TrustZoneTunnel: A Cross-world Pattern History Table-based Microarchitectural Side-channel Attack

Tianhong Xu
Northeastern University
xu.tianh@northeastern.edu

Aidong Adam Ding
Northeastern University
a.ding@northeastern.edu

Yunsi Fei
Northeastern University
y.fei@northeastern.edu

2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-7394-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/HOST55342.2024.10545376

***Abstract*—ARM's TrustZone is a hardware-based trusted execution environment (TEE), prevalent in mobile devices, IoT edge systems, and autonomous systems. Within TrustZone, security-sensitive applications reside in a hardware-isolated secure world, protected from the normal-world's applications, OS, debugger, peripherals, and memory. However, microarchitectural side-channel vulnerabilities have been discovered on shared on-chip resources, such as caches and branch prediction unit (BPU). In this paper, we propose TrustZoneTunnel, the first Pattern History Table (PHT)-based side-channel attack on TrustZone, which is able to reveal the complete control flow of a trusted application in the secure world. We reverse-engineer the PHT indexing for ARM processors and develop key primitives for cross-world attacks, including well-controlled world-switching, PHT collision construction between two worlds, and precise PHT state-setting and checking functions. Furthermore, we introduce a novel model extraction attack against TrustZone based deep neural network, which can recover model parameters using only the side-channel leakage of vital branch instructions, obviating the need for model output or logits while prior research work requires such knowledge for model extraction.**

***Index Terms*—ARM TrustZone, Side Channel Attack, Branch Prediction**

I. Introduction

With ever increasing requirements for security and trust of the running applications, many modern CPUs are companioned with a Trusted Execution Environment (TEE). By executing security-sensitive applications in an isolated environment, called secure enclave or secure world, TEEs dedicate separate resources to the secure applications and disallow access by untrusted applications or even OS from the rest of the system (usually called host or normal world) [24]. TEEs can protect the confidentiality of valuable code and data as well as integrity of the system. Intel's Software Guard Extensions (SGX) and ARM's TrustZone are two common TEEs that are widely used in modern computing systems, while Intel has dis-continued the support for SGX on client machines from its 12th-generation core. ARM TrustZone remains the most popular TEE found in billions of mobile systems, edge and IoT devices. In TrustZone, the secure world and normal world are two software worlds with separate hardware components and access mechanisms, including debugger, peripherals, and memory.

However, recent research demonstrated that TEEs are vulnerable to microarchitectural side-channel attacks [10], [14], [16], [18], [23], as many on-chip microarchitectural units are shared across the worlds, including caches and branch prediction unit (BPU). Through shared resources, an adversary in the normal world can glean details of a critical application in the secure world, breaking the protection provided by the TEE. On Intel SGX platforms, various microarchitectural side-channel attacks have been presented that exploit cache, BPU, and page table [8], [13], [14], [28]. On ARM TrustZone, cache-based and BPU-based side-channel attacks are also proposed [10], [16], [17], [23]. In these cross-world/enclave attacks, the OS is not trusted and the attacker can have kernel-privilege but still cannot access the secure hardware protected by the TEE. With the shared microarchitecture for which applications in both worlds can set the state and also monitor the change, the attacker in the normal world can bypass the protection of the secure world and retrieve critical information about the victim application.

Previous side-channel attacks on Intel BPU target different microarchitecture components. Branch target buffer (BTB)-based attacks [2], [3] were first introduced, exploiting the shared BTB across processes to infer a victim application's control flow and confidential data. There are already several countermeasures against BTB attacks, adopting the principle of isolation, including a hardware-level mechanism to partition BTB entries among processes [30] and a complier-assisted protection in Half&Half [29]. The other BPU component, Pattern-History-Table (PHT) for predicting the direction of branches' execution, has received less attention. The prior work, including Branchscope [12] and BlueThunder [14], both target simple PHT architectures with Intel SGX enabled. Most of the prior attacks and countermeasures primarily target Intel systems, with very few addressing ARM architectures. This oversight is significant given the substantial differences between ARM's and Intel's BPU and TEE designs. The only BPU based attack on ARM's platforms was hardware-backed Heist [23] that implements a BTB-based attack, while currently there is no PHT-based attack targeting ARM TrustZone.

In this paper, we propose TrustZoneTunnel, a cross-world PHT-based side-channel attack targeting ARM Processors. We reverse-engineer the PHT indexing mechanism of Cortex-A53, one of the most widely used processor for mobile and embedded devices since 2014. We construct collisions on a complex TAGE PHT, employ the Load-Step mechanism [16], a method to control the world-switching in TrustZone with adjustable resolution, and propose a PHT-based microarchitectural side-

channel attack framework against ARM TrustZone. We further apply TrustZoneTunnel onto TrustZone based secure deep neural network implementation for model extraction [19]. The experimental results show that TrustZoneTunnel can extract the direction of any single target branch of a secure world program, and can successfully retrieve the entire DNN model.

II. Background

This section provides background on ARM TrustZone, the vulnerable shared microarchitecture BPU, and cross-world switching mechanisms to create contention on the shared microarchitecture.

A. ARM TrustZone

Since its introduction in 2004 with the ARMv6 architecture, ARM's TrustZone technology has been a key security feature in the ARM processor family. As the most popular TEE, TrustZone has been adopted in billions of lightweight devices to protect both confidentiality and integrity of sensitive code and data. With TrustZone, on the same processor there are two software worlds: secure and normal, where the secure world has its dedicated hardware resources and peripherals, ensuring sensitive data and applications to operate in a trusted environment, isolated from the regular operation modes in the normal world.

Starting from the Cortex-A8 architecture, the TrustZone technology has seen significant advancement. This work mainly targets Cortex-A53 processor [6], which has been the most widely adopted processor model in smartphones since 2014 until 2017 [1], and is still commonly used in game consoles and embedded devices at present. Cortex-A53 is also one of the first two ARM processors implementing ARMv8-A ISA, which pivoted to 64-bit cores from 32-bit. It embodies TrustZone features including robust isolation for secure data handling, integrated cryptographic support for enhanced data protection, and secure boot functionality for verified software execution.

B. Branch Prediction Unit (BPU)

Modern computer architectures utilize BPU to speed up the control flow of instruction streams. In a processor pipeline, when the Instruction Fetch Unit (IFU) fetches a branch instruction, the BPU in the processor front-end is looked up to predict the direction and/or the target address before the branch is resolved, as there is cycle delay in the branch instruction execution. The IFU then fills the pipeline with the predicted instruction and starts speculative execution. There is a mechanism to detect mis-speculation later and roll back the execution. The BPU usually comprises several integral components, each responsible for specific aspects of branch prediction. The Branch Target Buffer (BTB) predicts destinations of taken branch instructions. The Conditional Branch Predictor (CBP) gauges whether a branch will be taken or untaken (predicting the branch direction), crucial for performance optimization.

In modern processors, CBP is based on a Pattern History Table (PHT), a multi-entry table where each entry contains a saturation counter for the direction prediction. The index function of a PHT is based on the current branch address and the processor's execution context, provided by a Global History Register (GHR), which is a shift register recording the history of the previously executed branches. Previous work [29] shows that a GHR can be implemented in two different ways: a Direction History Register (DHR) that records the directions of prior conditional branches, and a Path History Register (PHR) that stores the address of prior taken branches. In each PHT entry, a 2-bit saturation counter specifies four states, 00 for strong not-taken (NT), 01 for weak NT, 10 for weak taken (T) and 11 for strong taken. When making direction prediction for a conditional branch, the PHT is looked up by the generated index to hit an entry, and the 2-bit state will predict the direction of the target branch (actually the upper bit of the state, 1 for T and 0 for NT). When the conditional branch instruction is resolved and finishes its execution, the real direction will update the counter's state, following a 4-state finite machine. Fig 1 shows the mechanism of an example PHT-based branch direction prediction, where the GHR is a DHR.

Some advanced BPUs adopt TAGE (TAgged GEometric history length) structure for branch prediction [22], [25], [26], where multiple PHTs are included, each is associated with a different history length. In this way the TAGE architecture can capture and exploit diverse execution patterns across various timescales, optimizing its overall prediction accuracy. By ensuring an appropriate PHT is selected for a given branch scenario.

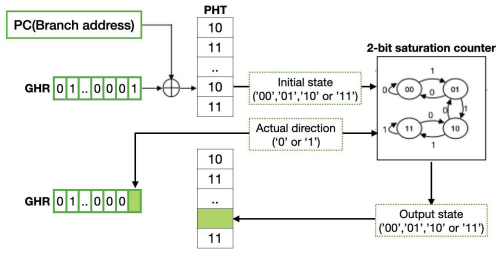

Fig. 1. The mechanism of PHT based branch direction prediction

C. Load-Step World-switching

For high-resolution switching between the adversary and the target branch execution, we exploit Load-Step [16], a method to periodically interrupt the secure-world victim program by the normal-world adversary program. Load-Step is implemented as a Linux kernel driver, where all the adversary's codes are embedded in the context-switching process by customizing the interrupt handler functions. The interrupt resolution can be per-instruction.

Fig 2 shows the procedure of the Load-Step world-switching mechanism adapted for our attack. It involves two cores, one is the auxiliary control core and the other is the victim core that runs the victim application in the secure world. The auxiliary core sets a timer and first receives a time-up event from the timer. It then controls the Generic interrupt controller (Arm-GIC) to generate a cross-core interrupt and

directs the interrupt to the victim core. The host OS is responsible for handling cross-core interrupts. Once the victim core receives the interrupt, the secure world is forced to save its context and the execution switches to the normal world. The attacker's malicious functions (PHT_Preset & PHT_Check) are embedded in the OS interrupt handling routine. When the interrupt handler finishes execution, a signal is sent to the auxiliary core to reset the timer for the next attack epoch, and meanwhile the victim core switches to the secure world to resume the victim application.

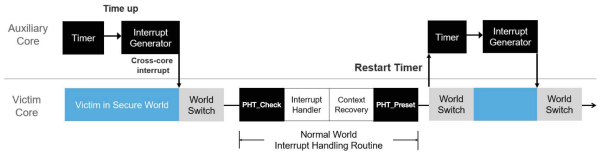

Fig. 2. Process of Load-Step

III. Reverse Engineering ARM BPU

To understand the vulnerable microarchitecture so as to build the side-channel attack, we conduct thorough reverse-engineering of the CBP of a 64-bit Cortex-A53 processor. These reverse engineering results facilitate building attack components, such as collisions on a specific PHT entry and manipulating the state of PHT entries.

A. An Overview of Cortex-A53's CBP

We use a Raspberry Pi 3B+ board, a widely-used ARM-based development board, in our experiments. It has a quad-core 1.2GHz ARM 64-bit Cortex-A53 processor and 1 GB RAM. We install OPTEE [21] on the system, a TEE designed as a companion to a Linux kernel running on ARM.

The only documented information about the CBP of Cortex-A53 is that it uses a Direction History Register (DHR) and a 3072-entry PHT [5], while other details such as the PHT indexing, the size of the DHR, and what bits of the branch address contribute to the index, are all unknown. In our experiments, we observed that the CBP has a TAGE structure. A typical TAGE predictor is shown in Fig 3, where there are three PHT tables: a base one which is only indexed by the branch addresses (T_0), and two tagged tables (T_1 and T_2) which are indexed with different GHR sizes (s_1 and s_2) in addition to the branch address, where each entry keeps a tag for hitting comparison and a usefulness counter (u) for replacement decisions based on temporal locality.

For TAGE predictors, all the tables are simultaneously queried during the prediction time, while only one predictor will be selected to perform the prediction. A tag is generated by a hash function over the branch address and the respective GHR for each of the two tagged predictors. If no matches in either table, i.e., the branch is not executed with the same histories the predictors have witnessed before, the base predictor (without tag) is used. When multiple tagged predictor tables have a matched tag for a branch, certain policy guides the selection of the appropriate predictor [22], [25], [26]. In

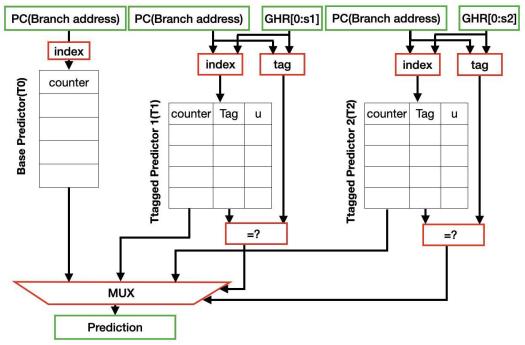

Fig. 3. The TAGE prediction structure

our experiments, we reverse engineered the predictor selection mechanism employed by ARM Cortex-A53: by default the predictor indexed with a shorter GHR (T_1) is chosen when both T_1 and T_2 have matching tags; however, the predictor with the longer GHR (T_2) is utilized when the recent prediction accuracy of T_2 is significantly higher than that of T_1.

Next in Section III-B, we first design microbenchmarks to determine the size of GHR for each tagged predictors (s_1 and s_2). In Section III-C, we reverse engineer the effective bits in a branch address for PHT indexing.

B. Measuring the Size of GHR

As the PHT in Cortex-A53 only has 3072 entries, we assume that the GHR has no more than 16 bits. We first design a function, shown below in Listing 1, to access the PHT by a target branch with a pre-set GHR.

```
1  Access_PHT(h, d){
2      if((h>>15)&1) {...}
3      if((h>>14)&1) {...}
4      ...                          //Training GHR
5      if((h>>0)&1) {...}
6      m0=misprediction_counter();
7      //read the event counter
8      if(d) {...}          //Target branch
9      m1=misprediction_counter();
10     return (m1-m0);
11 }
```

Listing 1. Function for accessing a PHT entry

The function has two input parameters, h, a 16-bit unsigned value to set the GHR for a target branch, and d, a one-bit direction for the target branch where 1 means Taken and 0 means Untaken. In the function, Line 8 is the target branch, and before it we put 16 conditional branches with pre-set conditions specified by the h value (Lines 2-6) to set the GHR value seen by the target branch. During execution, although each of the 16 branches also updates a respective PHT entry, it is not likely they index into the same PHT entry as the target branch, because they all have different addresses and have seen a different branch global history. Before and after the target branch execution, the mis-prediction counter is read, which is an event counter in the Performance Monitor Unit (PMU) that counts mis-predictions of conditional branches and indirect branches. The function returns the differential of the

two readings of the mis-prediction counters, with 1 indicating the target branch experiences a mis-prediction while 0 meaning a correct prediction. We hypothesize that the GHR size is smaller than 16 bits, therefore, the branch history closer to the the target branch (the lower part of the 16-bit h) is stored in GHR for the target branch, while the other upper bits do not contribute to the PHT index for the target branch.

Based on this function, we designed a microbenchmark with the pseudo-code shown below to measurer the size of GHR, i.e., how many bits in h contribute to the PHT look-up by the target branch, and also glean the predictor selection preference.

```
1  h1=value;
2  h2=h1^(1<<x);  //set the x-bit different
3  m=0;
4  for(i=0:n)  //Training loop
5  {
6      Access_PHT(h1, 1);
7      Access_PHT(h1, 1);
8      Access_PHT(h1, 1);
9      Access_PHT(h2, 0);
10     Access_PHT(h2, 0);
11     Access_PHT(h2, 0);
12 }
13 m += Access_PHT(h1, 1); //Testing period
14 m += Access_PHT(h1, 1);
15 m += Access_PHT(h1, 1);
16 print(m);
```

Listing 2. Microbenchmark 1

In this microbenchmark two h values are set, h_1 and h_2, with only one bit at a chosen position, x^{th} bit, different. In a single tagged predictor table, if the x^{th} bit contributes to the indexing, h_1 and h_2 will make the target branch index into different entries. However, if the x^{th} bit does not contribute to PHT indexing, the effective bits in h_1 and h_2 (lower than the x^{th} bit) that are kept in the GHR for the target branch are the same, and therefore the target branch will index into the same entry with h_1 and h_2. There are two main parts in this microbenchmark, the training loop (Lines 4-12) and the testing period (Lines 13-15). In the training loop, we have two groups of $Access\_PHT$ functions, setting an entry in the PHT for the target branch Taken (Lines 6-8) and Untaken (Lines 9-11), respectively. If h_1 and h_2 are indexing into different entries, the entry indexed with h_1 is set to strong T and the other entry with h_2 to strong NT. However, if h_1 and h_2 are indexing into the same entry, in each iteration, one entry is trained to be strong T first and then strong NT. In the test period, we test which situation is currently occurring by checking the state of the entry indexed with h_1. If on different entries, the starting state of the entry indexed by h_1 in the testing period would be strong T, and therefore no mis-prediction will appear and the microbenchmark returns 0 for the m value. If on the same entry, the starting state of the entry would be a strong NT, and two mis-predictions will appear in the testing period and the microbenchmark returns 2 for the m value.

We conducted experiments on Microbenchmark 1, observing the value of m as x varies from 0 to 15 with different number of iterations (n) in the training loop. We count the proportion of h_1 and h_2 indexed into different entries, which is the proportion of $m = 2$. In Fig 4, we present the changes in this proportion with respect to x for different values of n. It is evident that the result can be categorized into three distinct phases: when $0 \leq x \leq 4$, h_1 and h_2 always index into different entries, indicating that 0^{th}-4^{th} bits of GHR always contribute to PHT indexing no matter which predictor is chosen, and therefore the shorter size of GHR (s_1) is 5; when $x \geq 8$, h_1 and h_2 always index into the same entry regardless of which predictor chosen, indicating that those bits of GHR do not contribute to PHT indexing, and therefore the longer size of GHR (s_2) is 8; when $5 \leq x \leq 7$, the proportion exhibits a progressive increment as n increases, and we speculated that in this phase two tagged predictors(T_1 and T_2) have different behaviors/accuracy and the choice between them is not stable.

The change trend in the phase of ($5 \leq x \leq 7$) can help us understand the selection mechanism between T_1 and T_2. When n is small, T_1, which has the shorter GHR, is the preferred choice, to result in the same entry indexed by h_1 and h_2. As n gradually increases, the proportion of selecting T_2 increases (i.e., indexed into different entries). We speculated that by default T_1 is prioritized for selection, while certain mechanisms are in place to reduce the priority of T_1 when its accuracy drops significantly, favoring T_2 instead. In the training loop, when $5 \leq x \leq 7$, if T_1 is selected, there are in total 4 mis-predictions in each iteration, causing a low prediction accuracy of 33%. As n increases, the low accuracy progressively reduces the priority of T_1, and the selection of the predictor leans toward T_2.

Our further experiments show that although it takes thousands of loops to train the target branch prediction to favor T_2, only a few loops of high accuracy pattern will train it back to select T_1. This shows that the default choice of T_1 is more stable. In the following sections we will always train the target branch to use T_1 for prediction.

In summary, we conclude that the CBP of ARM Cortex-A53 is composed by three components: a base predictor table which is only indexed by the branch address, a tagged table that uses 5 bits GHR for indexing, and a tagged table that uses 8 bits GHR for indexing.

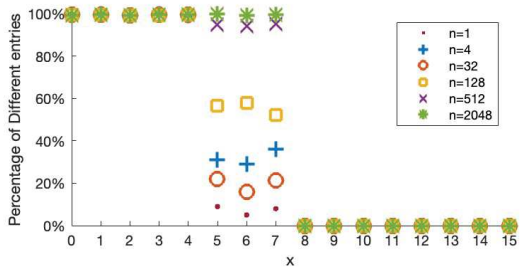

Fig. 4. Percentage of different PHT entries when changing x and n

C. Effective Bits in the Branch Address

When deciding the PHT index for a target branch, not all the bits in the branch address are involved in the PHT indexing and tag comparison, i.e., only some bits are effective. Next we figure out the effective bits in a branch address. We design an

advanced PHT-accessing function, where we not only set the GHR, but also select different branch instructions to access the PHT, as shown below.

```
1    Access_PHT_Adv(h, x, d){
2        if((h>>7)&1) {...}
3        ...          //Training GHR
4        if((h>>0)&1) {...}
5        return branch[x](d);
6    }
7    branch[0](d){
8        m0=misprediction_counter();
9        //read the event counter
10       if(d) {...}       //Target branch
11       m1=misprediction_counter();
12       return (m1-m0);
13   }
14   branch[1](d){...}
15   ...
16   branch[4095](d){...}
```

Listing 3. Advanced function for specifying a branch instruction and accessing a PHT entry

In addition to an 8-bit h vector and the branch direction d value, the function has a third parameter, x. We build an array of functions, $branch[4096]$, in order to generate different target branch addresses (one target branch is shown in Line 10). $branch[4096]$ contains 4096 functions, with each simply executing a conditional branch with the same global branch history (specified by the h value) and checking if a misprediction occurs for the target branch. All the $branch[x]$ functions have the same code, but would have different addresses for the target branches. We then perform an experiment shown below to check if two different target branches with the same global branch history can index into the same PHT entry.

```
1    h=value;
2    m=0;
3    for(i=0:100)
4    {
5        m += Access_PHT_Adv(h, x1, 1);
6        m += Access_PHT_Adv(h, x1, 1);
7        m += Access_PHT_Adv(h, x1, 1);
8        m += Access_PHT_Adv(h, x2, 0);
9        m += Access_PHT_Adv(h, x2, 0);
10       m += Access_PHT_Adv(h, x2, 0);
11   }
```

Listing 4. Identifying the effective bits of a branch address

In the TAGE predictor, although each table uses different GHR size, all the three tables use the same effective branch address bits. If the two target branches in the two branch functions (x_1 and x_2) have the same values on all the effective bits, they will use the same table entry, no mater which predictor is used. We define this condition where two different branches use one PHT entry as a PHT collision. In this microbenchmark, we can detect whether branches x_1 and x_2 collide on a PHT entry by the value of m. As the PHT only has 3072 entries in total, there must be pairs of branches among the total 4096 branches that will construct PHT collisions according to the simple birthday problem.

We keep changing the value of x_1 and x_2 to select different target branches, and save the addresses of the two target branches if a PHT collision is detected. Fig 5 shows the average difference between x_1 and x_2 on each bit for PHT collisions, where the x-axis indicates the bit position. The results show that the lower 4th to 13th bits of the two addresses, x_1 and x_2, are identical when they collide on a PHT entry, i.e., the effective branch address bits. Note the last two address bits are always zero as each word is four-byte.

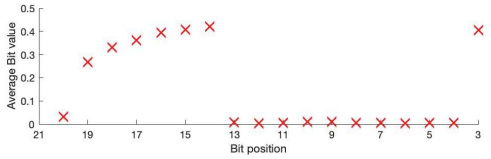

Fig. 5. Average difference between two colliding addresses on each bit

IV. The TrustZoneTunnel Attack

In this section, we elaborate and propose TrustZoneTunnel, a PHT-based side-channel attack framework, where the attacker in the normal world can extract the direction of a target branch in the secure world. We describe how to build a side-channel across the two worlds of TrustZone, and then implement the attack against an RSA implementation of the mbedTLS library [20] in the secure world.

A. Threat Model

We assume the common threat model for TrustZone, where the victim is a trusted application in the secure world and the normal world applications can only invoke it as service. The attacker has full control of the normal world, including the operating system. It can install external kernel modules to modify the interrupt handlers, has the privilege to access the PMU, and is able to assign specific cores to run the adversary program. We also assume that the attacker figures out the virtual address of a target branch, through knowledge to the source code and the binary of the victim application.

B. Attack Overview

TrustZoneTunnel aims to extract secrets from a victim application running in the secure world, if the conditional branches are dependent on the secrets. Our reverse engineering results indicate that with the TAGE predictor architecture, there are three prediction tables each of which uses a different length GHR in indexing. To avoid unexpected switching between the three predictors, we first train the target branch to use the tagged predictor that uses a 5-bit GHR for indexing, which is the most reliable predictor.

TrustZoneTunnel consists of three salient parts: the PHT Preset & Check function, world-switching, and PHT collision construction.

PHT Preset & Check mechanism: We propose a PHT Preset & Check mechanism, where we first preset the state of a PHT entry by executing a collision branch, and then checks the state update of the entry by the victim target branch. Both the collision branch in the normal world and the target branch in the secure world index to the same PHT entry during execution, i.e., a collision on the PHT occurs.

World-switching: to achieve a fine-grained control on the switching between the victim and the adversary, we implement

our attack with the Load-Step framework. As shown in Fig 2, the victim application is set to run on a victim core (in the secure world), and the adversary program keeps sending interrupts to the victim core from an auxiliary core. The interrupt handler is customized to embed the Preset & Reset functions to operate on a common PHT entry which the victim target branch collides with the collision branch on.

PHT collision construction: when constructing a PHT collision between the two worlds, the target branch in the victim application is already fixed, we then set the collision branch in the normal world to have the same values for the effective bits as the target branch, and also set the common branch global history for them.

C. PHT Preset & Check

We first construct a $PHT\_Preset()$ function to set the state of the 2-bit counter in a PHT entry to a specific value. After the target branch execution, we then monitor the state update via the $PHT\_Check()$ function, and speculate the direction of the target branch. The pseudo-code for the two functions is given below. We use an $Activate\_T1()$ function to train the CPU to use the tagged predictor table that is indexed with the 5-bit GHR (T_1). Note that to ensure the CPU consistently utilizes T_1, h needs to be 8 bits instead of 5 bits, otherwise the other table (T_2) might still be selected occasionally.

```
1  PHT_Preset(){
2      Activate_T1(h, x);
3      Access_PHT_Adv(h, x, 1);
4      Access_PHT_Adv(h, x, 1);
5      Access_PHT_Adv(h, x, 1);
6      Access_PHT_Adv(h, x, 0);
7  }
8  PHT_Check(){
9      m=0;
10     m+=Access_PHT_Adv(h, x, 0);
11     m+=Access_PHT_Adv(h, x, 0);
12     return m;
13 }
```

Listing 5. PHT Preset & Check Functions

The adversary collision branch instruction is in the branch function $branch[x]$ (shown in Listing 3). In the $PHT\_Preset()$ function, we use the collision branch to access a PHT entry with three taken and one non-taken, so as to train the two-bit counter in the entry to be a weak T (10). In the $PHT\_Check()$ function, we execute the same collision branch twice with direction of non-taken, and count how many mis-predictions have occurred. We deliberately control the victim to run the target branch between the Preset and Check functions, the m value returned by the $PHT\_Check()$ function will leak the execution direction of the target branch. Assuming no other branches are having a PHT collision with the target branch or the collision branch, i.e., no noise, and the target branch executes at most once in this piece of code. If m=0, no mis-prediction occurs when executing the two non-taken, meaning that the state of the two-bit counter at the beginning of the $PHT\_Check()$ is either Strong non-taken (00) or Weak non-taken (01), so the target branch must have executed as non-taken. If m=1, we can speculate that the state of the 2-bit counter start as weak-taken (10), implying that no target branch is executed between the Preset and Check. If m=2, the state of the 2-bit counter should be strong-taken (11), indicating that the target branch has been executed taken. With this method, we can build a PHT based side-channel, and extract the direction of the target branch.

D. Implementing TrustZoneTunnel with Load-Step

TrustZoneTunnel is implemented with the Load-Step framework, as shown in Fig 2, where the world-crossing interrupt handler is prefixed with a $PHT\_Check()$ function, to detect the impact of the prior execution of the victim application (possibly one target branch) in the secure world, followed by *Interrupt Handler* and *Context Recovery*, before a $PHT\_Preset()$ function, to set the target entry in the PHT to a known state before the victim core resumes the secure-world application. The Load-Step framework is installed in the Linux OS as an external kernel module. For the adversary program on the auxiliary core, it starts a timer once the TEE OS is activated. When the timer is up (after a fixed time interval), an interrupt signal is sent to the victim's core, forcing the secure-world application to pause and switching to the normal world for handling the interrupt. When the routine is over, the adversarial execution returns to the auxiliary core while the victim core is released to resume the victim application, to start next epoch. Note the $PHT\_Check()$ function will store its output in a trace file.

E. PHT collision construction

To build a collision branch, we need to first obtain the address of the target branch, and then select a branch in the normal world whose lowest 4^{th}-13^{th} address bits are the same as the target branch. Then we need to set the context for the target branch and the collision branch the same, i.e., setting a common branch global history (GHR) for both branches' execution.

For the collision branch in the Preset and Check functions, the branch global history is easy to set to any specific value with the method shown in Listing 1. So the important thing is to figure out the branch history for the target branch and then align the collision branch with it. We assume that the attacker can always interrupt the victim right before the target branch execution, so that the recent branch global history of the target branch is provided by the *World Switching* (WS) function, as shown in Fig 2. We assume these operations have a fixed branch profile and design an experiment to recover it.

We first write a simple trusted application shown below, where the directions of the target branch are decided by the value of the 64-bit secret. We put the trusted application in our Load-step framework for experiments, and adjust the interrupt timer to make sure that the 64 iterations (each with one target branch execution shown on Line 4) can all be monitored by interrupts (with PHT Preset+Check functions) preceding them.

```
1      uint64_t k = secret;    // secret
2      for(i=0;i<64;i++)
3      {
4          if((k>>i)&1)    //target branch
```

```
        { ... }
    }
```

Listing 6. A simplified victim

As the number of branches in the world switching function may be less than 5, we need to complement it with more branch executions to contribute to a full controllable GHR for the target branch, and the PHT_Preset() function is therefore extended, as shown below.

```
PHT_Preset(){
        ... //same with Listing 5
        h2=0;
        if((h2>>7)&1) {...}
        if((h2>>6)&1) {...}
        ...  //Setting target GHR
        if((h2>>0)&1) {...}
}
```

Listing 7. Extended PHT Preset Function

With this extended function, the value h_1 sets the GHR for the collision branch, while the lower part of the value h_2 combined with the branch pattern of the WS routine set the GHR for the target branch. Both h_1 and h_2 have 8 bits to activate the prediction table with the shorter GHR (5-bit), while only the least significant 5 bits contribute to PHT indexing. We first set h_2 as zero, and vary the other vector h_1 value from 0b00000000 to 0b00011111 to find one h_1 value, where PHT collisions between the target branch and the collision branch can be detected. The results are shown in Fig 6, where we calculate how many PHT collisions we can detect on this simplified victim among different h_1 values. The result shows that a PHT collision only happens when h_1=0b00000001, and a common GHR has resulted for the collision branch and the target branch.

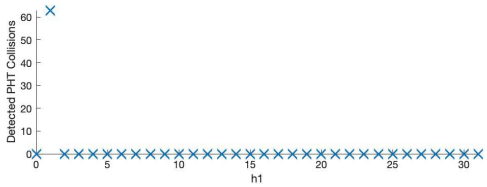

Fig. 6. Detected PHT collisions when varying h_1 in the range of 0 and 31

We design another experiment, now with h_1 fixed at value of 0b00000001 but varying h_2, to figure out the composition of the target branch GHR, i.e.,how many lower bits of GHR are contributed by the branch profile of the world switching routine, and how many upper GHR bits are contributed by part of h_2. The result is given in Fig 7, showing that as long as h_2 is an even value, PHT collisions can be detected. This indicates that only the last bit of h_2 affects collisions, and therefore the number of branches in the world switching routine must be four with their directions as 0b0001. Meanwhile, from Fig 7 we can also observe that only when h_2=0b00000 or 0b10000 we can detect all the 64 PHT collisions, while when the 2^{nd}-4^{th} bits dismatch the corresponding bits in h_1, we will always miss several collisions because the selection of the predictor T_1 is unstable. In our following experiments we still need to consider eight bits for the GHR, to train towards selecting the right predictor table, while only the last five bits are used for PHT indexing.

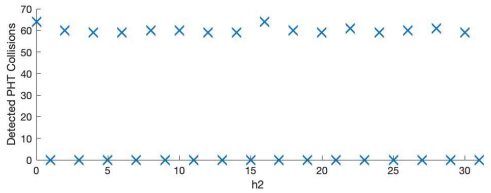

Fig. 7. Detected PHT collisions when varying h_2 in the range of 0 and 31

With this reverse engineering result, in attacks, the PHT_Preset function only needs to append four conditional branches, together with the world switching routine, to set the context GHR for the target branch.

F. TrustZoneTunnel on RSA

We next evaluate TrustZoneTunnel on real world trusted applications. We chose a sliding-window secure RSA implementation from the mbedTLS library as the victim, where our goal is to retrieve the key bits through the PHT-based side-channel. For any victim application, there are some application-specific adjustments that need to be made to tailor TrustZoneTunnel.

1) The target branch address: with the source code of the victim application, the attacker first needs to locate a conditional branch whose direction is related to sensitive information, such as key or the sign of a neuron output. For this RSA implementation, we select a target branch whose direction is directly decided by a key bit. By disassembling the binary file, the page offset of the target branch address can be retrieved so that majority of the effective bits (4th to 12th) are known.
2) The interrupt time interval: we usually aim at interrupting the victim every 3-10 instructions. To achieve this resolution, we need to adjust the timer according to the victim application. As different applications have different processor resources footprint (including the registers and live memory), the time for context saving and recovery is different. We implement a highly efficient software timer based on iterative plain arithmetic instructions, and the typical interrupt interval is 2000-4000 instructions. For this RSA implementation, the time interval is set at 2330 so that our side-channel can track the target branch execution with high precision.

We exemplify the RSA-4096 decryption process, a partial result is shown in Fig 8. Each PHT-Check result is presented by a scatter, showing a detection on the target branch to be taken (T), non-taken (NT) or not executed (NE). For this RSA victim, a single run of the TrustZoneTunnel contains about 10,000,000 interrupts to the victim, which takes about 200 seconds to run. Our results shows that we can detect all the executions of the target branch, and distinct their directions correctly. With these results we're able to recovery all 4096 bits in the RSA private key.

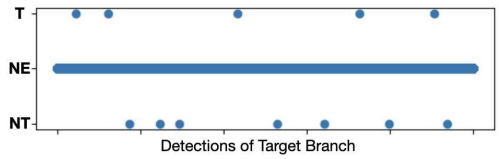

Fig. 8. Partial result of the RSA attack

V. Model Extraction Attack of Neural Networks

With TrustZoneTunnel, we further propose a model extraction attack on deep neural networks, where the adversary aims at stealing a function-equivalent copy of a deployed machine learning model. Previous works [7], [15] apply software methods to recover the model, where they need both the model outputs and the detailed logits when being queried in order to perform the attack. In contrast, our attack can recover a high-fidelity model with only the controlled inputs and the execution information leaked through our PHT-based side-channel.

A. Attack Overview

We start with a target application of simple multi-layer perceptron (MLP), which is composed of fully-connected layers and `ReLU` activations, with an `argmax` function in the last layer. We target at the branch in the `ReLU` activation function and the `argmax` function and observe their directions through our PHT-based side-channel. We iteratively change the input and observe the direction of a chosen `ReLU`, until the `ReLU` activation reaches a *critical condition*, i.e., its input is so small to be approximated as zero. We call such input a *witness* to the `ReLU`'s critical condition. The attack is conducted in two steps:

1) Weights recovery: for the layers except for the last layer, we search for the critical conditions for each neuron, i.e., the input to the `ReLU` function of the neuron is zero. We recover the input weights of each neuron based on many witnesses to its critical condition through linear regression.
2) Last-layer weight recovery: the last layer uses the `argmax` function instead of the `ReLU` function. We search for another type of critical condition under which two output logits are equal, and subseuqently recover the weights in the last fully-connected layer.

Next we illustrate the attack with a sample $2 * 2 * 3$ MLP model, which consists of two input features, one fully-connected layer with two neurons each followed by a `ReLU` function, and a last-layer with three neurons and an `argmax` function, as shown in Fig 9. The model is implemented with Trusted-DNN [19], a TrustZone-based adaptive isolation strategy for DNN models.

B. Weight Recovery

Weight recovery relies on searching for witnesses of the critical condition for each neuron of the hidden layers. Previous work [7], [15] exploits the gradients on the model output logits to search for witnesses, while in our attack we only use the proposed side-channel without using logits, i.e., treating the model inference execution as a black-box, a more realistic attack scenario.

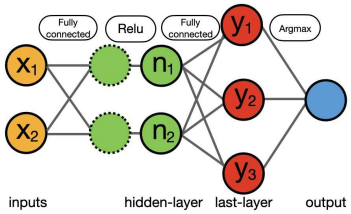

Fig. 9. A simple victim MLP model

The implementation of the `ReLU` (Rectified Linear Unit) activation function is presented below, where Line 6 is our target branch, whose direction is determined by the sign of the function's input, essentially the output from the preceding fully-connected layer. Specifically, the branch direction hinges on whether this input value exceeds zero.

```
1  void relu_op_forward(nonlinear_op *op)
2  {
3      for (int i=0; i<(op->out_units); i++)
4      {
5          op->output[i] =
6          op->input[i]>0 ? op->input[i]: 0;
7          //Target branch
8      }
9  }
```

Listing 8. `ReLU` activation function

For the hidden fully connected layer of the example DNN model shown in Fig 9, there are two neurons, n_1 and n_2, with their output calculated by $O_i = \mathbf{W_i} * \mathbf{X} + B_i$, where $\mathbf{X}$ is the input vector $\{x_1, x_2\}$, $\mathbf{W_i}$ the associated weight vector $\{W_{1i}, W_{2i}\}$, and B_i the bias for this neuron. The neuron output will go through the `ReLU` function to rectify it, where the `ReLU` function contains a conditional branch as shown in Listing 8. By monitoring this branch instruction using our PHT-based side-channel, the sign of the input to `ReLU`, O_i, is detected.

We first set the model input at zero, $x_1 = x_2 = 0$, to reveal the sign of the biases B_i. Previous work has shown that such model extraction attack only retrieves a function-equipment network, with the weights and biases determined relatively, i.e., with a scalar multiplier [15]. We normalize the weights and biases according to the biases, assuming $B_i = 1$ if it is positive; and $B_i = -1$ if negative.

We next search for the witness to the critical condition for each neuron. That is, taking neuron n_1 as example, search for $\mathbf{X}$ that makes $O_1 = W_{11} * x_1 + W_{21} * x_2 + B_1 = 0$. We randomly generate inputs until we find two points $\mathbf{X}_1, \mathbf{X}_2$, such that their corresponding outputs O_1 have opposite signs. Then we use binary search iteratively until we find an input $\mathbf{X}$ that makes O_1 a small positive value, below the preset ϵ. We need to find multiple witnesses to the critical condition of neuron n_1, so that we can have sufficient linear equations over the weights $\{W_{11}, W_{21}\}$ in order to solve them (note B_1 is already known to be 1 or -1). We apply this method one by one to other neurons in the first layer. We then apply this weight recovery process layer by layer. For deeper layers, because the

weights and biases in previous layers are all recovered, we can set the input to the current layer to specific values for binary searching and finding the critical conditions and witnesses in a similar fashion as the first layer.

C. Last-layer Parameter Recovery

The last layer of a DNN model normally utilizes `argmax` instead of `ReLU` function to pick the highest logit. For example, for the MLP model we target shown in Fig 9, the three logits computed by the three neurons in the last layer are:

$$\begin{cases} y_1 = W_{11}^2 * FO_1 + W_{12}^2 * FO_2 + B_1^2 \\ y_2 = W_{21}^2 * FO_1 + W_{22}^2 * FO_2 + B_2^2 \\ y_3 = W_{31}^2 * FO_1 + W_{32}^2 * FO_2 + B_3^2 \end{cases}$$

where the FO_i are the feature map outputs from the first layer, and the superscript 2 indicates it is the second layer (we will drop it in the following description for simplicity). Since the classifier outputs the class corresponding to the highest value among y_1, y_2 and y_3, its classification only depends on the relative comparisons among the three y_i. Therefore, the network is functionally equivalent when the last layer is re-parameterized as:

$$\begin{cases} y_1^* = 0 \\ y_2^* = \bar{W}_{21} * FO_1 + \bar{W}_{22} * FO_2 + \bar{B}_2 \\ y_3^* = \bar{W}_{31} * FO_1 + \bar{W}_{32} * FO_2 + \bar{B}_3 \end{cases}$$

where $\bar{W}_{i1} = W_{i1} - W_{11}$, $\bar{W}_{i2} = W_{i2} - W_{12}$ and $\bar{B}_i = B_i - B_1$ for $i = 2, 3$. This conversion reduces the number of variables from nine to six. Similar to the weight recovery process shown in Section V-B, we just need to recover the relative parameters while assuming $\bar{B}_i$ as 1 or -1.

We look into the `argmax` function and track conditional branches used. The source code of `argmax` function is shown below.

```
1  void argmax(float *arr, int n, TEE_Param * temp)
2  {
3      ...
4      for (int p = 0; p<n; p++)
5      {
6        if (arr[p] > max) //Target branch
7        {
8           idx = p;
9           max = arr[p];
10        }
11     }
12     ...
13 }
```

Listing 9. The last-layer `argmax` function

The source code shows that with n neurons (logits) in the last layer, there are n comparisons, implemented as conditional branches (Line 6), to bubble sort the highest logit. The first execution of the target branch is always taken, while the second execution compares y_2 and y_1, and the third execution compares either y_3 and y_1 or y_3 and y_2, depending on the result of the second execution. Assuming the attacker has already recovered all the previous layers and is able to set value for FO_1 and FO_2, with the critical condition setting method described in Section V-B, we can find witness to the critical condition of y_1=y_2 and y_1=y_3 (or y_2=y_3), which can be represented by:

$$\begin{cases} \bar{W}_{21} * FO_1 + \bar{W}_{22} * FO_2 + \bar{B}_2 = 0 \\ \bar{W}_{31} * FO_1 + \bar{W}_{32} * FO_2 + \bar{B}_3 = 0 \\ OR \\ (\bar{W}_{21} - \bar{W}_{31}) * FO_1 + (\bar{W}_{22} - \bar{W}_{32}) * FO_2 + (\bar{B}_2 - \bar{B}_3) = 0 \end{cases}$$

With witnesses for these critical conditions found, i.e., a set of $\{FO_1, FO_2\}$ values, the four parameters, $\bar{W}_{21}, \bar{W}_{31}, \bar{W}_{22}, \bar{W}_{32}$, will be solved. Note in our computation, we do not rely on knowing the values of logits (y_i), as the previous work did [15] (in a white or grey box fashion), but just exploit the PHT-based side-channel for weight recovery, a complete black-box attack model for the victim application.

D. Evaluation

We evaluate the performance of our model extraction attack on different MLP model sizes and architectures, and also extend our attack to a CNN model - Lenet-5. We compare the number of queries needed to recover weights of the whole network by our attack with the prior cryptanalytic/software model extraction work [7], [15]. We also give the execution time of our attack, compared to the original model execution time. The results are shown in Table I.

TABLE I
PERFORMANCE OF THE MODEL EXTRACTION ATTACK

Architecture	#of Parameters	Prior Queries	Our Queries	Model Time	Our Attack Time
784-32-1	25,120	$2^{18.2}$	2^{19}	0.5s	3.9s
784-128-1	100,480	$2^{20.2}$	2^{21}	1.8s	13.6s
10-10-10-1	210	2^{16}	2^{13}	79ms	238ms
10-20-20-1	420	$2^{17.1}$	2^{14}	86ms	517ms
40-20-10-10-1	1,110	$2^{17.8}$	2^{15}	95ms	836ms
80-40-20-1	4,020	$2^{18.5}$	2^{17}	145ms	1.7s
Lenet-5	44,426	$----$	2^{20}	265ms	11.6s

Note that in our attack the number of queries is linearly dependent on the number of neurons, and not related to the number of layers. So for deeper networks, our attack requires less queries than cryptanalytic methods. Also, the prior work does not apply to CNN models like LeNet-5 due to high complexity, while our attack successfully applies.

The result also shows that our attack increases the total execution time by about tenfold for all these models, which is acceptable for performing a feasible model extraction attack. As shown in Fig 2, this time overhead is due to frequent interruptions of the attack, including the customized interrupt handler and the world switching, where each interruption in the attack of DNN execution causes a delay of about 15 μs. As different victim applications use different amount of registers and memory resources, the context saving and restoring time may vary and such interruption delay may also vary, we set the timer interval as 2130 CPU cycles in the attack of the sample MLP model. For DNN inference, as the execution of ReLU activations only takes a small fraction of the time, we do not need to attack the entire network execution. We can profile the network execution and only start interruptions near the first

ReLU layer, to reduce unnecessary interruptions and therefore reduce the execution overhead. On average, 10 interrupts are needed for one ReLU iteration.

VI. Discussion

A. Applicability Across Cortex-A Models

We further analyze the BPU structure on other ARM processors. We choose four models, with distinct micro-architectures and from different vendors, including STMicroelectronics, Broadcom (Rasberry Pi), Qualcomm (XIAOMI 10), and Apple. The results are presented in Table II. We find that on the older ARM Cortex A7, the PHT only uses the GHR for indexing, without relying on the branch address. On newer ARM processors including Cortex-A72, Cortex-A76 and Apple M1, the GHR is comprised by the Path History Register (PHR), consisting of addresses of past taken branches, instead of Direction History Register (DHR) for the directions of conditional branches. We reverse-engineered the number of past branches associated with the Path History Register, as well as the effective bits in PC used for PHT indexing.

TABLE II
PHT Indexing Mechanism of other ARM Processors

Architecture	Platform	GHR	PC bits for index
Cortex A7	STM32MP157A	8-bit DHR	×
Cortex-A72	Rasberry Pi 4	32-address PHR	[14:4]
Cortex-A76	XIAOMI 10	48-address PHR	[14:4]
Firestorm(M1)	Macbook Air	96-address PHR	✓

The proposed attack framework can be easily applied to older ARM models, while the newer ARM processors are more similar to Intel processors with the use of PHR in indexing the PHT [14], [29]. We found that the BPUs on Cortex-A72 and Cortex-A76 are similar to each other, differing significantly from that of Cortex-A53 and Intel processors: the PHT table is only for storing branches predicted to be taken. The PHT is two-way associative. There is still a 2-bit counter in each PHT entry, set to 11 when a branch is put in. The counter decrements when the branch is resolved as non-taken, and increments when taken. A branch will be removed from the PHT when its counter decreases to 00. For branch prediction, if the executing branch hits one of the PHT entry, it is predicted to be taken, and non-taken otherwise on table miss. Correspondingly, the salient attack gadgets, including the PHT entry collision construction and the PHT Preset & Check mechanism would need to adapt for the newer processors.

B. Attack Mitigations

Half&Half [29] introduced a mitigation for BPU-based side-channel attacks on Intel systems by partitioning conditional branch addresses between domains with different privileges, i.e, make their effective bits different to prevent PHT collisions across these domains. However, this compiler-level solution may not be effective in a TEE setting, where attackers with OS-level privileges could circumvent compiler-based defenses. An alternative strategy is to capture unique features of the attack for detection, and prior works [9], [27] have suggested observing frequent interrupts in a victim enclave as an anomaly - indicator of attacks, which will incur performance degradation. Another common software based mitigation is getting rid of conditional branches in security-critical applications [4], [11], while its algorithm-specific nature limits its broad applicability. Other methods, including flushing all the counters in the PHT entries when the execution switches from the secure world to the normal world, and using two different BPUs on a single core for different worlds, are also effective, with non-negligible implementation and execution overhead.

Specific mitigation tailored to features of TrustZoneTunnel would be more effective. To create collision on the PHT in our TrustZoneTunnel, the adversary has to deliberately align the GHR seen by the collision branch (in the Preset and the Check functions) with the GHR seen by the target branch (in the victim function), as described in Section IV-E. We propose a countermeasure to introduce dummy branch executions with random directions in the world-switching routine, and therefore the adversary fails to profile it and align their GHR for the collision branch with it. For our experimental platform, we modify OPTEE's secure world unbanked registers restore function, which will be called when the execution switches from the normal world to the secure world. We utilize the CPU cycle counter to introduce a pseudo-random number to specify the directions of dummy branches. On Cortext-A53, we insert eight dummy branches to ensure that the entire GHR for the target branch is random. Owing to the minimal resource and time consumption of the branch execution, this mitigation is light-weight and efficient.

VII. Conclusion

In this paper, we propose TrustZoneTunnel, the first PHT-based side-channel attack framework against ARM TrustZone, which can extract the directions of any conditional branches that execute in the secure world from the normal world. We reverse engineer the PHT indexing mechanism in a Cortex A53 processor and several other ARM processors, and design several important primitives including PHT Preset & Recheck, Load-Step world-switching method and PHT entry collisions. We apply TrustZoneTunnel to trusted DNN applications to evaluate the attack effectiveness and performance, and implement a strong model extraction attack that can recover model parameters using only the side-channel leakage. TrustZoneTunnel show that although ARM TrustZone provides considerable hardware-level isolation to protect applications in the secure world, the shared micro-architecture for speculative execution becomes the vulnerability and can leak critical information from the secure world, breaching confidentiality and privacy and undermining the protection of ARM TrustZone.

Acknowledgment

This work was supported in part by National Science Foundation under grants CNS-2212010, SaTC-1929300, and IUCRC-1916762/CHEST center.

REFERENCES

[1] Arm's cortex-a53: Tiny but important. https://chipsandcheese.com/2023/05/28/arms-cortex-a53-tiny-but-important/, 2023.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *Cryptography and Coding: 11th IMA International Conference, Cirencester, UK, December 18-20, 2007. Proceedings 11*, pages 185–203. Springer, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*, pages 225–242. Springer, 2006.

[4] Giovanni Agosta, Luca Breveglieri, Gerardo Pelosi, and Israel Koren. Countermeasures against branch target buffer attacks. In *Workshop on fault diagnosis and tolerance in cryptography (FDTC 2007)*, pages 75–79. IEEE, 2007.

[5] Arm. *Arm® Cortex®-A53 MPCore Processor*.

[6] ARM. Cortex-a53. https://developer.arm.com/Processors/Cortex-A53.

[7] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III*, pages 189–218. Springer, 2020.

[8] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution.(2018). *arXiv preprint arXiv:1802.09085*, 2018.

[9] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[10] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Prime+ count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 441–452, 2018.

[11] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 182–182. IEEE Computer Society, 2001.

[12] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[13] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[14] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 321–347, 2020.

[15] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1345–1362, 2020.

[16] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+ evict. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 979–984. IEEE, 2021.

[17] Ben Lapid and Avishai Wool. Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis. In *International Conference on Selected Areas in Cryptography*, pages 235–256. Springer, 2018.

[18] Xinyao Li and Akhilesh Tyagi. Cross-world covert channel on arm trustzone through pmu. *Sensors*, 22(19):7354, 2022.

[19] Zhuang Liu, Ye Lu, Xueshuo Xie, Yaozheng Fang, Zhaolong Jian, and Tao Li. Trusted-dnn: A trustzone-based adaptive isolation strategy for deep neural networks. In *ACM Turing Award Celebration Conference-China (ACM TURC 2021)*, pages 67–71, 2021.

[20] mbedTLS. "mbedtls". https://github.com/Mbed-TLS/mbedtls.

[21] OPTEE. "optee". https://www.op-tee.org/.

[22] Stephen Pruett, Siavash Zangeneh, Ali Fakhrzadehgan, Ben Lin, and Yale Patt. Dynamically sizing the tage branch predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[23] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[24] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[25] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9:1–6, 2007.

[26] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[27] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[28] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. Interface-based side channel attack against intel sgx. *arXiv preprint arXiv:1811.05378*, 2018.

[29] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237. IEEE Computer Society, 2023.

[30] Lutan Zhao, Peinan Li, Rui Hou, Michael C Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. A lightweight isolation mechanism for secure branch predictors. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1267–1272. IEEE, 2021.