Article

CacheHawkeye: Detecting Cache Side Channel Attacks Based on Memory Events

Hui Yan [1,2] and Chaoyuan Cui [2,*]

[1] Institutes of Physical Science and Information Technology, Anhui University, Hefei 230601, China; jackyan951001@gmail.com
[2] Institutes of Intelligent Machines, Hefei Institutes of Physical Sciences, Chinese Academy of Sciences, Hefei 230031, China
* Correspondence: cycui@iim.ac.cn

Abstract: Cache side channel attacks, as a type of cryptanalysis, seriously threaten the security of the cryptosystem. These attacks continuously monitor the memory addresses associated with the victim's secret information, which cause frequent memory access on these addresses. This paper proposes *CacheHawkeye*, which uses the frequent memory access characteristic of the attacker to detect attacks. *CacheHawkeye* monitors memory events by CPU hardware performance counters. We proved the effectiveness of *CacheHawkeye* on Flush+Reload and Flush+Flush attacks. In addition, we evaluated the accuracy of *CacheHawkeye* under different system loads. Experiments demonstrate that *CacheHawkeye* not only has good accuracy but can also adapt to various system loads.

Keywords: cryptanalysis; side channel; Flush+Reload; cache side channel; Flush+Flush; side channel detection

Citation: Yan, H.; Cui, C. CacheHawkeye: Detecting Cache Side Channel Attacks Based on Memory Events. *Future Internet* **2022**, *14*, 24. https://doi.org/10.3390/fi14010024

Academic Editors: Ji-Hwei Horng, Chia-Chen Lin and Ching-Chun Chang

Received: 8 December 2021
Accepted: 7 January 2022
Published: 8 January 2022

1. Introduction

The security of a cryptosystem can be compromised via cryptanalysis. Cache side channel cryptanalysis is a type of cryptanalysis. Cache was implemented into current CPUs to alleviate the speed disparity between the CPU and memory. L1 cache, L2 cache, and L3 cache are the three levels that it is divided into. All cores share the largest L3 cache. Even if the operating system provides strong process isolation, an attacker can utilize a shared L3 cache as a side channel to steal the victim's confidential information [1–3]. Time-driven attacks and trace-driven attacks are two types of cache side channel attacks. Trace-driven attacks include Flush+Reload [2] and Flush+Flush [3]. These two attacks take advantage of shared memory technology and leverage the L3 cache as a side channel for data leakage.

Recently, researchers have proposed many countermeasures to mitigate cache side channel attacks. Wang et al. [4] designed a dynamic cache partition strategy to protect the L3 cache. Zhou et al. [5] proposed a method for preventing attackers and victims from sharing memory by dynamically managing physical memory pages between security domains. Oliverio et al. [6] changed shared page from copy-on-write to copy-on-access to prohibit attackers and victims from sharing memory. These countermeasures require major modifications to hardware and operating system. These approaches cannot be applied to current computer systems because modifications on the hardware cannot be deployed to existing systems immediately and modifications on the operating system incur a performance overhead that is hard to ignore.

Therefore, there is an urgent need for a detection technology that does not modify software and hardware. The detection techniques for these attacks are full of challenges. First of all, cache side channel attack does not have any malicious behaviors, it is difficult for traditional malware detection methods to detect these attacks. Secondly, existing detection methods monitor cache hits and cache misses of programs to detect cache side channel attacks. However, cache hits and misses are susceptible to interference from other workloads, which ultimately affect the detection accuracy. Finally, because Flush+Flush has a low cache hit rate, it is difficult for many detection methods to detect this attack.

This article focuses on cache side channel attacks that exploiting shared memory. To detect such attacks, we design *CacheHawkeye*. Our approach is based on the assumption that cache side channel attacks frequently access specific memory addresses, so we detect these attacks through frequent memory access. *CacheHawkeye* monitors memory events by using CPU hardware performance counters(HPCs). To the best of our knowledge, this is the first time memory events have been used to detect cache side channel attacks. These attacks infer the victim's privacy by continuously monitoring specific memory addresses, which generate frequent memory access on the specific memory addresses. *CacheHawkeye* detects these attacks based on these memory addresses and corresponding data symbols. According to our experiments, *CacheHawkeye* not only provides good accuracy, but also adapts to different system loads.

2. Background and Related Work

2.1. Cache Side Channel Attacks

Flush+Reload and Flush+Flush are two types of cache side channel attacks. They both take the last-level cache as a covert channel for information leakage. Flush+Reload and Flush+Flush rely on the operating system's or hypervisor's shared memory technology, whereas Prime+Probe [7–9] does not. Flush+Reload and Flush+Flush pose a great threat to information security, so we study the detection method of these two attacks. In the scenario of these attacks, there is a victim and an attacker. Typically, the victim is a cryptographic program. The attacker's goal is to steal the victim's secret key.

When the victim is running, it accesses some memory addresses. We call these memory addresses associated with the secret key sensitive memory addresses. The attacker monitors these memory addresses for a long time and collects the victim's sensitive memory access sequences. These memory access sequences can be used by the attacker to deduce secret key bits. Flush+Reload, for example, constantly monitors the RSA cryptosystem's memory access sequences and can retrieve 96.7% of the secret key bits in a round of attack.

We explain in detail how to steal RSA and AES keys. Square and Multiply modular exponentiation of RSA in GnuPG 1.4.13 and T-table implementation of AES in OpenSSL 0.9.8 are vulnerable to cache side channel attacks. The following experiments are carried out with these two versions. The attacker monitors the cache to get the victim's memory access sequences. The memory access sequences of a cryptographic program depend on the value of the key bit. The attacker steals the secret key by monitoring the memory access sequences of some sensitive functions' addresses. For example, in the Square-and-Multiply implementation of RSA, the Square-Reduce memory access sequence represents the key bit is 0 and Square-Reduce-Mul-Reduce represents the key bit is 1. The AES algorithm uses the T-tables to compute the ciphertext based on the secret key k and the plaintext p. During the first round, table accesses are made to entries $T_j[p_i \oplus k_i]$ with $i \equiv j \bmod 4$ and $0 \leq i < 16$. Cache attack is possible to derive values for $p_i \oplus k_i$ and thus, possible key-byte values k_i in case p_i is known.

Trace-driven attacks manipulate traces of the victim's cache access. Trace-driven attacks include Flush+Reload and Flush+Flush. In these attacks, the attacker repeatedly checks and analyzes the status of cache lines used by the victim [10]. The process of trace-driven attacks is depicted in Figure 1. Each monitoring round, the attacker manipulates cache lines and measures the accessing time. At most a bit of secret key can be recovered in each monitoring round. At least a few thousand monitoring rounds are required for a successful attack. Because the attacker does not know when the victim begins executing in the real world, monitoring time will be longer. The attacker monitors cache by manipulating cache lines, which causes frequently access to specific memory addresses.

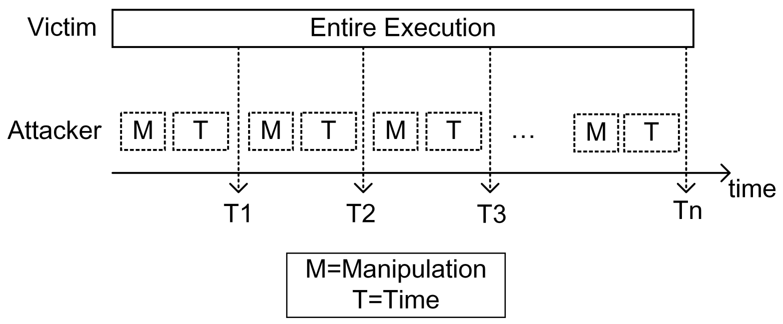

Figure 1. Process of trace-driven attacks.

2.2. Related Work

Cache side channel detection technology could be divided into two main categories: signature-based detection [11–16] and anomaly-based detection [17–19]. Some detection techniques [20–22] use a combination of signature and anomaly-based detection techniques.

Some researchers propose several signature-based detection methods [11–16]. For example, Demme et al. [11] use L1 hits event in HPCs to detect malware and cache side channel attacks. Allaf et al. [12] propose another signature-based detection technique to detect Prime+Probe and Flush+Reload attacks when using machine learning (ML) models and HPCs and running an AES cryptosystem. The hardware events they use are core cycles, reference cycles and core instructions. NIGHTs-WATCH [13] can detect access-driven cache side channel attacks. It consists of various machine learning models that make use of LLC misses and CPU cycles in HPCs. This method trains the model under some preset system loads, and we are not sure whether it still performs well under some unknown system loads. Mushtaq et al. [14] use linear and non-linear ML classifiers to detect variants of Prime+Probe attacks running under the AES cryptosystem. HexPADS [15] uses the values of cache miss rates and page faults to detect Flush+Reload and cache template attacks [16].

The literature [17–19] proposes several anomaly-based detection methods. For example, CacheShield [17] is an anomaly-based detection mechanism on legacy software (victim application) that involves monitoring L3 cache misses by HPCs. Bazm et al. [18] detect cross-VM cache side channel attacks through using hardware fine-grained information provided by Intel Cache Monitoring Technology (CMT) and HPCs following the Gaussian anomaly detection method. SpyDetector [19] can detect Flush+Reload, Flush+Flush and Prime+Probe attacks running on RSA, AES and ECDSA cryptosystems through monitoring L3 cache and L1 data cache by HPCs.

Some detection techniques [20–22] use the combination of signature and anomaly-based detection techniques. Chiappetta et al. [20] propose a machine learning based detection mechanism for Flush+Reload attack on AES and ECDSA cryptosystem. They monitor L3 access to detect attacks. CloudRadar [22] is a signature and anomaly-based detection system. Attacks are detected in two steps. The first step is to detect cryptographic applications by branch instructions and dynamic time warping. The second step is to define a criterion for identifying between benign and malignant programs. CloudRadar believes an attack has occurred when the detected value surpasses this criterion. Alam et al. [21] present a multi-layer detection approach based on machine learning. This approach collects microarchitecture events(e.g. branch misses, LLC accesses and LLC misses) when attacks occur. Alam et al. train some machine learning models based on these events to detect attacks.

The above work uses microarchitecture events as feature vectors for detection. Because the component capacity of the microarchitecture is very small, it is very susceptible to interference from the system load. This paper, unlike the previous work, detects cache side channel attacks by memory events. The memory capacity is very large, so the interference of the system load can be ignored. We will prove this point through experiments in Section 4.3. Moreover, our approach does not require pre-trained models, which makes it more adaptable to unknown system loads and hardware environments.

2.3. Hardware Performance Counters

Hardware performance counters(HPCs) are a set of special registers built into x86 (e.g., Intel and AMD) and ARM processors. They use event selectors to track certain hardware events and update counters as they happen. Most modern processors provide performance monitoring units that enable applications to control these counters.

HPCs were originally designed for software debugging and system performance optimization. Later, researchers used these counters to detect security vulnerabilities [11,23,24]. These counters can display the execution characteristics of the program, which can further reflect the security status of the program. Microarchitecture events (e.g., cache loads and cache misses) can be monitored by hardware performance counters.

Perf tool of Linux system provides an interface for controlling HPCs. Only privileged users can use HPCs in the Linux system. The *perf mem record* command is used to monitor an unknown application, while the *perf mem report* command is used to display the result. The end result is a ten-column table. We are most interested in the data symbol column and the overhead column. The overhead column indicates the percentage of the sample collected in a specific function (or memory address) to the total sample. The data symbol column displays the address of the memory location that the row was targeting.

3. *CacheHawkeye* Design

3.1. Overview of CacheHawkeye

We designed a system named *CacheHawkeye* to detect cache side channel attacks. It detects cache side channel attacks through frequent memory access on specific addresses. The memory events monitored by the *perf* tool automatically associate these sensitive memory addresses with data symbols. *CacheHawkeye* detects attacks based on these data symbols. The structure of *CacheHawkeye* is shown in Figure 2. *CacheHawkeye* consists of three modules: monitor, cache attack library, and detector. In the cloud scenario, the hypervisor is responsible for the implementation of *CacheHawkeye*.

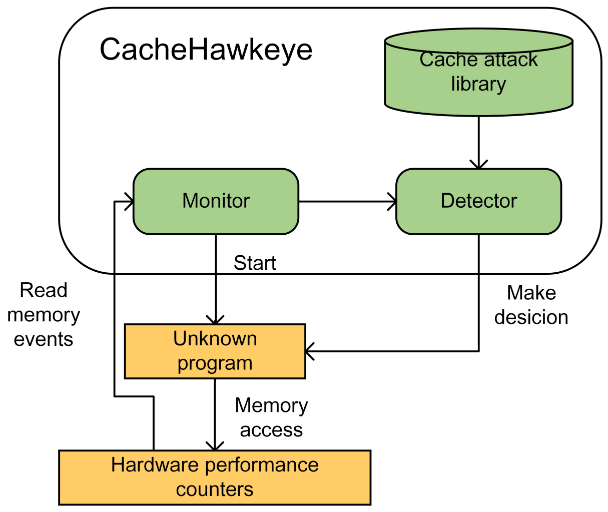

Figure 2. *CacheHawkeye* architecture.

The monitor controls the execution of unknown programs and reads the results of HPCs. We use the *perf* tool to monitor unknown programs. Only privileged users can enable the monitor. The monitoring command is *perf mem record -U./programName*. *-U* parameter is used to monitor events in user mode, which can avoid interference from memory activity in kernel mode. *programName* is a binary file compiled by an unknown program. When the execution is finished, the monitor runs the *perf mem report* command to read memory events and record the results. To the best of our knowledge, this is the first time memory events have been used to detect cache side channel attacks.

Considering some programs that run permanently or have a long-running time, we assign a maximum running time in advance. If the execution time of the unknown program exceeds the maximum running time, the monitor stops the program. In this case, memory events can still be obtained.

The cache attack library gathers existing known attacks. As a proof of concept, we chose the data segment and code segment from the shared library, as shown in Table 1. The data segment takes the AES of the *libcrypto* library as an example. T-Tables of AES are vulnerable to cache side channel attacks, we store T-Tables's name into the cache attack library, as listed in the second row of Table 1. There is a cache side channel vulnerability of RSA in GnuPG. We store the name of function corresponding to sensitive memory addresses in the third row of Table 1. *CacheHawkeye* stores these names in an array. Only the attack library needs to be improved when a new attack is revealed.

The detector preprocesses memory event data from the monitor, analyzes it by the detection algorithm, and then makes a decision. The details of the detector will be described in Section 3.2.

Table 1. Cache attack library.

Cryptosystem	**Sensitive Memory Address**				
AES	Te0	Te1	Te2	Te3	Te4
RSA	mpih_sqr_n_basecase		mpihelp_divrem		mpihelp_mul_karatsuba_case

3.2. Detector Design

The monitor reads the memory events and stores results in a file named *Perf.data*. Detector analyzes the file and makes a judgment on the unknown program. The detector's detection mechanism is depicted in Figure 3. The *perf.data* file is initially preprocessed by the detector. The *perf.data* file contains information about memory loads and memory stores. The result of memory loads and memory stores is a table with N rows and 10 columns. Monitoring frequency determines the values of N. In general, the greater the frequency, the more samples are generated, and N is larger. In the preprocessing phase, the detector extracts all rows of the data symbol column of the two types of memory events to generate a sequence.

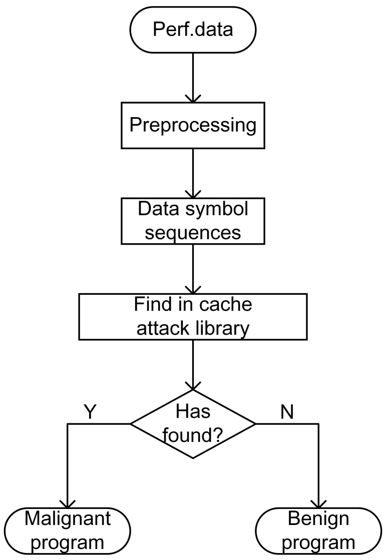

Figure 3. Detector execution flow.

For each data symbol in the monitored result sequence, the detector looks for the corresponding symbol name in the cache attack library. If a symbol in the sequence matches the cache attack library, it indicates that the program is a side channel attack program. If none of the data symbols in the sequence are in the cache attack library, the program is considered benign.

3.3. Improve the Detection Accuracy

In most attack scenarios, the attacker usually runs for a long time because he does not know when the victim executes. Because the longer the execution time, the more complete the sample results. For a long-running attack program, *CacheHawkeye* can easily judge the program as a malicious program. We believe that the detection accuracy of *CacheHawkeye* is positively related to the running time of the attacker. Considering the ideal attack scenario, in which both the attacker and the victim carry out and complete at the same time. Because the attacker's execution time is so short, *CacheHawkeye* has a hard time detecting this case correctly.

In order to identify these programs, we need to make some improvements to *CacheHawkeye*. When monitoring the unknown program, *CacheHawkeye* chooses a frequency parameter and assigns a large value to this parameter to collect more complete samples. Frequency adopts an odd value to avoid lockstep sampling. The time spent sampling may rise as the sampling frequency increases, We will compare different sampling frequencies and choose the best one in Section 4.2.

4. Evaluation

We used an MSI laptop with an Intel(R) Core(TM) i7-6700HQ hyper-threading quad-core processor in this phase of the experiment. The frequency of the processor is 2.60 GHz. The size of the last level cache is 6 MB. The operating system is Ubuntu 18.04LTS. The *perf* tool version is 5.4.114. We evaluated the performance of *CacheHawkeye* in this section.

4.1. Detect Flush+Reload and Flush+Flush Attacks

In this subsection, we evaluated the performance of *CacheHawkeye* on Flush+Reload and Flush+Flush attacks. Flush+Reload attack program monitored target addresses 10,000 times and Flush+Flush attack monitored target addresses 5600 times. As a control experiment, we also evaluated the behavior of *CacheHawkeye* on legit AES and RSA encryption and decryption programs. These legit programs also use shared libraries and may access sensitive functions (memory addresses) stored in the cache attack library, so we evaluated the performance of *CacheHawkeye* on these programs. We will analyze the performance of *CacheHawkeye* under various system loads in Section 4.3, and we temporarily overlook the interference of system load in this subsection.

The results of *CacheHawkeye* detected Flush+Reload attacking RSA are listed in Table 2. For the sake of brevity, we only list the data in user mode. The results of mem-loads are listed in columns 1 and 2, while the results of mem-stores are listed in columns 3 and 4. The data symbol columns of mem-loads and mem-stores both contain monitored functions (such as mpihelp_mul_karatsuba_case and mpihelp_divrem). Some data symbols appear repeatedly in the same column because the other parameters of this row are different. These functions are frequently accessed and account for 10% to 12.99% overhead. Because these function names are stored in the cache attack library, *CacheHawkeye* determines that this program is malicious.

Table 3 shows the results of *CacheHawkeye* detecting Flush+Flush attacking AES. We list a few lines which we care about. For the results of mem-loads, the data symbol in the first column does not contain any monitored addresses in the cache attack library. For the results of mem-stores, the data symbols in the third column are contained in the cache attack library. As a result, *CacheHawkeye* considers this program to be vicious.

Let us explain the above results. In Flush+Reload, Flush is memory storage procedure in which the cache line is evicted to memory, and Reload is a memory loading procedure in which the memory block is placed into cache. As a result, Mem-stores and mem-loads contain the names of the sensitive functions. Different from Flush+Reload, Flush+Flush only includes the Flush process, so the sensitive memory addresses are only found in mem-stores.

Table 2. Results of Flush+Reload attacking RSA.

Mem-Loads		**Mem-Stores**	
Data Symbol	**Overhead (%)**	**Data Symbol**	**Overhead (%)**
mpihelp_mul_karatsuba_case	12.99	0x00007fffef0e6938	20.00
mpihelp_mul_karatsuba_case	12.93	mpihelp_mul_karatsuba_case	10.00
mpihelp_mul_karatsuba_case	12.55		
mpihelp_divrem	11.48		
mpihelp_divrem	11.35		

Table 3. Results of Flush+Flush attacking AES.

Mem-Loads		**Mem-Stores**	
Data Symbol	**Overhead (%)**	**Data Symbol**	**Overhead (%)**
0x00007fbf41991acc	6.06	Te2+0x0	1.05
0x000056511b858641	1.56	Te2+0x3c0	1.05
0x00007fbf41d97f0a	1.45	Te0+0x3c0	1.05

We ran tests to evaluate the performance of *CacheHawkeye* on legit cryptographic programs that use shared libraries and monitored functions or memory addresses. We tested four programs: AES encryption, AES decryption, RSA encryption, and RSA decryption.

Table 4 lists all results of a legit AES encryption program. The data symbol column does not contain any sensitive functions or memory addresses, So *CacheHawkeye* believes that this program is legit. The results of the legit AES decryption program are similar to Table 4 and also do not contain any sensitive functions or memory addresses. These results are not presented for the sake of brevity.

It is worth noting that we divide the detection of the legit RSA decryption program into two tables. Table 5 shows the memory load results, and Table 6 shows the memory store results. There are no sensitive function names in Table 5. However, this does not mean that the legit RSA decryption program does not access these addresses, but because the access is not frequent enough, they are not caught by *CacheHawkeye*. There are some sensitive function addresses(such as mpih_sqr_n_basecase and mpihelp_divrem) in the symbol column of Table 5, this indicates that the legit RSA program has accessed these target addresses(functions). But there are no sensitive function addresses in the data symbol column. These demonstrate that the symbol column represents the legit program's memory access, whereas the data symbol represents the cache side channel attack's malicious memory access. The *perf* tool automatically puts the sensitive function name of the attack program in the data symbol column and puts the sensitive function name which the legit program also accesses in the symbol column. We conjecture that the reason behind it may be the memory access (by *clflush* instruction and *movl* instruction) of the attack program is somewhat different from the memory access of the legit program. *CacheHawkeye* only pays attention to the data symbol column, so it does not misjudge the legit cryptographic program.

For legit RSA encryption programs, the results of the data symbol column still do not contain any sensitive functions or memory addresses. These results are not presented for the sake of brevity. Therefore, *CacheHawkeye* can distinguish between benign cryptographic programs and side channel attack programs.

Table 4. Legit AES encryption program detection results.

Mem-Loads		Mem-Stores	
Data Symbol	**Overhead (%)**	**Data Symbol**	**Overhead (%)**
0xffff8afe01fd5d18	100.00	0xfffffe000006fde8	25.00
		0xffff8afdf1ab1a80	12.50
		0xffff8afcef77674c	12.50
		0xffffb8f545bf7cc0	12.50
		0xffffb8f545bf7d08	12.50
		0xfffffe000006fe28	12.50
		0xffffb8f545bf7c18	12.50

Table 5. Legit RSA decryption program detection results of Mem-loads.

Symbol	Data Symbol	Overhead (%)
vma_interval_tree_insert	0xffff8afd591d93f0	19.83
copy_page	0xffff8afd710e3da0	16.99
filemap_fault	0xffffde478cd0f048	15.43
filemap_map_pages	0xffffde478ef1ba88	15.43
filemap_map_pages	0xffffde478f2f75c8	14.97

Table 6. Legit RSA decryption program detection results of Mem-stores.

Symbol	Data Symbol	Overhead (%)
des_setkey.part.0	0x00007fffbc126589	6.25
mpih_sqr_n_basecase	0x00007f6b27fac7b8	6.25
mpihelp_addmul_1	0x00005638ece97148	6.25
mpihelp_divrem	0x00007fffbc1268e8	6.25
mpihelp_mul	0x00007f6b27fac628	6.25

4.2. Sampling Frequency Configuration

In this subsection, we evaluated the performance of *CacheHawkeye* at different frequencies and determine an appropriate sampling frequency. We tested *CacheHawkeye* to detect 4 representative attack programs and 4 legit cryptographic programs which may access sensitive addresses. we chose programs that are extremely difficult to detect when configuring the frequency, this can make the configured frequency more universally adaptable. For *CacheHawkeye*, the shorter the execution time of the attack program, the fewer sensitive memory address accesses, and the more difficult it is to detect. We chose 4 programs with very short execution time to configure the sampling frequency. As shown in Table 7, the execution time of these programs is only 7–12 ms. Real-world attacks must be much longer than these times because the attacker cannot synchronize with the victim. Therefore, the frequency configured according to these attack programs far meets the requirements of detecting real-world attacks.

We monitored 4 representative attack programs and 4 legit cryptographic programs with sampling frequencies of 2999, 5999, 8999, 11,999 and 14,999. Each attack program is executed 1000 times at each frequency. 4000 attacking samples are generated per frequency. *CacheHawkeye* also tests legit AES encryption/decryption programs and RSA encryption/decryption programs under different frequencies. Each legit program is executed 1000 times at each frequency. We use accuracy to evaluate the performance of *CacheHawkeye* at different frequencies and then determine the appropriate sampling frequency configuration. Accuracy refers to the percentage of samples that are judged correctly in the total samples. The formula for accuracy is as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

In Equation (1), True Positive(TP) represents that the malignant program is correctly recognized, True Negative(TN) represents that the benign program is correctly recognized, False Positive(FP) represents that the benign program is recognized as a malignant program, and False Negative(FN) represents that the malignant program is recognized as a benign program.

Table 7. Execution time of attack programs.

Attacks	**F+F**		**F+R**	
	AES	**RSA**	**AES**	**RSA**
Execution(ms)	10.6	7	12	10.4

The accuracy of malicious and legit programs at different sampling frequencies is shown in Figure 4. The accuracy of the *CacheHawkeye* is only 82.7% when the sampling frequency is 2999. *CacheHawkeye's* accuracy improves as the sampling frequency rises. The accuracy rate reaches 100% when the sampling frequency reaches 14,999.

We hypothesized that a greater sample frequency would lengthen the sampling time, so we measured it at various frequencies. We define sampling time as the time it takes to collect memory events and store them as a file. Figure 5 shows the average sample time of four malicious programs at various frequencies. We can see that when the frequency increases, the sample time does not change significantly. We only need to consider the accuracy when configuring the frequency. As a result, *CacheHawkeye's* sample frequency configuration is 14,999.

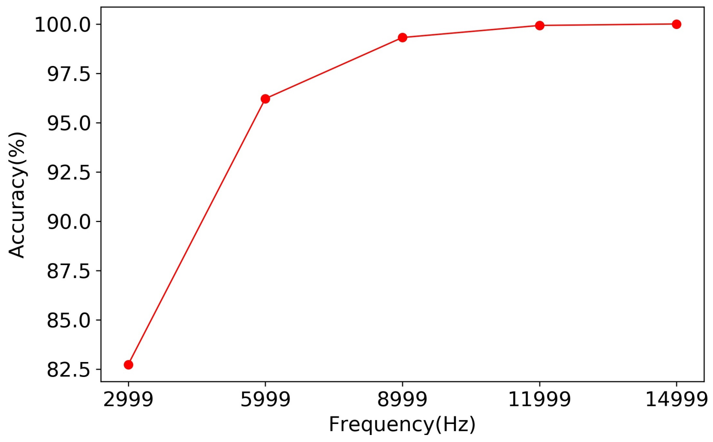

Figure 4. Accuracy of *CacheHawkeye*.

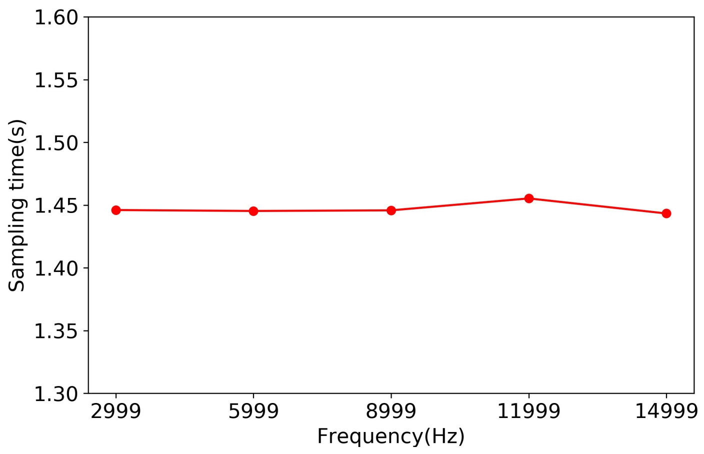

Figure 5. Sampling time at different frequencies.

4.3. Performance under Different System Loads

In this subsection, we evaluated *CacheHawkeye* under different system loads. We used *unixbench* and *sysbench* to generate system load. We used the default configuration of *unixbench*. The configuration settings of *sysbench* are listed in Table 8. During the execution of *sysbench*, we randomly picked one of the five routines. The system loads are divided into three categories: no-load, average-load, and full-load. No-load means that there is no system load when *CacheHawkeye* is running. The average-load has two workloads, one runs *sysbench*, the other runs *unixbench*. The full-load has four workloads, two of which run *sysbench*, and the other two run *unixbench*.

Table 8. Configuration settings of *sysbench*.

Test	Setting
cpu	cpu-max-prime = 2000
threads	num-threads = 500 thread-yields = 100 thread-locks = 4
fileio	num-threads = 16 file-total-size = 2G file-test-mode = rndrw
memory	memory-block-size = 8k memory-total-size = 1G
mutex	num-threads = 100 mutex-num = 1000 mutex-locks = 100,000 –mutex-loops = 10,000

We tested *CacheHawkeye* to detect 4 representative attack programs and 4 legit cryptographic programs which may access sensitive addresses under different system loads. Each program is executed 1000 times. 4000 benign samples and 4000 malignant samples are generated under each system load. The experimental results are listed in Table 9. We discovered that *CacheHawkeye* is 100% accurate under no-load and full-load, and 99.99% accurate under average-load. Because *CacheHawkeye* has not been pre-trained under different system loads, it can be expected that *CacheHawkeye* still performs excellently under unknown system loads. As a result, it can be inferred that the performance of *CacheHawkeye* performance is unaffected by system load. Because the memory capacity is substantially more than the capacity of the microarchitecture components (such as the branch instruction buffer and cache), memory events are very little affected by system loads.

Table 9. Detection results under different load conditions.

	No Load		**Average Load**		**Full Load**	
	Positive	**Negative**	**Positive**	**Negative**	**Positive**	**Negative**
True	4000	4000	3999	4000	4000	4000
False	0	0	1	0	0	0

Table 10 summarizes some limitations of the above work. CacheRadar and Alam et al.'s methods cannot detect Flush+Flush attacks. These two strategies, however, do not take system loads into account. We believe that these strategies are extremely sensitive to system loads because hardware events such as cache hits and misses are highly susceptible to interference from system loads. NIGHTs-WATCH has a good performance in known system loads and can detect Flush+Flush attacks. However, system loads still bring an accuracy loss of 4.97% [13] and this pre-trained model may perform poorly under unknown system load. Microarchitecture events are used as feature vectors for detection in all of the approaches listed above. Our approach detects cache side channel attacks using memory events. Compared with the above methods, our method has a very strong ability to adapt to the system loads and close to 100% accuracy.

Table 10. Related work.

Method	Flush+Flush	System Load	Hardware Events
CacheRadar	No	Sensitive	Branches, Cache Hits
Alam et al.	No	Sensitive	Branch Misses, LLC Accesses/Misses
NIGHTs-WATCH	Yes	Less Sensitive	LLC Misses, CPU Cycles
CacheHawkeye	**Yes**	**Neglect**	**Memory Events**

5. Discussion

CacheHawkeye has a shortcoming. When *CacheHawkeye* monitors the attacking program, the secret key is leaked when the victim is running. In order to overcome this shortcoming, we have two suggestions for using *CacheHawkeye*. The first suggestion is that the user should not execute the cryptographic program when *CacheHawkeye* is detecting unknown programs. If the user must execute a cryptographic program during the detection, the second suggestion is that the user can use Flush+Prefetch [25] approach to protect the key in real-time.

6. Conclusions

This paper designs and implements a system called *CacheHawkeye* to detect cache side channel attack programs. *CacheHawkeye* digs deeper into the semantics of cache side channel attacks and detects these attacks by memory events. Our evaluation shows the detection accuracy of *CacheHawkeye* is close to 100%. The detection accuracy of *CacheHawkeye* is hardly affected by any system loads. *CacheHawkeye* is a lightweight program that can be immediately deployed on existing software and hardware platforms.

Author Contributions: Conceptualization, H.Y.; methodology, H.Y.; software, H.Y.; validation, H.Y. and C.C.; formal analysis, H.Y.; investigation, H.Y.; resources, C.C.; data curation, H.Y.; writing—original draft preparation, H.Y.; writing—review and editing, C.C.; visualization, H.Y.; supervision, C.C.; project administration, C.C.; funding acquisition, C.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by National Key Research and Development Program of China(2018YFD0700302).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: Not applicable.

Acknowledgments: The authors want to thank the experts for dedicating time to a series of interviews leading to the completion of this paper and the whole, broader study.

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

References

1. Liu, F.; Yarom, Y.; Ge, Q.; Heiser, G.; Lee, R.B. Last-level cache side-channel attacks are practical. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 605–622.
2. Yarom, Y.; Falkner, K. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Proceedings of the 23rd {USENIX} Security Symposium ({USENIX} Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
3. Gruss, D.; Maurice, C.; Wagner, K.; Mangard, S. Flush+ Flush: A fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessmen*t; Springer: Berlin/Heidelberg, Germany, 2016; pp. 279–299.
4. Wang, Y.; Ferraiuolo, A.; Zhang, D.; Myers, A.C.; Suh, G.E. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In Proceedings of the 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 5–9 June 2016; pp. 1–6.
5. Zhou, Z.; Reiter, M.K.; Zhang, Y. A software approach to defeating side channels in last-level caches. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 871–882.

6. Oliverio, M.; Razavi, K.; Bos, H.; Giuffrida, C. Secure Page Fusion with VUsion: https://www.vusec.net/projects/VUsion. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; pp. 531–545.
7. Inci, M.S.; Gulmezoglu, B.; Irazoqui, G.; Eisenbarth, T.; Sunar, B. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 368–388.
8. Osvik, D.A.; Shamir, A.; Tromer, E. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at the RSA Conference*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 1–20.
9. Percival, C. Cache Missing for Fun and Profit. 2005. Available online: http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf (accessed on 6 January 2022).
10. Mushtaq, M.; Mukhtar, M.A.; Lapotre, V.; Bhatti, M.K.; Gogniat, G. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. *Inf. Syst.* **2020**, *92*, 101524.
11. Demme, J.; Maycock, M.; Schmitz, J.; Tang, A.; Waksman, A.; Sethumadhavan, S.; Stolfo, S. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Comput. Archit. News* **2013**, *41*, 559–570. [CrossRef]
12. Qader, Z.; Mo, A.; Gegov, A. A Comparison Study on Flush+Reload and Prime+Probe Attacks on AES Using Machine Learning Approaches. In *UK Workshop on Computational Intelligence*; Springer: Berlin/Heidelberg, Germany, 2017.
13. Mushtaq, M.; Akram, A.; Bhatti, M.K.; Chaudhry, M.; Lapotre, V.; Gogniat, G. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, Los Angeles, CA, USA, 2 June 2018; pp. 1–8.
14. Mushtaq, M.; Akram, A.; Bhatti, M.K.; Rais, R.N.B.; Lapotre, V.; Gogniat, G. Run-time detection of prime+ probe side-channel attack on AES encryption algorithm. In Proceedings of the 2018 Global Information Infrastructure and Networking Symposium (GIIS), Thessaloniki, Greece, 23–25 October 2018; pp. 1–5.
15. Payer, M. HexPADS: A platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 138–154.
16. Gruss, D.; Spreitzer, R.; Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In Proceedings of the 24th {USENIX} Security Symposium ({USENIX} Security 15), Washington, DC, USA, 12–14 August 2015; pp. 897–912.
17. Briongos, S.; Irazoqui, G.; Malagón, P.; Eisenbarth, T. Cacheshield: Detecting cache attacks through self-observation. In Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, Tempe, AZ, USA, 19–21 March 2018; pp. 224–235.
18. Bazm, M.M.; Sautereau, T.; Lacoste, M.; Sudholt, M.; Menaud, J.M. Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In Proceedings of the 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC), Barcelona, Spain, 23–26 April 2018; pp. 7–12.
19. Kulah, Y.; Dincer, B.; Yilmaz, C.; Savas, E. SpyDetector: An approach for detecting side-channel attacks at runtime. *Int. J. Inf. Secur.* **2019**, *18*, 393–422. [CrossRef]
20. Chiappetta, M.; Savas, E.; Yilmaz, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.* **2016**, *49*, 1162–1174. [CrossRef]
21. Alam, M.; Bhattacharya, S.; Mukhopadhyay, D.; Bhattacharya, S. Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks. *IACR Cryptol. ePrint Arch.* **2017**, *2017*, 564.
22. Zhang, T.; Zhang, Y.; Lee, R.B. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 118–140.
23. Tang, A.; Sethumadhavan, S.; Stolfo, S.J. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 109–129.
24. Wang, X.; Konstantinou, C.; Maniatakos, M.; Karri, R. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Barcelona, Spain, 23–26 April 2015; pp. 544–551.
25. Mukhtar, M.A.; Mushtaq, M.; Bhatti, M.K.; Lapotre, V.; Gogniat, G. Flush+ Prefetch: A countermeasure against access-driven cache-based side-channel attacks. *J. Syst. Archit.* **2020**, *104*, 101698. [CrossRef]