applied sciences

MDPI

Article

Memory and Cache Contention Denial-of-Service Attack in Mobile Edge Devices

Won Cho and Joonho Kong *

School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, Korea; jowon@knu.ac.kr
* Correspondence: joonho.kong@knu.ac.kr

Abstract: In this paper, we introduce a memory and cache contention denial-of-service attack and its hardware-based countermeasure. Our attack can significantly degrade the performance of the benign programs by hindering the shared resource accesses of the benign programs. It can be achieved by a simple C-based malicious code while degrading the performance of the benign programs by 47.6% on average. As another side-effect, our attack also leads to greater energy consumption of the system by 2.1× on average, which may cause shorter battery life in the mobile edge devices. We also propose detection and mitigation techniques for thwarting our attack. By analyzing L1 data cache miss request patterns, we effectively detect the malicious program for the memory and cache contention denial-of-service attack. For mitigation, we propose using instruction fetch width throttling techniques to restrict the malicious accesses to the shared resources. When employing our malicious program detection with the instruction fetch width throttling technique, we recover the system performance and energy by 92.4% and 94.7%, respectively, which means that the adverse impacts from the malicious programs are almost removed.

Keywords: memory and cache contention; denial of service attack; shared resources; performance; energy

Citation: Cho, W.; Kong, J. Memory and Cache Contention Denial-of-Service Attack in Mobile Edge Devices. *Appl. Sci.* **2021**, *11*, 2385. https://doi.org/10.3390/app11052385

Academic Editor: Régis Leveugle

Received: 24 December 2020
Accepted: 4 March 2021
Published: 8 March 2021

1. Introduction

Recently, computing systems are ubiquitous and pervasive. Particularly, mobile computing systems account for a large portion of the computing systems in the real-world. For example, smartphones or tablet PCs are widely used in people's everyday life. Although mobile edge computing systems are convenient for people, they also have limitations in terms of computing capability and availability. In the perspective of computing capability, mobile edge devices have limited computing resources, as they have small form factors. In the perspective of availability, they can only be used when there are available power sources (e.g., battery) to operate the devices. On the other hand, though they have less computing resources when compared to the high-performance computing environments, recent mobile edge device users demand high computational capability in their mobile devices for satisfactory mobile user experiences. Thus, mobile processors (that are a key component in mobile computing systems) are becoming heavier and trying to catch up with desktop performance [1].

Multi-core mobile processors have been introduced and widely used in mobile computing systems in order to meet those requirements. For example, Exynos [2] mobile system-on-chips (SoCs), which are employed in many smartphones and tablet PCs, have adopted an octa-core CPU. Although multiple cores can enhance system performance by exploiting parallelism in workloads, it is also a problem that some of the computing resources must be shared between the cores. In typical mobile processor designs, a few cores share L2 caches and main memories. In this case, when the multiple cores are running simultaneously, L2 cache or main memory access requests must compete with

each other to gain control to those shared resources. Eventually, there remains a possibility of denial-of-service (DoS) attack in mobile edge computing systems when there is a malicious program(s) that impede(s) normal (i.e., benign) programs access to the shared resources [3,4]. In addition, DoS attacks may result in huge energy consumption because the malicious code not only delays the execution of the normal programs (hence, it increases the static power consumption of the system), but it also consumes non-negligible energy by itself. This in turn leads to faster battery exhaustion in mobile systems, deteriorating user experiences.

In this paper, we introduce a memory and cache contention denial-of-service attack that causes a denial-of-service in normal programs. There have been a few previous studies [3–7] on the attack that is similar to that introduced in this paper. We bring the main concept of the attack from those works while revisiting it in the mobile edge devices. Our memory and cache contention attack can be easily performed by a simple C-based malicious code that is similar to memory copy operations with large data array size. The malicious code consecutively carries out successive memory operations (a sequence of load and store), resulting in much slower memory access of the normal programs.

We propose efficient hardware-based detection and mitigation techniques for countermeasures. Our detection mechanism exploits the differences on workload characteristics between the malicious code and general programs. By profiling the intensiveness of shared resource accesses, our detection technique can effectively and efficiently detect the malicious code for the memory and cache contention attack with very low false alarm rates. For mitigation, we also introduce a simple solution that reduces the instruction fetch bandwidth. It can be implemented by reducing the instruction fetch width to completely stop executing the malicious codes. We evaluate performance degradation from the memory and cache contention denial-of-service attack and how much our mitigation mechanism can recover performance losses with cycle-accurate simulations. In addition, we provide energy results, as our memory and cache contention denial-of-service attack leads to huge energy consumption in mobile systems and also shows how our countermeasure effectively reduces the energy consumed by the malicious program.

We summarize our contributions, as follows:

- we introduce memory and cache contention denial-of-service attack and demonstrate how this attack threatens mobile edge computing systems in terms of both performance and energy;
- through our evaluations, we show our memory and cache contention attack can degrade system performance by 47.6% on average and also increase the system energy consumption by 2.1×;
- our proposed hardware-based detection method effectively detects the malicious codes by analyzing the intensiveness and steadiness of the shared resource accesses; and,
- with our countermeasure that reduces the instruction fetch bandwidth, one can recover performance degradation and exhaustive energy consumption by 92.4% and 94.7%, respectively, when compared to the case without our countermeasure.

The rest of the section organization is composed, as follows. Section 2 introduces related work regarding the attacks that exploit micro-architectural vulnerabilities, malicious code detection methods, and studies that are related to the shared resource access and control in computer systems. Section 3 explains our baseline system model, assumptions, threat model, and attack mechanism with example malicious codes. Section 4 describes our detection and mitigation techniques. Section 5 shows our evaluation framework and the results in terms of performance and energy. Lastly, Section 6 concludes this paper.

2. Related Work

Micro-architectural denial-of-service attack has already been already introduced in several studies [3–7]. Woo and Lee introduced the performance vulnerability of micro-architectural DoS attacks, which hinder the normal programs accessing the shared re-

sources [4]. They introduced several attack methods (e.g., continuous memory loads with a fixed stride, exploitation on inclusion properties, or locked atomic operations, etc.) to cause DoS in the chip multi-processors (CMPs). They also show that the malicious micro-architectural DoS attack can severely degrade performance of the normal program by up to 91%. However, the malicious code detection methods are not proposed in [4] and also their solutions to mitigate the attacks are not evaluated. Although our work is based on the malicious code simpler than that introduced in [4], our work shows that the memory and cache contention DoS attack can be performed, even without well-prepared and meditated malicious codes. It implies that our attack could be easily exploited, even though the attacker does not have deep knowledge in target processor architecture. In addition, we also present energy evaluation results, which are significant consequences of the memory and cache contention attack.

In [3], another type of micro-architectural DoS is introduced. The main focus of [3] is unfairness in memory bandwidth across the programs running simultaneously in the system. Their solution to relieve memory bandwidth unfairness tries to guarantee the fair sharing of the available memory bandwidth with their proposed memory scheduling algorithm. Although their solution could relieve performance losses from the attack to some extent, it cannot fully recover performance of the normal programs because it only provides the fairness of the memory bandwidth across the programs running simultaneously in the system while not fully restricting the shared resource accesses from the malicious program.

In [5], the DoS attack in the cloud system is introduced. It exploits that multiple virtual machines (VMs) are typically co-running in one physical machine in the cloud. A malicious VM can try to encroach on the effective cache capacity of the benign (victim) VM, incurring performance losses. For the defense mechanism, they exploit the observation that the memory bandwidth requirements for the malicious and normal VMs are different. In order to identify the malicious VM, they sample the memory access statistics. Once detected as the malicious VM, the proposed mitigation mechanism throttles or migrates the malicious one not to encroach on the effective cache capacity and memory bandwidth. However, the main target for the attack in [5] is cloud systems, while ours is mobile devices. Although the proposed detection and mitigation methods are suitable for high-performance cloud systems, it might be difficult to employ them in the mobile systems. This is because the maximum memory bandwidth and available hardware resources in the cloud systems are typically much higher than the mobile systems. Thus, if the attacker tries to fully exploit the available memory bandwidth and resources in the cloud system, it would be relatively easy to distinguish between the malicious and normal VMs. However, in mobile systems, only considering the memory bandwidth statistics would make it difficult to detect the malicious program. This is because the maximum achievable memory bandwidth requirement by the malicious code would be much smaller in the mobile system, which, in turn, makes it very hard to distinguish between the malicious and benign programs. In addition, the mitigation method that is presented in [5] is architecture-dependent (e.g., execution throttling) or cloud system-dependent (e.g., VM migration), which is difficult to employ in mobile systems.

In [6], Bechtel and Yun have demonstrated memory DoS attacks. As shown in [6], they have shown two types of the attack: read attack and write attack, which can significantly degrade the performance of the system. Additionally, they have introduced an OS-level detection technique that exploits last-level cache (LLC) writebacks and LLC misses (by using hardware performance counters) to monitor the memory traffic. However, our solution is a hardware-based solution that can be activated faster than the OS-level solution. In addition, our solution can be employed in multi-core embedded systems where operating systems are not available (e.g., firmware-based systems). In [7], Bechtel and Yun have also demonstrated another type of the memory DoS attacks by using parallel linked lists. By additionally exploiting the DRAM bank conflicts, the DoS attack has shown up to 111× worst-case execution time increases. However, for a successful attack, the attacker should be aware of the memory address mapping in advance, while our attack

can be launched without any knowledge of the memory mapping, as discussed in [7]. Moreover, the mitigation mechanism for the attack is not presented in [7].

In [8], thermal-induced micro-architectural DoS attack is introduced. Their attack incurs thermal stresses in simultaneous multi-threading (SMT) processors, which results in performance degradation in the normal thread sharing the pipeline with the malicious thread. When compared to the attack introduced in [8], our attack is totally different, as we focus on generating meaningless requests to the shared resources instead of incurring thermal stresses in the shared resources.

Except for the studies introduced above, thermal-induced instruction cache attack [9], micro-architectural attack for side-channel analysis [10–12], and cache timing attacks to crack AES keys [13] were also proposed to exploit micro-architectural vulnerability for circumventing security. However, those studies are not related to denial-of-service attacks. The malicious code detection methods have also been studied [14–18]. When compared to the previous studies that focus on a general malicious code detection mechanism, we solely focus on the memory and cache contention attack detection method with low-cost hardware-based mechanisms. In [19,20], hardware-based online malware detection mechanisms are proposed. These detection mechanisms focus on typical malwares incurring non-negligible hardware overhead (e.g., logic for neural networks in [20]). In contrast, our detection and mitigation mechanism can not only be implemented with negligible hardware overhead, but it also does not cause any performance overhead for detection.

There have also been works that consider performance quantification and modeling due to the resource sharing in the multi-core system. In [21], Eklov et al. introduced a method for measuring the application performance, depending on the effective last-level cache capacity. The proposed method uses a specialized program, which is called Pirate program. The Pirate program is carefully designed to adjust the effective last-level cache (LLC) capacity based on the knowledge of the LLC replacement policy in the system. By simultaneously running the Pirate program and target program (whose performance will be measured), the proposed method accurately measures performance by reading the performance counters. In [22], the Bandwidth Bandit is proposed to measure the performance impact of sharing the off-chip main memory, while the method for performance measurement in [22] is very similar to that in [21]. In [23], an online performance model that considers the performance slowdown due to cache and memory sharing is proposed. With their performance model, they also proposed to partition the shared resources for minimizing the performance slowdown. However, those works focus on measuring or modeling the performance impacts due to resource sharing, while our work focuses on malicious DoS attack and its countermeasure.

3. Memory and Cache Contention Denial-of-Service (DOS) Attack

The main objective of our attack is to incur shared resource contention in the shared cache and main memory, which, in turn, degrades the performance and energy efficiency of the normal program. We also explain assumptions and limitations for the system in detail in Section 3.1.

3.1. System Assumptions

Our baseline system is composed, as shown in Figure 1. There are two processing cores inside of the mobile CPU, which has an L2 cache shared between the cores. In the lowest-level of the memory hierarchy, there is an LPDDR3 DRAM main memory. To model our system similar to the mobile edge computing systems, the processing core parameter is modified to model out-of-order execution-based mobile processing cores (such as ARM Cortex-A15 [24]) as close as possible. The L2 cache capacity is 2 MB and it has a stride-prefetcher that detects a memory access stride and prefetches data accordingly.

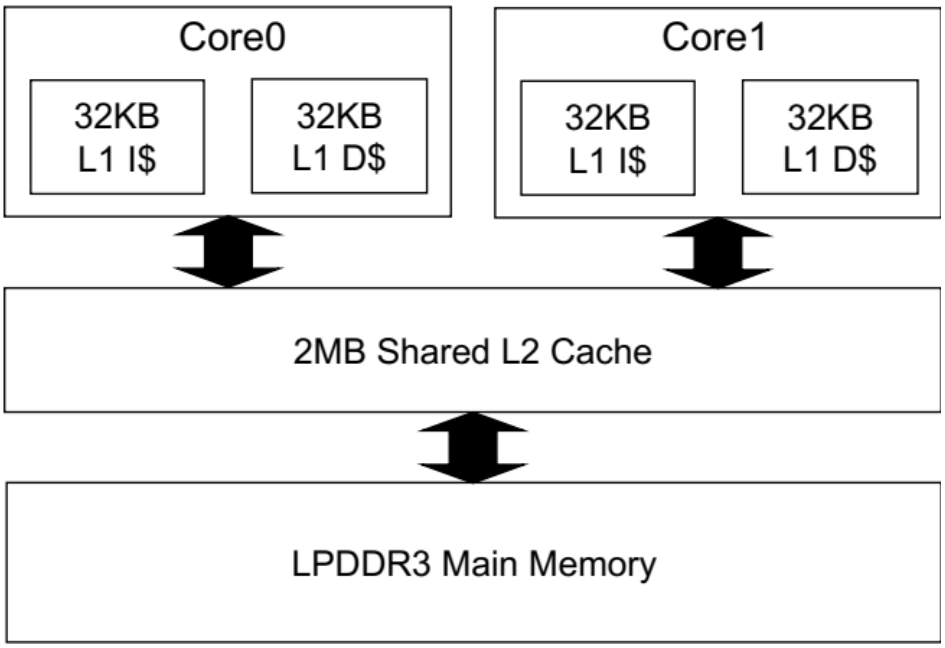

Figure 1. The cache and memory hierarchy in the system.

Each core has 32 KB 2-way L1 data and 32-KB 2-way L1 instruction caches. The number of entries in the L1 D-cache and L1 I-cache miss status handling registers (MSHRs) are 6 and 2, respectively. Table 1 summarizes our core and system-level architectural parameters.

Table 1. System and processor (CPU) specifications.

		Specification
Processor core	Branch predictor	Bimodal predictor, 2 K BTB
	L1 data cache	32 KB, 2-way set-associative, 2-cycle, LRU
	L1 instruction cache	32 KB, 2-way set-associative, 1-cycle, LRU
	Fetch width	3
	Issue width	8
	Commit width	8
	Clock frequency	2 GHz
L2 cache		2 MB, 8-way set-associative, 12-cycle, LRU, Stride-prefetcher
Memory		LPDDR3, 800 MHz, 1-channel, 1-rank, 8-bank

Although our attack is not limited to the system entirely same as that shown in Figure 1 and Table 1, the most important assumptions for our attack are: (1) there must be multiple cores in the CPU and (2) there must be shared L2 (or last-level) cache and main memory. The first assumption is required for simultaneously running the normal (benign) and malicious programs in the system. The second assumption is required for causing denial-of-service in the normal program as the normal and malicious program share the cache and memory resources in the system. A clear limitation of our attack is that the attack can only be launched in the system in which both of the assumptions are satisfied. However, please note that most of the mobile systems are adopting the multi-core

CPUs, shared caches, and unified main memory, which means that most of the mobile systems are vulnerable to our attack.

3.2. Our Attack and Threat Model

We assume that the attacker can hide (or inject) the malicious code inside of the application (or application update package), pretending to be benign (i.e., in order not to be detected). In addition, the attacker can compromise untrusted communication mediums, such as public wireless access points. Once they are compromised, the attacker can eavesdrop and/or manipulate the traffic across the compromised medium. The attacker does not need to gain physical access to the victim mobile devices.

Our attack can be triggered by an injected malicious code in the unauthorized application package (e.g., apk file in Android), as shown in Figure 2a. The attacker can make a benign user install the application package that includes our malicious codes (e.g., by deceiving the user). The installed application process (pretending to be a normal application) can create a thread that includes our malicious code inside of the process, which runs as a background process not to be detected by the user. In Scenario 1, a single mobile device user can be affected. However, the attacker may deceive a huge number of mobile device users, so that the attack can be widespread.

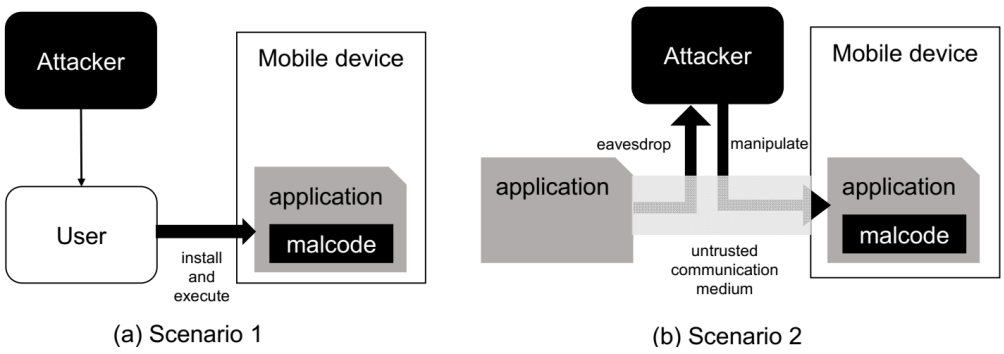

Figure 2. Possible attack scenarios.

Another scenario is also possible, as shown in Figure 2b. For example, the attacker can establish a rogue wireless access point, as presented in [25]. Based on our assumption on the attacker's ability, the attacker can inject a malicious code when the user downloads resources or applications via the untrusted/compromised communication medium. In this scenario, many devices that are connected to the rogue access point (or other types of compromised communication medium) can be affected. In addition to personal mobile devices, it is also possible to attack the mobile edge servers, such as unmanned aerial vehicles (UAVs) [26]. In this case, our malicious code can be injected into the system by the attacker through Scenario 1 or 2 shown in Figure 2. When causing DoS in the mobile edge server, the connected other mobile or Internet-of-Things (IoT) devices can be affected by our DoS attack. It would incur a catastrophic impact if the devices are communicating with the mobile edge servers for hard real-time tasks.

In the mobile device, our attack can be realized in multi-core CPUs, where multiple cores share last-level (L2) cache and main memory. In the attack scenario, we assume that the normal (i.e., legitimate) program is running on Core0, while the malicious program for intentional memory and cache contention is running on Core1. The Core0 tries to access the shared L2 cache and main memory when the normal program exhibits L1 cache misses and L2 cache misses (or prefetch requests from the L2 cache), respectively. In this case, when in the normal situation (Figure 3a), the normal program accesses the shared resources without any interference from the other programs.

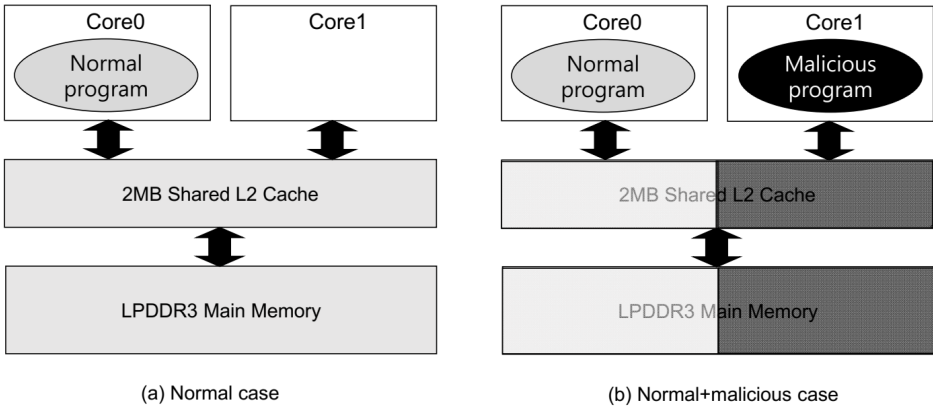

Figure 3. The cases where (**a**) the normal program is solely running and (**b**) both normal and malicious programs are running in the system.

On the contrary, in the case where the normal and malicious programs are running together in the system, the malicious program consecutively generates memory access requests, as it merely loads data and stores it again into the memory with a fixed stride streaming behavior (an example malicious code will be shown in the next subsection). For a successful attack, a data size to be accessed must also be sufficiently larger than the L2 cache size. Otherwise, a large portion of the data requested from the malicious code can be found in the L2 cache, failing to generate a large number of the malicious main memory requests. Because the accessed data size is sufficiently larger than the L2 cache size (thus, much larger than the L1 data cache size), the malicious code can consecutively generate a large number of L1 data cache misses, which, in turn, tries to consecutively access the L2 cache. Consequently, the malicious code can rapidly generate a number of L2 cache access requests (i.e., L1 data cache miss requests) by sufficiently utilizing the MSHR entries in the L1 data caches. Thus, the generated requests from the L1 data caches put a strong bandwidth pressure to the L2 caches. In the L2 cache, because of the stride-prefetcher, it may not generate a large number of the L2 cache misses from the malicious code. However, the L2 cache prefetcher will consecutively generate the memory access requests to fetch the data in the malicious program into the L2 cache in advance. In this case, the effective main memory bandwidth for the normal programs is also reduced. In summary, our memory and cache contention DoS attack eventually leads to two adverse impacts on the normal programs:

- a large number of repetitive access requests to the L2 cache incurred by the malicious code will consecutively prefetch the data of the malicious code into the L2 cache, evicting the data of the normal program from the L2 cache (i.e., higher L2 miss rate of the normal programs); and,
- a huge number of L2 prefetch requests that are caused by the malicious code will exhaust the effective main memory bandwidth, resulting in available memory bandwidth reduction and longer memory access latency of the normal program.

Consequently, there can be two system-level adverse impacts from the attack. Firstly, the shared resource contention between the normal and malicious program will lead to performance degradation (i.e., denial-of-service) of the normal programs. Secondly, the malicious program will draw non-negligible power, while a longer execution time of the normal programs results in higher static energy consumption. It means that the adverse consequences of the memory and cache contention attack would lead to user dissatisfaction due to the shortened battery life in mobile edge devices as well as performance degradation.

Please note that typical mobile device users are hard to detect whether the malicious code is running or not in the device, because it is executed in the background. In addition, the application that is used by the attacker to hide the malicious code pretends to be a normal application (e.g., a category of applications that are widely and frequently executed in mobile devices), which also makes a detection of our malicious code even more difficult.

3.3. Malicious Code Examples

Figure 4 demonstrates code snippets of our example malicious codes for memory and cache contention DoS attack. Simple memory data copy operations are repeatedly performed in the infinite loop, as shown in Figure 4a. When copying the data, the malicious code only loads 4-byte (one integer element) within the 64-byte cache line. It eventually leads to faster issuing of data fetching requests to the shared resources (L2 cache and main memory) while preventing the L1 data cache hits for the remaining 15 integer elements that reside in the same cache line. Because our malicious code shows a streaming behavior with the fixed stride (64-byte), the stride-prefetcher employed in the L2 cache also generates a number of prefetching requests (default degree: 8) to the main memory, as we explained in the previous subsection. This, in turn, gives further bandwidth pressure to the main memory. For memory footprint of our malicious code, we touch 32 MB (it actually copies 2 MB of the data, as we only copy 4 bytes in a 64-byte cache line) of the memory address space for each source (*sender* in Figure 4) and destination array (*receiver* in Figure 4), which is much larger than the L2 cache size. This eventually incurs a large number of malicious L2 and main memory access requests.

```
for(;;) {
    for(i=0; i<131072; i++) {
      receiver[0+64*i]  =sender[0+64*i];    // first cache line
      receiver[16+64*i]=sender[16+64*i];  // second cache line
      receiver[32+64*i]=sender[32+64*i];  // third cache line
      receiver[48+64*i]=sender[48+64*i];  // fourth cache line
    }
  }
```

(a) Simple assignment-based malicious code (type-1)

```
for(;;) {
    for(i=0; i<131072; i++) {
      int_val1 = int_val1 + int_val2;
      receiver[0+64*i]  =sender[0+64*i];    // first cache line
      receiver[16+64*i]=sender[16+64*i];  // second cache line
      receiver[32+64*i]=sender[32+64*i];  // third cache line
      receiver[48+64*i]=sender[48+64*i];  // fourth cache line
    }
  }
```

(b) Assignment and integer addition-based malicious code (type-2)

```
for(;;) {
    for(i=0; i<131072; i++) {
      int_val1 = int_val1 * int_val2;
      receiver[0+64*i]  =sender[0+64*i];    // first cache line
      receiver[16+64*i]=sender[16+64*i];  // second cache line
      receiver[32+64*i]=sender[32+64*i];  // third cache line
      receiver[48+64*i]=sender[48+64*i];  // fourth cache line
    }
  }
```

(c) Assignment and integer multiplication-based malicious code (type-3)

```
for(;;) {
    for(i=0; i<131072; i++) {
      fp_val1 = fp_val1 + fp_val2;
      receiver[0+64*i]  =sender[0+64*i];    // first cache line
      receiver[16+64*i]=sender[16+64*i];  // second cache line
      receiver[32+64*i]=sender[32+64*i];  // third cache line
      receiver[48+64*i]=sender[48+64*i];  // fourth cache line
    }
  }
```

(d) Assignment and floating point addition-based malicious code (type-4)

Figure 4. Code snippets of the example malicious codes.

Because there could be a number of variations (i.e., derivatives) for possible malicious codes, we also show more malicious code examples. A variety of malicious codes can be derived, as shown in Figure 4b–d. The overall shapes of the malicious codes are similar to the codes that are shown in Figure 4a, while there is a slight difference inside of the main loop. In Figure 4b–d, integer-type addition, integer-type multiplication, and float-type addition are performed, respectively. Because they also contain consecutive memory accesses (load and store), they also have an effect of the memory and cache contention DoS attack, although the program behaviors and characteristics (e.g., functional unit usage) are different from other types of the malicious codes. In this case, if we try to detect the malicious codes by referring to the functional unit usage patterns (e.g., by sampling the performance counters), the detection of the memory and cache contention DoS attack will be difficult, as they can utilize a variety of the functional units in the CPU.

Although one may perform assembly code-level optimizations for more effective attack, we do not perform any code optimization from our C-code example. One of the main threats of our memory and cache contention DoS attack is that a simple and naive malicious code can also lead to a successful attack, which implies that it could be very widely exploited.

4. Hardware-Based Countermeasures for Memory and Cache Contention DoS Attack

We should provision malicious code detection and mitigation methods to alleviate adverse impacts from the memory and cache contention DoS attack. In this section, we first

explain how we can detect malicious code and mitigate the adverse impacts from the memory and cache contention DoS attack.

4.1. Our Proposed Detection Method

In this subsection, we provide a detailed explanation of our detection mechanism in order to thwart the memory and cache contention DoS attack. Our detection mechanism focuses on the specific behavior of the malicious codes for the memory and cache contention DoS attack. For a successful memory and cache contention DoS attack, the malicious code must intensively and steadily generate access requests to the shared resources (L2 cache or main memory). Consequently, it generates access requests to the L2 cache with a stable (i.e., not much fluctuating) request interval pattern. Based on this insight, we detect how steadily and intensively L1 data cache miss requests are generated from a certain core and issued to the L2 cache within a certain time interval, since the malicious code issues the requests with a relatively fixed rate when compared to the normal programs.

Figure 5 shows our L1 data cache miss request interval detection mechanism with a hardware block diagram. The detection is carried out, as follows. Our detection hardware monitors miss requests from the L1 data caches from the core. Once a miss request is generated and sent to the L2 cache, we increment RCounter by one. CCounter is also incremented by one every clock cycle. Once the CCounter value becomes greater than the $\boldsymbol{Thres_{int}}$, we check whether the following condition is met or not:

$$\boldsymbol{Thres_{lower}} < \boldsymbol{RCounter} < \boldsymbol{Thres_{upper}} \quad (1)$$

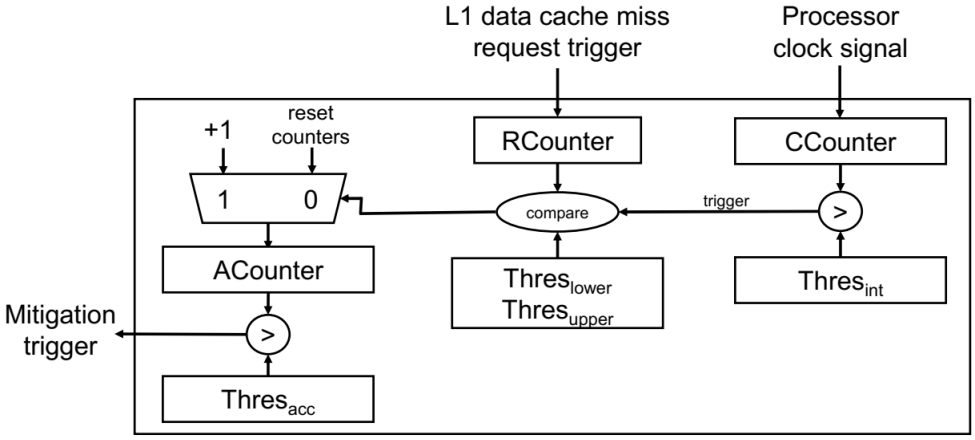

Figure 5. Hardware block diagram for our proposed detection mechanism. The detection hardware is installed in each core.

If the condition is met, then we increment ACounter value by one. Otherwise, we reset the ACounter value to '0'. Once the ACounter value hits $\boldsymbol{Thres_{acc}}$, the mitigation trigger signal is asserted. In other words, our mitigation mechanism is triggered when the condition in Equation (1) is consecutively satisfied by $\boldsymbol{Thres_{acc}}$ times.

In this work, the threshold values ($\boldsymbol{Thres_{int}}$, $\boldsymbol{Thres_{lower}}$, $\boldsymbol{Thres_{upper}}$, and $\boldsymbol{Thres_{acc}}$) are determined, as shown in Table 2. Because our detection method is novel and the threshold values are newly introduced in this paper, we use an empirical method by running the simulations to determine the threshold values without referring to the results from the literature. The $\boldsymbol{Thres_{int}}$ determines how frequently the detection mechanism is carried out. The lower $\boldsymbol{Thres_{int}}$, the more frequent detection mechanism we can perform. However, low $\boldsymbol{Thres_{int}}$ may increase the false alarm rates because of the short detection time intervals. Through architectural simulation, we empirically determine $\boldsymbol{Thres_{int}}$ as 100,000. The $\boldsymbol{Thres_{lower}}$, $\boldsymbol{Thres_{upper}}$, and $\boldsymbol{Thres_{acc}}$ should be determined by considering the program characteristics. $\boldsymbol{Thres_{lower}}$ and $\boldsymbol{Thres_{upper}}$ are related with the intensiveness of the shared resource accesses that are caused by the malicious codes. Figure 6 shows the range of the

RCounter for each workload (malicious codes and selected SPEC2006 workloads). The normal workloads show wider ranges of the RCounter values, while the malicious code only shows a limited range (roughly 600~1200). Based on this observation, we conservatively set the $\boldsymbol{Thres_{lower}}$ and $\boldsymbol{Thres_{upper}}$ as 600 and 1200, respectively. $\boldsymbol{Thres_{acc}}$ is related to the steadiness of the shared resource accesses caused by the malicious codes. From the simulations, we confirm that the normal programs are not likely to satisfy the condition in Equation (1) consecutively by 400. Thus, we conservatively set the $\boldsymbol{Thres_{acc}}$ as 500 to minimize the false alarm. Although there could be a little better threshold values for our environment (because our work determines the threshold values not formally, but empirically), our detection and mitigation techniques can still successfully thwart the attack, as will be shown in Sections 5.2 and 5.3. We will further investigate how to formally find the optimal threshold values for a certain environment and architecture as our future work. In terms of the hardware costs, the main part of our detection mechanism only requires three counters (RCounter, CCounter, and ACounter), three magnitude comparators, one multiplexor, and four registers (for threshold values). The hardware cost for our detection mechanism is negligible when considering that the modern mobile CPUs are already adopting an order of megabyte on-chip cache memories.

Table 2. Threshold values used for our detection mechanism.

	Values Used in This Work
$\boldsymbol{Thres_{int}}$	100,000
$\boldsymbol{Thres_{lower}}$	600
$\boldsymbol{Thres_{upper}}$	1200
$\boldsymbol{Thres_{acc}}$	500

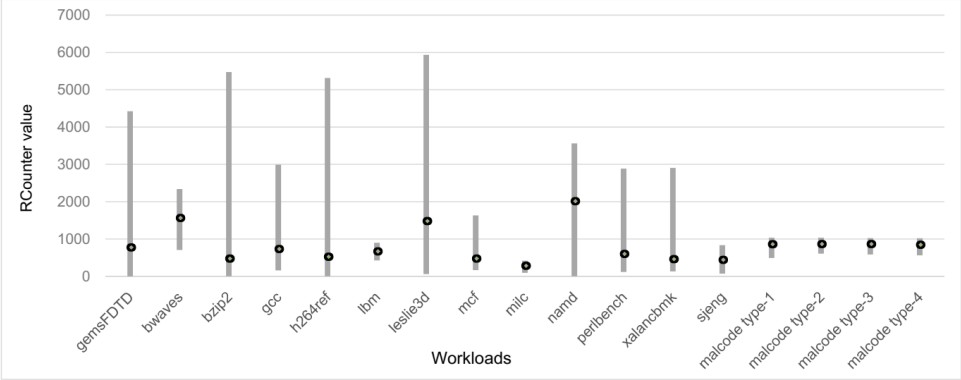

Figure 6. The ranges of RCounter values for each workload. The black-colored dot corresponds to the average value of RCounter while the gray bar corresponds to the range (i.e., min to max) of RCounter values.

Although our example malicious codes regularly and steadily issue L1 data cache miss requests to the shared L2 cache, the average RCounter values of some applications (e.g., bwaves, leslie3d, and namd) are actually higher than those of the malicious codes. These applications, which show higher RCounter values than the malicious codes, have two important characteristics: (1) high L1 data cache misses per cycle (intensiveness of the L1 data cache misses) and (2) low average L1 data cache miss latency (latency to obtain the data from the lower-level memories after the L1 data cache miss). Higher L1 data cache misses per cycle means that more L1 data cache miss requests are issued for a certain time period, which, in turn, leads to higher RCounter values. Low average L1 data cache miss latency means that many of the L1 data cache miss requests are served from the L2 cache (i.e., relatively low L2 cache miss rates) instead of the main memory, which implies that these workloads do not intensively access the main memory as much as our malicious

codes or other applications. In the case of low average L1 data cache miss latency, the CPU core pipeline or dependent instruction execution is likely to be stalled much less than in the case of high average L1 data cache miss latency. It makes the successive load/store instructions, which also have a high possibility to result in the L1 data cache miss, be issued relatively faster, which eventually increases RCounter values. On the other hand, when running the malicious codes, the average L1 data cache miss latency is much higher than those three applications. It means a non-negligible portion of the data requested from the L1 data cache is served from the main memory, resulting in resource contention in both L2 cache and main memory.

There would be a concern regarding how we can effectively detect variants of our malicious programs. For example, we could make our malicious codes fit into L2 caches (in order to cause L2 cache hits and try to evict the cache blocks of other programs), trying to prevent the malicious codes from accessing the main memory. In this case, we only cause DoS in the shared L2 caches. In addition, we may construct a malicious code to access memory arrays with irregular indexing sequences to increase row buffer misses in the DRAM main memory instead of linearly increasing the index with a certain stride. It may cause the higher latency to serve the data from the main memory, which also wastes more main memory bandwidth for benign programs. However, the main characteristic of our malicious code and its variants is to execute the load/store instruction regularly and consecutively (i.e., steadily), which causes DoS to the shared resources in the system. In order to efficiently characterize the shared resource access pattern of the malicious code, we focus on the point (i.e., location) at which the malicious code and its variants inevitably show a very similar pattern. To accomplish it, our detection mechanism monitors a connection port (where we can monitor the L1 data cache miss request pattern) between the L1 data cache and shared L2 cache. In addition, our example variants of the malicious codes can use multiple different functional units (such as floating point ALUs in Figure 4d) along with the memory accesses, as shown in Figure 4. It could circumvent the malicious code detection that uses the performance counters or functional unit usage characteristics, while our proposed mechanism can successfully detect the malicious codes for the memory and cache contention DoS attack.

The regular and data-intensive applications (e.g., image processing) might be falsely detected as a malicious code when adopting our detection mechanism because the mobile domain typically has wider spectrum of applications compared to the high-performance domain. However, mobile devices are employing various accelerators to improve performance of those regular and data-intensive applications [27]. For example, the image or graphical processing can be performed in graphic processing units (GPUs) or other image processing accelerators, such as digital signal processors (DSPs). On the other hand, the mobile CPUs are more focusing on control-intensive programs in mobile devices. Although the attacker may launch the memory contention attack with those GPUs or DSPs, it only has a limited impact, because it can only cause the main memory contention. In addition, exploiting GPUs or DSPs requires much more efforts for attackers to successfully attack the system because the attacker should use specialized application programming interfaces (APIs) to employ those hardware accelerators. On the contrary, our simple C-based malicious code can easily and simply be executed in mobile CPUs. Because mobile devices are employing more specialized accelerators, the possibility of falsely detecting those mobile applications (such as image processing) as malicious codes would be lowered.

Our proposed detection method may be subject to the adversarial mimicry attack. However, it requires a huge effort on the attacker side, which can be justified, as follows. In order to evade the detection mechanism, the attacker should satisfy the following condition: there should be at least one interval that satisfies RCounter value is out of the range between the $\boldsymbol{Thres_{lower}}$ and $\boldsymbol{Thres_{upper}}$ within $\boldsymbol{Thres_{acc}}$ (or less) consecutive intervals. In other words, the mimicry attacker should put a code that generates slower or extremely faster memory accesses at least once a $\boldsymbol{Thres_{acc}}$ interval for evasion of the detection. However, it is very hard to be achieved, because the attacker cannot know the exact threshold

parameters (such as $\boldsymbol{Thres_{lower}}$ and $\boldsymbol{Thres_{upper}}$) set for a certain mobile device. In order to figure out the threshold parameters for the detection, the attacker should put huge effort on reverse engineering by testing a huge number of malicious code variants, which is not practically feasible.

4.2. Our Proposed Mitigation Method

4.2.1. Instruction Fetch Bandwidth Throttling

Once we detect the memory and cache contention DoS attack in the system, we should apply a mitigation method that can neutralize the consequences of the attack. Because the memory and cache contention DoS attack is mainly caused by resource contentions between the normal and malicious programs, restricting malicious program's accesses to the shared resources would be an effective solution for mitigation. In this paper, we propose reducing instruction fetch bandwidth when the malicious program is detected by our detection mechanism. We employ the instruction fetch throttling to reduce instruction fetch bandwidth, which dynamically adjusts the fetch width of a certain processing core. When our detection mechanism sends the mitigation mechanism trigger signal, it reduces the instruction fetch width from 3 to 0 by controlling the MUX selection signal, as shown in Figure 7. When the instruction fetch width of a certain core becomes 0, the core cannot fetch the instruction, fully restricting malicious program's accesses to the shared resources. In terms of hardware overhead, our mitigation mechanism can be implemented with one multiplexor and the fetch width control unit, as already shown in Figure 7. The fetch width control unit can be implemented by temporarily halting (e.g., clock gating) the instruction fetch units. Please note that the clock gating scheme is very widely used in digital circuits.

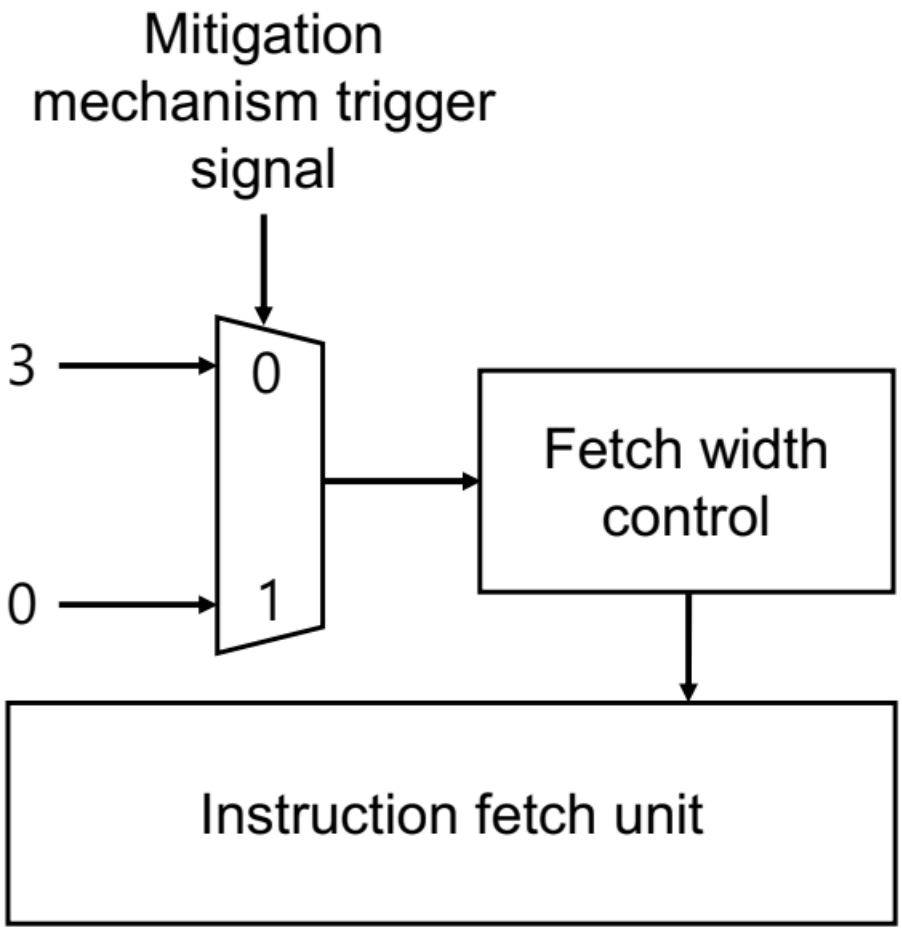

Figure 7. Our dynamic instruction fetch width throttling technique.

4.2.2. Throttling Periods

It is also crucial to determine how long we should maintain the throttling in order to fully suppress the effect of the malicious memory and cache contention. Figure 8 describes an example timing diagram for our throttling engagement and disengagement. To trigger the throttling mechanism, we should spend at least $\boldsymbol{Thres_{acc}} \times \boldsymbol{Thres_{int}}$ cycles (in this work, $500 \times 100,000$ cycles) for profiling. Once our throttling mechanism is engaged, it lasts 500 million processor clock cycles. After that, the throttling mechanism is disengaged and the CPU returns back to the normal execution mode while resetting the profiling counters (i.e., CCounter, RCounter, and ACounter) to '0'.

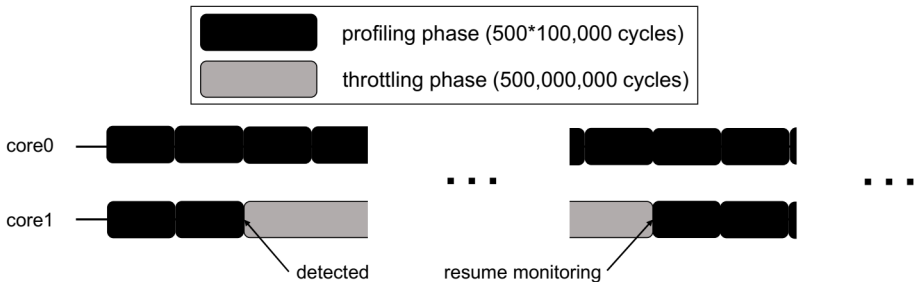

Figure 8. An example timing diagram for the instruction fetch width throttling.

We set the throttling phase to be ten times longer than the profiling phase, as shown in Figure 8. It means that we can recover performance and energy by at least over 90%, on average, as long as our detection mechanism correctly detects the malicious codes. Because our detection mechanism is hardware-based one that profiles runtime program execution patterns, we cannot forcibly terminate the detected malicious code (although the profiling phase is much shorter than the throttling phase), which is why we cannot achieve 100% recovery rates. This is a clear limitation of a hardware-based approach (the similar approach was also employed in [9]). 500 million cycles correspond to 250 ms in our simulated system, with 2 GHz CPU. We believe that it would be sufficient time to minimize the impact of the attack. Obviously, the recovery rate will vary, depending on the time periods of profiling and throttling phases though the investigation on the relationship between the recovery rate and profiling/throttling time periods is out-of-scope of this paper.

The instruction fetch throttling is independently performed across the processing cores, because the fetch width throttling should be applied to each core. For example, once our detection mechanism detects the malicious program in Core1, only Core1 is engaged to the instruction fetch width throttling while the Core0 is not engaged. Once 500 million cycles have been spent since the throttling was engaged in Core1, it resumes the execution by recovering the instruction fetch bandwidth from '0' to '3' in Core1, returning back to the profiling phase.

5. Evaluation

5.1. Evaluation Methodology

For evaluations of the attack and our detection and mitigation methods, we model a mobile edge computing system in gem5 architectural simulation tool [28], as described in Section 3.1. The performance of the system is measured as IPC (instructions per cycle) of the normal program. Please note that performance of the malicious code is meaningless, as it is only used to cause DoS to the normal programs. For normal programs, we use a set of SPEC2006 workloads (selected 13 workloads) [29]. For the accuracy of the simulations, we show the results of running 1 billion instructions after we fastforward two billion instructions. The instruction count is measured based on SPEC2006 programs for fair comparisons. In other words, we only count the number of executed instructions of the normal programs, while the executed instruction count of the malicious program is ignored. For energy evaluations, we use McPAT [30] with 32 nm process technologies. In our experiments, we assume that the malicious code was already injected to the system,

because our focus is to unveil adversarial performance and energy impacts from our DoS attack. The attacker can easily inject and execute the malicious codes using various methods, as we explained in Sections 3.1 and 3.2 [31]. The malicious codes can also be injected via downloading the malicious apps or phishing, as reported in [32].

In this paper, we compare performance and energy across three broad cases: *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*, where *X* represents the number of the malcode type (type-*X* in Figure 4). *spec* represents the case, where only the normal program is executed in the system. *spec+mal_tX* corresponds to the case where the normal program and malicious code are running in the system simultaneously without our detection and mitigation techniques. *spec+mal_tX_ft* represents the cases where our malicious code detection and mitigation techniques are applied when running both normal and malicious programs. We have a total of nine different cases for comparisons, as we have four types of the malicious codes.

Our detection and mitigation mechanisms are geared towards our memory and cache contention DoS attack and corresponding malicious codes. Thus, a comparison between the detection and mitigation mechanisms for different types of the attacks may result in unfair comparisons. Thus, we perform the comparison among the three cases explained above (*spec*, *spec+mal_tX*, and *spec+mal_tX_ft*).

5.2. Performance

Figure 9 shows the results of performance comparison for nine cases. When we compare *spec* and *spec+mal_t1*, the performance of *spec+mal_t1* is degraded by 47.7%, on average, as compared to *spec*. In the case of *xalancbmk*, the performance degradation is up to 78.0%, which means that the malicious memory and cache contention incurs huge performance losses of the normal programs running at the same time. This is because of the shared resource (L2 cache and main memory) contention between the normal and malicious program. In the cases of the other types of the malicious codes, the overall trends are also similar to the results of *spec+mal_t1*. The performance degradations of the normal programs due to malcode type-2, type-3, and type-4 are 47.7%, 47.8%, and 47.3%, respectively, as shown in Figure 9.

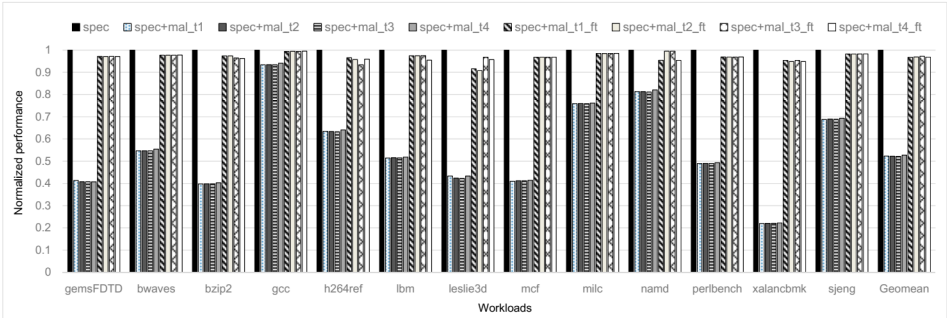

Figure 9. Performance comparison across three broad cases (total nine cases): *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*. The results are normalized to *spec* for each workload.

On the contrary, if our detection and mitigation mechanisms are employed, the performance of the normal programs is improved, as they restrict the shared resource usage of the malicious program. On average, with our detection and mitigation mechanisms (i.e., in the cases of *spec+mal_tX_ft*) performances of the normal programs are improved by 84.9%, 85.4%, 86.1%, and 83.6% as compared to the cases of *spec+mal_t1*, *spec+mal_t2*, *spec+mal_t3*, and *spec+mal_t4*, respectively.

Table 3 shows the performance recovery rates. The performance recovery rates ($R_{perf\_recovery}$) are calculated, as follows:

$$R_{perf\_recovery} = 1 - \frac{IPC_{SPEC} - IPC_{SPEC+mal\_tX\_ft}}{IPC_{SPEC} - IPC_{SPEC+mal\_tX}} \quad (2)$$

where IPC_{SPEC}, $IPC_{SPEC+mal\_tX}$, and $IPC_{SPEC+mal\_tX\_ft}$ represents IPC in the cases of *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*, respectively. In the case of *spec+mal_t1_ft*, a performance recovery rate is 91.7% on average. One of the main reasons why we cannot obtain 100% performance recovery rate is that there must be a profiling phase to detect the malicious code before the throttling phase. If we would increase the duration of each throttling phase (i.e., more than 500 million cycles), then we could further improve performance recovery rates though it may also lead to performance losses of the normal programs in case of misdetection (i.e., false alarm). For the other types of the malicious codes, our detection and mitigation mechanisms also result in very high $R_{perf_recovery}$s (93.1%, 93.4%, and 91.5% in the cases of *spec+mal_t2_ft*, *spec+mal_t3_ft*, and *spec+mal_t4_ft*, respectively).

Table 3. $R_{perf\_recovery}$ for each workload in the cases of *spec+mal_tX_ft*.

	spec+mal_t1_ft	***spec+mal_t2_ft***	***spec+mal_t3_ft***	***spec+mal_t4_ft***
gemsFDTD	95.1%	95.1%	95.2%	95.1%
bwaves	95.0%	95.0%	95.0%	95.0%
bzip2	95.7%	95.7%	94.1%	93.7%
gcc	91.1%	91.3%	91.0%	91.5%
h264ref	90.5%	88.5%	81.9%	88.7%
lbm	94.8%	94.7%	94.8%	90.6%
leslie3d	85.2%	83.9%	94.5%	92.5%
mcf	94.6%	94.6%	94.6%	94.6%
milc	93.9%	93.8%	93.9%	93.9%
namd	75.6%	97.3%	97.4%	74.3%
perlbench	94.0%	93.9%	93.9%	93.9%
xalancbmk	94.1%	93.4%	94.1%	93.3%
sjeng	94.3%	94.3%	94.3%	94.3%
Geomean	91.7%	93.1%	93.4%	91.5%

With our detection mechanism, the RCounter pattern that is shown in Figure 6 should be consecutively seen by $Thres_{acc}$ (i.e., 500 in this paper) times to be regarded as the malicious code. As a result, during our simulation, we did not see any false alarm case (i.e., falsely detect and actually throttle the normal program) for 13 SPEC2006 programs. When considering that our throttling phase is 500 million cycles while we simulate total one billion instructions, even with only one false alarm and throttling case, a serious performance loss in the SPEC2006 programs will occur. However, we can see a performance recovery rate of 92.4%, on average, which is attributed to zero false alarm case in our simulations, as shown in our recovery rate results. This is because we set the threshold values very conservatively to prevent false alarms. It also implies that our proposed mechanism accurately differentiates normal and malicious programs.

Although the worst-case $R_{perf\_recovery}$ is still higher than 74%, in some workloads, they show relatively lower $R_{perf_recovery}$s. For example, in the case of *namd*, $R_{perf_recovery}$s when running malicious code type-1 and type-4 are relatively lower than the cases when running type-2 and type-3. Because of the conservative $Thres_{upper}$ and $Thres_{lower}$, our detection mechanism may falsely regard the malicious program as normal one, which, in turn, leads to a little lower $R_{perf\_recovery}$. However, it is very rare case, as relatively low $R_{perf_recovery}$s are only shown in several workloads. Although we could increase the range of $Thres_{upper}$ and $Thres_{lower}$, it might also increase the false alarm rates. Therefore, we use relatively conservative $Thres_{upper}$ and $Thres_{lower}$, since it still shows over 90% $R_{perf_recovery}$s on average.

5.3. Energy Comparison

Figure 10 depicts the results of energy comparison for nine cases: *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*, where *X* is 1–4. The memory and cache contention DoS attack not only causes significant performance degradation, but also significantly increases the energy consumption of the system. In the case of *spec+mal_t1*, the energy consumption of the system increases by 2.1× on average. In the case of *xalancbmk*, system energy consumption increases by up to 4.8×. It means that the memory and cache contention DoS attack maliciously consumes system energy, which may eventually result in catastrophic impacts on availability and user experience in mobile edge devices. In the other cases, where we run different types of the malicious codes, it also shows similar energy increases (~2.1×).

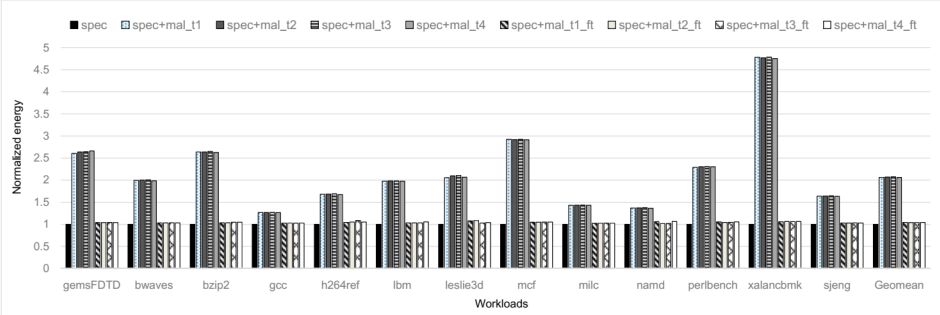

Figure 10. Energy comparison across three broad cases (total nine cases): *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*. The results are normalized to *spec* for each workload.

When we employ our detection and mitigation techniques, we can also reduce energy consumption that is caused by the attack. When applying our detection and mitigation techniques (*spec+mal_tX_ft*), energy consumption is significantly reduced by 43.5% on average, as compared to *spec+mal_tX*. When comparing the energy consumptions between *spec+mal_tX_ft* and *spec*, *spec+mal_tX_ft* leads to more energy consumption only by 4.4% when compared to the case of *spec*.

Table 4 summarizes the energy recovery rates ($R_{energy\_recovery}$) when employing our detection and mitigation method across the 13 SPEC2006 workloads. The energy recovery rates are calculated, as in Equation (3):

$$R_{energy\_recovery} = 1 - \frac{E_{SPEC+mal\_tX\_ft} - E_{SPEC}}{E_{SPEC+mal\_tX} - E_{SPEC}} \quad (3)$$

where E_{SPEC}, $E_{SPEC+mal\_tX}$, and $E_{SPEC+mal\_tX\_ft}$ represent energy consumptions of the CPU in the case of *spec*, *spec+mal_tX*, and *spec+mal_tX_ft*, respectively. When we employ the detection and mitigation techniques (*spec+mal_tX_ft*), we can recover energy consumption by 94.7% on average, leaving only a marginal gap between energy consumptions in the cases of *spec* and *spec+mal_tX_ft*. In summary, our detection and mitigation techniques successfully recover malicious energy consumption from the memory and cache contention DoS attack, which eventually results in longer battery life as well as better availability and user experience in mobile edge devices.

Table 4. $R_{energy\_recovery}$ for each workload in the cases of *spec+mal_tX_ft*.

	spec+mal_t1_ft	***spec+mal_t2_ft***	***spec+mal_t3_ft***	***spec+mal_t4_ft***
gemsFDTD	97.4%	97.4%	97.3%	97.4%
bwaves	96.5%	96.5%	96.4%	96.5%
bzip2	97.7%	97.7%	97.0%	96.8%
gcc	89.9%	89.9%	89.4%	89.8%
h264ref	93.4%	92.1%	87.4%	92.1%
lbm	96.6%	96.6%	96.5%	94.3%
leslie3d	92.6%	92.2%	97.1%	96.1%
mcf	97.3%	97.3%	97.2%	97.2%
milc	94.1%	94.0%	93.8%	94.1%
namd	83.1%	94.9%	94.7%	82.6%
perlbench	95.7%	96.5%	96.4%	95.7%
xalancbmk	98.4%	98.3%	98.4%	98.3%
sjeng	95.2%	95.2%	95.0%	95.1%
Geomean	94.4%	95.3%	95.1%	94.2%

6. Conclusions

In this paper, we introduce memory and cache contention attack, which causes denial-of-service (DoS) and exhaustive energy consumption in mobile edge devices. The memory and cache contention DoS attack can be realized with a simple malicious code that performs consecutive memory data transfer operations. The malicious program mainly hinders shared resource accesses of the normal programs, incurring performance degradation of the normal programs and excessive energy consumption in the system. We propose profiling the intensiveness and steadiness of the L1 data cache miss request issues in order to detect and mitigate the memory and cache contention DoS attack. By utilizing the differences between the workload characteristics of normal and malicious program, our proposed detection technique successfully detects the malicious program for the memory and cache contention DoS attack with near-zero false alarm rates. To mitigate the memory and cache contention DoS attack, we propose employing an instruction fetch bandwidth throttling. Our evaluation results demonstrate how much the memory and cache contention DoS attack can degrade performance (by 47.6% on average) and cause excessive energy consumption (by 2.1× on average). In addition, our detection and mitigation techniques successfully alleviate the adverse impacts of the memory and cache contention DoS attack. With our detection and mitigation techniques, the performance and energy of the normal programs are recovered by 92.4% and 94.7% (on average), respectively. As our future work, we will perform the detailed sensitivity study with various profiling and throttling time periods and then investigate broader types of memory and cache contention attacks (considering various accelerators) in mobile system-on-chips. We will also investigate the synergistic effects by combining the hardware- and software-level detection and mitigation mechanisms.

Author Contributions: Conceptualization, W.C. and J.K.; methodology, W.C. and J.K.; Simulation, W.C.; Result analysis, W.C. and J.K.; Writing, W.C. and J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A3B07045908) and the Ministry of Science, ICT & Future Planning (NRF-2015R1C1A1A01051836).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on reasonable request from the corresponding author. The additional experiment data are not publicly available due to the reason that all the experiment data are already presented in this article's figures and tables.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Halpern, M.; Zhu, Y.; Reddi, V.J. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 12–16 March 2016; pp. 64–76.
2. Yang, S.H.; Lee, S.; Lee, J.Y.; Cho, J.; Lee, H.J.; Cho, D.; Heo, J.; Cho, S.; Shin, Y.; Yun, S.; et al. A 32nm high-k metal gate application processor with GHz multi-core CPU. In Proceedings of the 2012 IEEE International Solid-State Circuits Conference, San Francisco, CA, USA, 19–23 February 2012; pp. 214–216.
3. Moscibroda, T.; Mutlu, O. Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. In Proceedings of the 16th USENIX Security Symposium on USENIX Security Symposium, Boston, MA, USA, 6–10 August 2007; pp. 18:1–18:18.
4. Woo, D.H.; Lee, H.H. Analyzing performance vulnerability due to resource denial of service attack on chip multiprocessors. In Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnects in conjunction with the 13th Annual International Conference on High-Performance Architecture, Phoenix, AZ, USA, 11 February 2007.
5. Zhang, T.; Zhang, Y.; Lee, R.B. DoS Attacks on Your Memory in Cloud. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS'17), Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 253–265.
6. Bechtel, M.; Yun, H. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Montreal, QC, Canada, 16–18 April 2019; pp. 357–367.
7. Bechtel, M.; Yun, H. Memory-Aware Denial-of-Service Attacks on Shared Cache in Multicore Real-Time Systems. *arXiv* **2020**, arXiv:2005.10864.
8. Hasan, J.; Jalote, A.; Vijaykumar, T.N.; Brodley, C.E. Heat stroke: Power-density-based denial of service in SMT. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, 12–16 February 2005; pp. 166–177.
9. Kong, J.; John, J.K.; Chung, E.Y.; Chung, S.W.; Hu, J. On the Thermal Attack in Instruction Caches. *IEEE Trans. Dependable Secur. Comput.* **2010**, *7*, 217–223. [CrossRef]
10. Aciicmez, O. Yet another microarchitectural attack: Exploiting i-cache. In Proceedings of the 2007 ACM Workshop on Computer Security Architecture, Fairfax, VA, USA, 2 November 2007; pp. 11–18.
11. Spreitzer, R.; Griesmayr, S.; Korak, T.; Mangard, S. Exploiting Data Usage Statistics for Website Fingerprinting Attacks on Android. In Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WISEC 2016, Darmstadt, Germany, 18–22 July 2016; pp. 49–60.
12. Yarom, Y.; Genkin, D.; Heninger, N. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In Proceedings of the Cryptographic Hardware and Embed ded Systems-CHES 2016-18th International Conference, Santa Barbara, CA, USA, 17–19 August 2016; pp. 346–367.
13. Bernstein, D.J. Cache-timing attacks on AES. 2004. Available online: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (accessed on 4 January 2018).
14. Christodorescu, M.; Jha, S. Static Analysis of Executables to Detect Malicious Patterns. In Proceedings of the 12th Conference on USENIX Security Symposium, Washington, DC, USA, 4–8 August 2003; Volume 12, pp. 1–18.
15. Christodorescu, M.; Jha, S.; Seshia, S.A.; Song, D.; Bryant, R.E. Semantics-aware malware detection. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05), Oakland, CA, USA, 8–11 May 2005; pp. 32–46.
16. Kinder, J.; Katzenbeisser, S.; Schallhart, C.; Veith, H. Detecting Malicious Code by Model Checking. In Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Vienna, Austria, 7–8 July 2005; pp. 174–187.
17. Kolter, J.Z.; Maloof, M.A. Learning to Detect and Classify Malicious Executables in the Wild. *J. Mach. Learn. Res.* **2006**, *7*, 2721–2744.
18. Schmidt, A.D.; Bye, R.; Schmidt, H.G.; Clausen, J.; Kiraz, O.; Yuksel, K.A.; Camtepe, S.A.; Albayrak, S. Static Analysis of Executables for Collaborative Malware Detection on Android. In Proceedings of the 2009 IEEE International Conference on Communications, Dresden, Germany, 14–18 June 2009; pp. 1–5.
19. Demme, J.; Maycock, M.; Schmitz, J.; Tang, A.; Waksman, A.; Sethumadhavan, S.; Stolfo, S. On the Feasibility of Online Malware Detection with Performance Counters. In Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 23–27 June 2013; pp. 559–570.
20. Ozsoy, M.; Donovick, C.; Gorelik, I.; Abu-Ghazaleh, N.; Ponomarev, D. Malware-aware processors: A framework for efficient on-line malware detection. In Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), Burlingame, CA, USA, 7–11 February 2015; pp. 651–661.

21. Eklov, D.; Nikoleris, N.; Black-Schaffer, D.; Hagersten, E. Cache Pirating: Measuring the Curse of the Shared Cache. In Proceedings of the 2011 International Conference on Parallel Processing, Taipei, Taiwan, 13–16 September 2011; pp. 165–175.
22. Eklov, D.; Nikoleris, N.; Black-Schaffer, D.; Hagersten, E. Bandwidth Bandit: Quantitative characterization of memory contention. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Shenzhen, China, 23–27 February 2013; pp. 1–10.
23. Subramanian, L.; Seshadri, V.; Ghosh, A.; Khan, S.; Mutlu, O. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, USA, 5–9 December 2015; pp. 62–75.
24. ARM Processor Introduction. Cortex-A15 MPCore Processor. 2011. Available online: http://www.arm.com/products/processors/cortex-a/cortex-a15.php (accessed on 15 July 2017).
25. Choi, H.; Kim, Y. Large-Scale Analysis of Remote Code Injection Attacks in Android Apps. *Secur. Commun. Netw.* **2018**, *2018*, 1–17. [CrossRef]
26. Wang, J.; Liu, K.; Pan, J. Online UAV-Mounted Edge Server Dispatching for Mobile-to-Mobile Edge Computing. *IEEE Internet Things J.* **2020**, *7*, 1375–1386. [CrossRef]
27. Reddi, V.J. Mobile SoCs: The Wild West of Domain Specific Architectures. ACM SigArch, Computer Architecture Today. Available online: https://www.sigarch.org/mobile-socs/ (accessed on 18 April 2020).
28. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News* **2011**, *39*, 1–7. [CrossRef]
29. Spradling, C.D. SPEC CPU2006 Benchmark Tools. *SIGARCH Comput. Archit. News* **2007**, *35*, 130–134. [CrossRef]
30. Li, S.; Ahn, J.H.; Strong, R.D.; Brockman, J.B.; Tullsen, D.M.; Jouppi, N.P. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, New York, NY, USA, 12–16 December 2009; pp. 469–480.
31. Jin, X.; Hu, X.; Ying, K.; Du, W.; Yin, H.; Peri, G.N. Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, USA, 3–7 November 2014.
32. 5 Ways Your Mobile Device Can Get Malware. Available online: https://www.securitymetrics.com/blog/5-ways-your-mobile-device-can-get-malware (accessed on 24 February 2021).