memory access in the detected gadgets. Since our aim is to create realistic gadgets by inserting various instructions, we assume that gadget quality increases in parallel when a gadget is close to the x-axis and far from the y-axis.

It is more likely to obtain high-quality gadgets with fuzzing method as new instructions and operands 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 the x-axis in Figure 3. Moreover, the advantage of SpectreGAN is to automate the creation of gadgets with a higher accuracy (72%) compared to the fuzzing technique (5%).

4.3.3. Detection Analysis. Even though the microarchitectural and syntactic analyses show that fuzzing and SpectreGAN can produce diverse and high-quality sets of gadgets, we aim to enable a comprehensive evaluation of detection tools and determine the most interesting gadgets in our data set. For this reason, the generated gadgets are fed into *Spectector* [8] and *oo7* [6] tools to determine the novelty of the gadgets.

oo7 tool leverages taint analysis to detect Spectre-V1 gadgets. It is based on the Binary Analysis Platform (BAP) [73] which forwards taint propagation along all possible paths after a conditional branch is encountered. *oo7* [2] is built on a set of hand-written rules which cover the existing examples by Kocher [26]. 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 a 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 gadgets as novel Spectre-V1 gadgets listed in Listing 4 and Listing 5. Their corresponding assembly snippets are also given in Appendix A.1.

```
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.

```
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, that is controlled by the attacker, is used to leak the secret in the Spectre gadget, *oo7* cannot detect the XCHG gadget. This example show that control-flow graph extraction is not efficiently implemented in *oo7* tool.

We identified two potential issues of static taint analysis method in *oo7* tool. First, if a portion of a tainted variable is modified by an instruction such as *cmov* or *set*, the tainted variable is not tracked by the tool. However, an attacker still controls the remaining portion of the variable, which makes it possible to leak the secret from memory. In some cases, the implementation of static taint analysis is not sufficiently accurate to track partially modified tainted variables due to under-tainting. Secondly, the tainted variables are not tracked between the iterations of a loop. If an old attacker-controlled variable is used to access the secret, *oo7* tool is not able to taint the old variable between the iterations of a *for* loop. Hence, any old attacker-controlled variable can be used to bypass the tool. This shows that control flow graphs of multiple iterations may not be extracted correctly by *oo7*. Both weaknesses show that hand-written rules do not generalize well for Spectre gadget detection when new Spectre-V1 gadgets are discovered.

Spectector [8] makes use of a symbolic execution technique to detect the potential Spectre-V1 gadgets. For

2. https://gitlab.com/igoto/spectre-detector

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.

First, we eliminate the gadgets that include unsupported instructions as these gadgets are never detected by *Spectector*. When we analyze the remaining gadgets, 1% of the gadgets are not detected successfully. Then, undetected gadgets are examined to determine novel gadgets.

We determined two issues in the *Spectector* tool. The first issue is related to the barrier instructions. Even though *lfence*, *sfence* and *mfence* instructions have different purposes, the tool treats them as equal instructions. For instance, if an *sfence* instruction is present after the conditional branch, the tool classifies the gadget as safe. However, *sfence* instruction has no effect on the load operation so, the gadget still leaks the secret. Hence, Spectector's modeling of fences does not distinguish the differences between various x86 fence instructions. The second issue is about 8-bit registers in which a partial information of the elements in *array[x]* is stored. When 8-bit registers are used to modify the elements in Listing 6, *Spectector* is no longer able to detect the gadgets. This second issue is also mentioned in [8], i.e., sub-registers are currently not supported by the tool. Overall, these issues are due to the problems in the translation from x86 assembly into Spectector's intermediate language.

We show that our large-scale diverse gadget data set establishes a ground truth to evaluate the detection tools accurately. As shown in the case studies on *Spectector* and *oo7*, the success rate on detecting the gadgets in our 1.1 million sample data set could serve as a generic evaluation metric while identifying the flaws in the detection tools.

```
1  victim_function:
2      movl    size(%rip),%eax
3      cmpq    %rax, %rdi
4      jae     .B1.2
5      movzbl  array1(%rdi)%eax
6      shlq    $9, %rax
7      xorb    %al, %al
8      movb    array2(%rax)%dl
9      andb    %dl, temp(%rip)
10 .B1.2:
11     ret
```

Listing 6: *xorb %al, %al* is added to 1st example in Kocher's examples [26]. *Spectector* is no longer able to detect the leakage due to the zeroing %al register.

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 BERT version.

5.1. Training Procedures

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

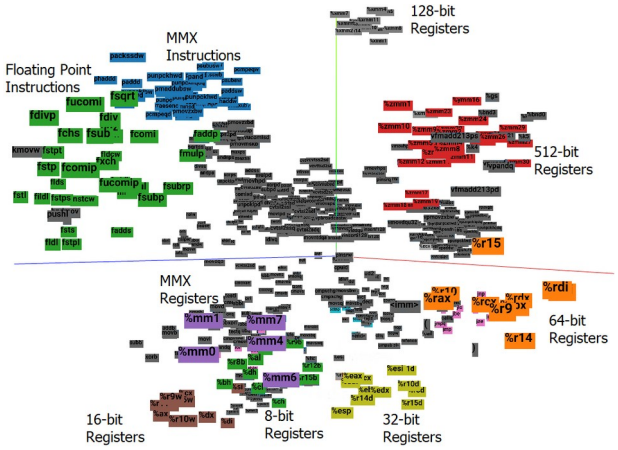

Figure 4: 3-D visualization for the distribution of instructions and registers after t-SNE is applied to embedding vectors. Similar instructions and registers have the same colors. The unrelated instructions are separated from each other in the three-dimensional space after the pre-training.

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. 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 other tokens' context, 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 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 impossible to visualize the high dimensional embedding vectors, we leverage the t-SNE algorithm [74] 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 types of tokens enables achieving a higher success rate in the 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 created for the pre-training phase are merged into a single sequence with a maximum 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 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 generated in Section 4 and the disassembled Linux libraries to train FastSpec. Although Linux libraries may contain Spectre-V1 gadgets, we assume that the total number of hidden Spectre gadgets is negligible, comparing the data set's total size. Therefore, the model treats those gadgets as noise, which does not affect 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 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 choose 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-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 classify 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 [20] 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, demonstrating the high performance of FastSpec. We assume that the false positives are Spectre-like gadgets in Linux libraries, which need to be explored deeply in future work. Moreover, we only have 55 false negatives (99.9% recall rate), which yield a 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: OpenSSL Analysis

We analyze FastSpec to validate with a separate ground truth data set other than the one we generate

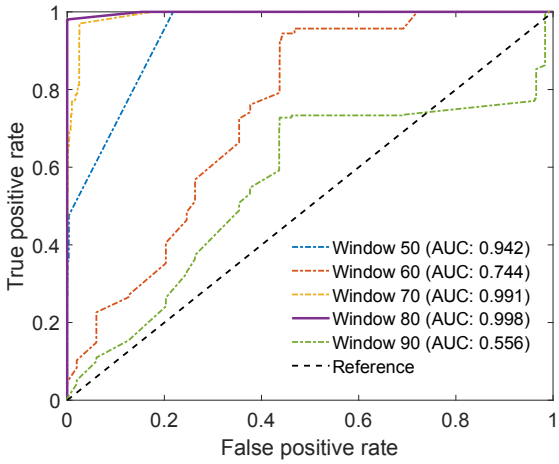

Figure 5: Solid line stands for the ROC curve of FastSpec for Spectre gadget class. Dashed line represents the reference line.

in Section 4. The purpose of this analysis is to measure the effect of the covariate shift and robustness of FastSpec against a real-world benchmark. We focus on OpenSSL v3.0.0 libraries [75], as it is a popular general-purpose cryptography library in commercial software. We use a subset of functions from RSA, ECDSA, and DSA ciphers in the OpenSSL *speed* benchmark. The function labels are obtained by running the *SpecFuzz* tool, which is a dynamic detection tool to find Spectre-V1 vulnerabilities using fuzzing [54]. The functions in which the *SpecFuzz* tool finds vulnerabilities are labeled as positive, and the remaining ones are labeled as negative. We also exclude the functions without any conditional branch instructions from the positive class and the functions that have a call to them. In total, 4242 functions are extracted from the aforementioned cryptography libraries to analyze with FastSpec. Positive and negative classes include 720 and 2500 functions, respectively.

First, we apply the same pre-processing procedures, as explained in Section 5.2 to obtain the tokens. The total number of tokens is more than 4 million. Since the labels are assigned on function-level, we choose the maximum confidence rate that we get among all the sliding windows. The maximum confidence rate is assigned as the prediction of our model for the corresponding input function. In order to find the optimal sliding window size, we scan through the functions with various different window sizes and compare the performances. Figure 5 shows that FastSpec achieves the highest performance to detect functions with Spectre-V1 vulnerability when the window size is set to 80 tokens, which corresponds to 0.998 as an area under the curve (AUC) value. The optimal threshold value is found as 0.48, which corresponds to the maximum F-score. The highest F-score is achieved as 0.99, where the false positive rate (benign functions that are mistakenly classified as Spectre gadget) is 0.04%, and false negative rate (functions that are mistakenly classified as benign) is 2%. We claim that further analysis of the detected functions by using symbolic execution or taint analysis tools can reduce the number of false negative samples and provide an efficient end-to-end security solution against Spectre-V1 vulnerability.

5.4. Case Study: Phoronix Test Suite Analysis

The performance comparison between FastSpec and other static analysis tools is evaluated on the Phoronix Test Suite v5.2.1 [76]. For the ground truth, the *SpecFuzz* technique is chosen as the tool that dynamically analyzes the binaries, and exploitable gadgets can be detected with a higher success rate compared to static tools. The selected benign files have source code since it is required to obtain the assembly files for the *Spectector* tool. The assembly files are generated by compiling the source C code with the *GCC* compiler. On the other hand, the binary files are generated at 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 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 observe that the pre-processing phase takes the major portion in the analysis time of FastSpec while the inference time is in the order of microseconds. We fuzz the Crafty benchmark for 24 hours and other benchmarks for 1 hour using *SpecFuzz* under the default configuration [3].

The effect of the increasing number of branches on time consumption is clear in the Crafty and Clomp benchmarks in Table 2. Even though the 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 6, 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 analyzes the same Crafty benchmark under 6 minutes, which is a significant improvement.

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

Baseline Evaluation 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. To be consistent, if a control or data leakage is found in a function, it is reported as a vulnerable function by all three tools.

The precision and recall rates for *oo7*, *Spectector* and FastSpec are given in Table 2. The precision is calculated as $TP/(TP + FP)$. TP is the number of overlapping gadgets detected by a tool. FP is the number of functions that are classified as Spectre gadgets mistakenly. The recall value is computed as $TP/(TP + FN)$ where FN is the number of gadgets that are not detected by a tool.

3. https://github.com/OleksiiOleksenko/SpecFuzz

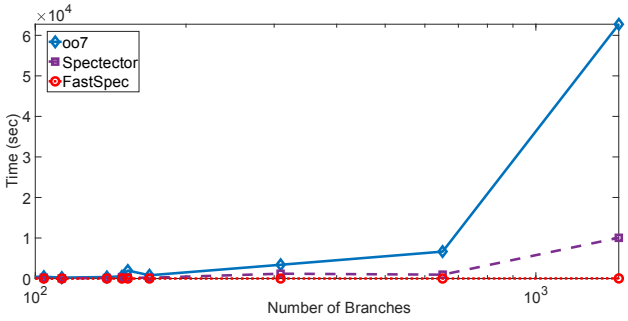

Figure 6: The processing time of FastSpec is independent of the number of branches whereas for Spectector and oo7 the analysis time increases drastically.

In some cases, *oo7* is not able to track the control flow when the number of function calls increases in a gadget, which yields high false negatives and low recall. Thus, *oo7* suffers from the extraction of complete control flow graph. *Spectector* tends to give more false positives compared to *oo7* and FastSpec. This is because some unsupported instructions are skipped by the tool and the broken Spectre gadgets by specific instructions are still classified as Spectre gadget. On the other hand, FastSpec has low false negatives since all the Spectre gadget patterns are detected with a confidence rate higher than 0.48. When the file size increases, the false positives may increase in parallel. However, these gadgets can be verified with other tools to increase the confidence. As a result, FastSpec scans the functions extremely quicker than other tools without sacrificing the precision and recall rates. Our tool also guarantees the security of the scanned assembly functions by detecting almost all Spectre gadgets with low FN rates. FastSpec outperforms all the compared tools in terms of recall and precision rates by a large margin.

6. Discussion and Limitations

6.1. Gadget Verification

The gadget verification process in Section 4.1 is implemented in an isolated core since the system interrupts frequently mistrain the targeted branch instructions in the gadgets, which decreases the gadget verification success rate significantly. This situation particularly affects the first step of the verification process where all the inputs are out-of-bounds, and the target branch is not expected to be mistrained. Therefore, there is a need for an isolated environment to run the verification code for Spectre gadgets. Even though the data cache side-channel is used for the secret decoding, other side-channels can be used to decode the secret in a Spectre gadget such as TLB structure. The secret elements in *array1* should be multiplied with a constant to decode the secret into different cache lines or pages. In the base examples [26], the secret elements are multiplied by 512 or 4096. The verification code only selects the Spectre gadgets with these specific multiplicands, which potentially introduces a bias in the data set. Since all multipliers in the Spectre gadgets are

TABLE 2: Comparison of *oo7* [6], Spectector [8], 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*. (#CB: Number of conditional branches, #Fc: Number of functions, #DFc: Number of detected functions)

Benchmark	Size (KB)	#CB	#Fc	SpecFuzz	oo7			Spectector			FastSpec		
				#DFc	Precision	Recall	Time (sec)	Precision	Recall	Time (sec)	Precision	Recall	Time (sec)
Byte	183.5	363	83	7	0.70	**0.90**	400	1.00	0.43	115	**1.00**	0.86	**14**
Clomp	79.4	1464	45	1	0	0	17.5 hr	0.05	0.9	2.8 hr	**1.00**	**1.00**	**35**
Crafty	594.8	10796	207	44	**1.00**	0.54	> 10 day	0.60	**0.91**	48 hr	0.23	0.80	**315**
C-ray	27.2	139	11	1	**1.00**	1.00	395	0.2	0.9	153	0.50	**1.00**	**8**
Ebizzy	18.5	104	6	3	0	0	467	0.60	1.00	206	**1.00**	0.33	**3**
Mbw	13.2	70	5	1	0	0	145	**0.50**	1.00	34	0.33	**1.00**	**2**
M-queens	13.4	51	4	1	1.00	1.00	136	0.50	1.00	24	**1.00**	**1.00**	**2**
Postmark	38.0	309	49	6	1.00	0.83	3409	0.43	0.95	1202	**1.00**	**1.00**	**10**
Stream	22.0	113	4	3	0	0	231	0	0	63	**1.00**	**0.66**	**4**
Tiobench	36.1	169	19	1	0	0	813	0.25	0.8	201	**0.33**	**1.00**	**9**
Tscp	40.8	651	38	13	0	0	6667	1.00	0.15	972	**1.00**	**0.92**	**12**
Xsbench	27.9	153	32	1	**1.00**	**1.00**	1985	0	0	249	0.50	0.90	**7**
Average					0.47	0.44		0.43	0.67		**0.74**	**0.87**	

represented with the same token, `<imm>`, our detection tool is not affected by the bias introduced by different multipliers. For instance, in OpenSSL and in Phoronix, we observed that gadgets with different multiplicands are detected by our detection tool.

Our verification codes also focus on more complex leakage snippets in which the secret is not simply leaked with a simple multiplication. We observed that similar control-flow statements and more complex encoding techniques are present among Kocher's examples [26] (Examples 10-15). After new gadgets are generated from these examples, we observed that these gadgets can still be detected by our verification code. However, if the leakage mechanism in the gadget is altered significantly, it is likely that the secret in the generated gadget is not recovered during verification. Unfortunately, this introduces a bias in our data set as the diversity of the gadgets is limited. Moreover, our detection tool might not be able to detect more complex gadgets as these gadgets are not included in the training data set. To include more complex gadgets in the data set, the verification code can be changed dynamically by analyzing each generated assembly code, which is left as future work.

6.2. Scalability and Flexibility

Other Spectre Variants: Since pre-training teaches the general assembly syntax and takes a major part in the training process, our pre-trained FastSpec model can be used after fine-tuning for any assembly code task. The modifications are needed only to Step 1 and Step 4 in Section 4.1 since we need an initial data set and verification code to build up a larger data set. For Spectre v1.1 [34], our verification code can be adapted by adding one more attacker-controlled input to verify whether a speculative load is executed or not. Similarly, the speculatively written value in Spectre v1.2 [34] can be mapped to cache lines to verify the generated gadgets. For Spectre v2 [1], verification procedure needs to be completely changed as the branch instruction is not a conditional branch anymore. For this purpose, the verification code can be modified to mistrain the indirect jumps with attacker known addresses, and then, the secret bytes in the attacker-controlled function are mapped to separate cache lines. Since Spectre-RSB [3] works in a similar way, except *ret* instruction is targeted, the same verification procedure can be adapted. Finally, in Spectre v4 [33], the verification code can supply attacker-controlled variables to specific registers, and then, speculatively loaded data can be decoded to a shared memory to verify the gadgets.

Other Attacks: Our approach can detect the target `SMoTher`-gadgets [48] in the code space. The verification procedure in Section 4.1, specifically Step 4, needs to be changed to analyze port fingerprints. For this purpose, the timing of various instructions that are mapped to certain ports can be measured to detect the leaked secrets as implemented in [77]. It is highly likely that the verification takes more time for the generated gadgets since we need to collect more timings to distinguish the cases between secret leakage and no secret leakage. In NetSpectre [5], there are two types of gadgets. The leak gadget is very similar to Spectre v1 whereas only one bit is transmitted. Hence, the verification procedure can be modified to profile a single cache line instead of 256 cache lines. The transmit gadget is used to leak the secret data over the network and has a different structure than the leak gadget. To detect the transmit gadgets with our verification code, the Thrash+Reload technique can be adapted to measure the timing difference between cached and non-cached accesses over the network. Again, the verification procedure potentially takes more time to analyze the generated gadgets since the secret transmission speed is significantly lower than Spectre V1.

Other Architectures and Applications: 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 [20]. 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.

The fuzzing tool increases the diversity of the generated gadgets by introducing variations that are later learned by the FastSpec tool. In addition, the detection

tool learns the generic gadget type rather than training on small details. In Section 5.2, the evaluation of FastSpec also shows that the tool can detect the potential Spectre gadgets with a 99.9% precision rate.

6.3. Comparison of FastSpec with Other Tools

The most significant advantage of FastSpec is the capability of detecting Spectre gadgets quicker than other tools. If an instruction is not introduced in the training phase, the instruction is treated as unknown, and it has a slight effect on the accuracy of FastSpec since a large window of instructions is analyzed to decide on the Spectre gadgets. While the unsupported instructions are an important issue for the *Spectector* tool, FastSpec can be deployed to other architectures such as ARM and AMD. While small modifications in the assembly code increase the chance of bypassing other tools, our tool is more robust against small modifications. It is easier to adapt FastSpec to other Spectre variants as the vector representations of assembly instructions can be directly used to train a separate model for the variants. Moreover, over-tainting and under-tainting issues decrease the accuracy of taint-based static analysis techniques. However, FastSpec tracks the registers, instructions, and memory accesses with a vector representation, which makes it more reliable in large-scale projects.

6.4. Scope and Limitations

Scope: Our scope is to generate Spectre-V1 gadgets by using mutational fuzzing and SpectreGAN methods as well as to detect potential Spectre gadgets in benign programs by significantly reducing the analysis time.

Guarantees: Our verification methods in Step 4.1 guarantee that the generated Spectre-V1 gadgets leak the secret bytes through cache side-channel attacks. Moreover, the FastSpec tool detects the Spectre gadgets with a high precision and recall rate by identifying the gadget patterns at the assembly level. Possible False Positive outputs do not affect the security guarantee provided by FastSpec. The analysis time is significantly reduced compared to rule-based detection tools.

FastSpec generalizes well, i.e., it can recognize similar patterns that are not in our training dataset. However, it does not provide assurance of coverage (completeness) since FastSpec is not based on hand-written rules or formal analysis. In order to decrease the False Negative rate, the probabilistic threshold is kept low in the case studies. In contrast, while FastSpec does not provide such guarantees, it is much faster and scales to larger codebases.

Assembly Code Generation: The challenges faced in the regular text generation with GANs [18], [65] 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 reduce mode collapse, we observed that our generator model still generates some tokens or patterns of tokens repetitively, reducing the quality of the generated samples and 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 still be 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 position of a few instructions and registers.

The fuzzing-based gadget generation technique is based on known gadget examples. Since there are already 15 versions of Spectre-V1, we use these gadgets as the starting point for fuzzing. On the other hand, the available gadgets for other variants are significantly lower compared to Spectre-V1 gadgets. To solve this issue, other detection tools can be used to detect Spectre gadgets in benign programs. Then, new gadgets can be generated with fuzzing technique. We leave the further investigation of generation other Spectre variants as future work.

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 using lower window sizes than maximum sequence length provides more accurate Spectre gadget detection and provides fine-grain information on the sequence.

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 the 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 assembly language model, an entirely different domain than natural language. We demonstrate 72% of the compiled code snippets behave as a Spectre gadget, a massive improvement over fuzzing based generation. Furthermore, we show that our generated gadgets span the speculative domain by introducing new instructions and their perturbations, yielding diverse and novel gadgets. The most exciting gadgets are also introduced as new examples of Spectre-V1 gadgets. Finally, we propose FastSpec, 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 our anonymous reviewers for their valuable comments, *oo7*'s [6] author Ivan Gotovchits and *SpecFuzz*'s [54] author Oleksii Oleksenko for helping us running the *oo7* and *SpecFuzz* tools accurately. This work is supported by the National Science Foundation, under grant CNS-1814406, and in part by Intel Corporation.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[4] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 142–157.

[5] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*. Springer, 2019, pp. 279–299.

[6] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," *arXiv preprint arXiv:1807.05843*, 2018.

[7] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, "Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution," *arXiv preprint arXiv:1909.00647*, 2019.

[8] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sanchez, "Spectector: Principled detection of speculative information flows," in *IEEE Symposium on Security and Privacy*. IEEE, May 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/spectector-principled-detection-of-speculative-information-flows/

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[10] B. J. Radford, B. D. Richardson, and S. E. Davis, "Sequence aggregation rules for anomaly detection in computer network traffic," *arXiv preprint arXiv:1805.03735*, 2018.

[11] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," *arXiv preprint arXiv:1812.09652*, 2018.

[12] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 3111–3119.

[13] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[14] 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*, 2014, pp. 2672–2680.

[15] T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz, and B. Catanzaro, "High-resolution image synthesis and semantic manipulation with conditional gans," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8798–8807.

[16] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 2642–2651.

[17] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee, "Generative adversarial text to image synthesis," *arXiv preprint arXiv:1605.05396*, 2016.

[18] W. Fedus, I. J. Goodfellow, and A. 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. [Online]. Available: https://openreview.net/forum?id=ByOExmWAb

[19] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in neural information processing systems*, 2017, pp. 5767–5777.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. 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)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[22] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[23] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[24] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[26] P. Kocher, ""spectre mitigations in microsoft's c/c++ compiler"," 2018. [Online]. Available: [Online].Available:https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[27] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree at https://transient.fail/.

[28] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *CCS*, 2019.

[29] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[30] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," 2018.

[31] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. 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.

[32] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[33] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[34] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018.

[35] G. Maisuradze and C. Rossow, "ret2spec," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jan 2018. [Online]. Available: http://dx.doi.org/10.1145/3243734.3243761

[36] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," 2018.

[37] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," 2019.

[38] A. Pardoe, "Spectre mitigations in msvc," Jan. 2018. [Online]. Available: [Online].Available:https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[39] P. Turner, ""retpoline: a software construct for preventing branch-target-injection."," 2018. [Online]. Available: [Online].Available: https://support.google.com/faqs/answer/7625886

[40] C. Carruth, "Rfc: Speculative load hardening (a spectre variant 1 mitigation)," 2018. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html

[41] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[42] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[43] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[44] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 387–401.

[45] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *IFIP International Information Security Conference*. Springer, 2005, pp. 295–307.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[47] M. Qiu, F.-L. Li, S. Wang, X. Gao, Y. Chen, W. Zhao, H. Chen, J. Huang, and W. 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)*, 2017, pp. 498–503.

[48] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[49] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," *arXiv preprint arXiv:2003.05503*, 2020.

[50] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "Context: Leakage-free transient execution," *arXiv preprint arXiv:1905.09100*, 2019.

[51] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. 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*, 2019, pp. 954–968.

[52] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo, "Specusym: Speculative symbolic execution for cache timing leak detection," *arXiv preprint arXiv:1911.00507*, 2019.

[53] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 747–761.

[54] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," *arXiv preprint arXiv:1905.10311*, 2019.

[55] N. Nethercote and J. Seward, "Valgrind: a framework for heavy-weight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[56] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

[57] S. H. Ding, B. C. Fung, and P. 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)*. IEEE, 2019, pp. 472–489.

[58] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

[59] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[60] 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.

[61] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4401–4410.

[62] H. Wu, S. Zheng, J. Zhang, and K. Huang, "Gp-gan: Towards realistic high-resolution image blending," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 2487–2495.

[63] C. Vondrick, H. Pirsiavash, and A. Torralba, "Generating videos with scene dynamics," in *Advances in neural information processing systems*, 2016, pp. 613–621.

[64] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, and D. Jurafsky, "Adversarial learning for neural dialogue generation," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 2157–2169. [Online]. Available: https://www.aclweb.org/anthology/D17-1230

[65] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, ser. AAAI'17. AAAI Press, 2017, p. 2852–2858.

[66] J. Guo, S. Lu, H. Cai, W. Zhang, Y. Yu, and J. Wang, "Long text generation via adversarial training with leaked information," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[67] W. Nie, N. Narodytska, and A. Patel, "Relgan: Relational generative adversarial networks for text generation," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=rJedV3R5tm

[68] K. Wang and X. Wan, "Sentigan: Generating sentimental texts via mixture adversarial networks." in *IJCAI*, 2018, pp. 4446–4452.

[69] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[70] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473

[71] W. L. Taylor, ""cloze procedure": A new tool for measuring readability," *Journalism Quarterly*, vol. 30, no. 4, pp. 415–433, 1953. [Online]. Available: https://doi.org/10.1177/107769905303000401

[72] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2019.

[73] T. A. D. Brumley, I. Jager and E. J. Schwartz, "Bap: A binary analysis platform," p. 463–469, 2011.

[74] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[75] T. O. Project, "Openssl v3.0.0," 2021. [Online]. Available: https://github.com/openssl/openssl

[76] P. media, "Open-source, automated benchmarking," 2021.

[77] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

Appendix A.

A.1. Assembly Gadget Examples

In this section, corresponding assembly gadget of given examples in Section 4 are provided.

```
victim_function:
.LFB23:
        movl    global_condition(%rip),%eax
        testl   %eax, %eax
        movl    $0, %eax
        cmovne  %rax, %rdi
        movslq  array_1_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 7: When the C code in Listing 4 compiled with certain optimizations (gcc 7-4 with O2 enabled), the generated assembly code contains CMOV instruction which fools *oo7*.

```
victim_function:
        xchg    %rdi, %r13
        cmpl    %esp, %esp
        movl    array_1_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 8: 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).

```
victim_function:
        seta    %sil
        cmpl    $0, (%rsi)
        jle     .LBB0_2
        addl    %r15d, %r12d
        sarq    $1, %r11
        addb    %sil, %r15b
        movzbl  array1(%rdi),%eax
        ja      .L13243337
        testw   %r10w, %ax
        shlq    $12, %rax
        nop
        movb    array2(%rax),%al
.L13243337:
        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 is not detected by *oo7*.

A.2. 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