Ghost Thread: Effective User-Space Cache Side Channel Protection

Robert Brotzman Danfeng Zhang Mahmut Kandemir Gang Tan
Pennsylvania State University

ABSTRACT

Cache-based side channel attacks pose a serious threat to computer security. Numerous cache attacks have been demonstrated, highlighting the need for effective and efficient defense mechanisms to shield systems from this threat. In this paper, we propose a novel application-level protection mechanism, called Ghost Thread. Ghost Thread is a flexible library that allows a user to protect cache accesses to a requested sensitive region to mitigate cache-based side channel attacks. This is accomplished by injecting random cache accesses to the sensitive cache region by separate threads. Compared with prior work that injects noise in a modified OS and hardware, our novel approach is applicable to commodity OS and hardware. Compared with other user-space mitigation mechanisms, our novel approach does not require any special hardware support, and it only requires slight code changes in the protected application making it readily deployable. Evaluation results on an Apache server show that Ghost Thread provides both strong protection and negligible overhead on real-world applications where only a fragment requires protection. In the worst-case scenario where the entire application requires protection, Ghost Thread still incurs negligible overhead when a system is under utilized, and moderate overhead when a system is fully utilized.

KEYWORDS

side-channel; cache; mitigation

ACM Reference Format:
Robert Brotzman Danfeng Zhang Mahmut Kandemir Gang Tan. 2021. Ghost Thread: Effective User-Space Cache Side Channel Protection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422337.3447846

1 INTRODUCTION

Side channel attacks observe physical characteristics of a machine such as cache usage [3, 35], power consumption [24], and even electromagnetic field fluctuations [1] to uncover sensitive data. These attacks have plagued both hardware and software developers for years, particularly developers working with sensitive information.

Cache-based side channel attacks (cache attacks) [3, 35] have been well known for over a decade. They are particularly dangerous due to their capability of leaking hundreds of kilobytes of information in seconds [14]. Recently, cache attacks have made a splash in the computing world: Spectre [23] and Meltdown [27] exploits show that when combined with other attack vectors, cache attacks can be a serious threat to computer systems. Most alarmingly, cache attacks have been shown to be possible across virtual machines in a cloud environment [38] and secure enclaves [40]. The root cause of cache side channels is that some programs modify the cache state in ways that depend on confidential data used by the programs. Hence, an adversary can observe the cache usage during program execution and infer what data could have been used to cause the state changes of the cache.

Many defenses have been proposed to mitigate those attacks, with the goal of eliminating or reducing the effects of sensitive data on cache. However, a big challenge is compatibility: *it is difficult to protect vulnerable applications on commodity hardware/OSes.* Many mechanisms [7, 10, 22, 25, 29–31, 42, 45, 48, 51] require substantial changes to operating systems and/or hardware. Such mechanisms are appealing in the long term since vulnerable applications can be protected with few or no changes, but they are unlikely to be adopted in the near future. On the other hand, cryptographic libraries, common targets of cache attacks, typically use tricks (e.g., the scatter-gather technique) to thwart those attacks. However, those tricks require code-rewriting, and sometimes, a constant-time version requires significant changes compared to the most efficient ones. For example, bit-sliced AES implementations [21] rely on a circuit implementing the AES S-box, which diverges from table-based implementations significantly. These tricks are subtle to implement, and they are highly tailored for cryptographic algorithms; it is still challenging to protect other applications, or even a different implementation of a cryptographic algorithm.

Recent work by Gruss et al. [13] offers a promising application-level solution requiring few code changes. The mechanism utilizes hardware transactional memory (HTM) to ensure that all sensitive cache lines are "locked" into the cache when sensitive code blocks are executed. However, the approach relies on HTM, which is still absent in many processors (i.e. AMD processors). To avoid frequent transaction failures, the protected memory is limited to the size of CPU's caches; for example, the write set of HTM is limited to the size of the L1 cache (typically several KB) [13]. Moreover, the overhead on memory-intensive applications can go as high as 3.5X, due to more frequent transaction abortions.

In this paper, we propose a novel application-level defense mechanism called Ghost Thread to mitigate cache side channels. Compared with existing defense mechanisms, Ghost Thread works on *any commodity OS and hardware that supports concurrency.* Moreover, it protects vulnerable applications *without any code change except for a few library calls.* Ghost Thread serves as a flexible library that protects accesses to the sensitive memory region by injecting random accesses to the same region. By doing this, an adversary

CODASPY '21, April 26–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8143-7/21/04…$15.00
https://doi.org/10.1145/3422337.3447846

now has to distinguish random behavior added by Ghost Thread from actual program behavior to launch a successful attack. While the overall idea seems simple, the insight behind Ghost Thread is that the overhead of using a concurrent thread to randomly pollute the cache will have limited impact on the total execution time of the protected application, particularly when the machine is not under heavy load. Meanwhile, Ghost Thread offers strong security: *we prove that the number of samples needed to launch a successful attack grows exponentially with the number of injected accesses.* Moreover, Ghost Thread defends against various kinds of existing attacks, including attacks on both symmetric and asymmetric ciphers.

The main contributions of this work are as follows:

- We propose a novel, versatile, and easily deployable mechanism for mitigating cache attacks. It offers low overhead and strong security assurances in practice without any changes to existing hardware and OS. These goals are accomplished by randomly injecting phony cache accesses to a sensitive memory region, making it difficult for an adversary to distinguish real memory accesses from noise memory accesses.
- We present a security analysis that quantifies the security Ghost Thread can provide. This is performed by estimating the number of samples needed to launch a successful attack. We also present empirical security experiments showing that Ghost Thread can thwart known cache attacks against common crypto algorithms.
- We implement Ghost Thread as an application-level library and evaluate the runtime overhead of Ghost Thread under different system loads, different Ghost Thread configurations, and different tradeoffs between security and performance. We show that Ghost Thread has negligible overhead when a system is under utilized, or in real-world situations where not all code requires protection. In the worst-case scenario where a system is fully utilized and the entire application is protected, the overhead is still acceptable. Compared to state-of-the-art application-level mitigation, Ghost Thread improves performance by up to one order of magnitude in the worst case with respect to overhead.

2 BACKGROUND AND ATTACK MODEL

2.1 Cache Side Channels

Although different cache attacks work in different ways, they all try to reveal confidential information by probing the existence or absence of data in data/instruction cache. Prior work has shown that cache side channels reveal private keys in the implementation of commonly used cryptographic algorithms such as AES, RSA and ElGamal [3, 14, 15, 26, 35, 46, 47]. Attacks on them manifest because their cache access patterns depend on their secret keys. Next, we discuss two categories of cache attacks as well as techniques to reveal information from the cache.

2.1.1 Synchronous/Asynchronous Cache Attacks. In a *synchronous* cache attack, the attacker controls the beginning and end of the victim instruction being monitored [35]. Hence, the attacker obtains cache behavior that is associated with a particular piece of code in the victim process. Synchronous cache attacks were used in [14, 20, 35, 49] to demonstrate the feasibility of attack techniques such as Prime+Probe, Flush+Reload and Flush+Flush. However, due to the strong assumption that the attacker has a fine-grained control over the victim process, they are considered less practical. This is particularly true for attacks monitoring a few instructions of interest by inlining monitoring code into the victim program.

Synchronous	Window (cycles)
Flush + Flush	200+(victim access time) [14][1]
Flush + Reload	520 [15]
Prime + Probe	500+(victim access time) [50]
Asynchronous	Window (cycles)
Flush + Reload	2500 [46]
Prime + Probe	5000 [32]

Table 1: Window sizes used in various cache side-channel attacks. Note that the window size for successfully launching the synchronous attacks depends on the execution time of the victim code. When the data is unavailable in the paper, we simply write "victim access time" in the table.

In contrast, an *asynchronous* attack [17, 32, 35, 46] does not control the beginning and end of the victim process. Instead, they wait for the application to begin and make observations at fixed intervals of time, which we will call a *window* in this paper, throughout the execution of victim. Since asynchronous attacks assume less control over the victim programs, they are considered to be more realistic.

2.1.2 Attack techniques. The *Flush+Reload* technique, first described in [17] and its name popularized in [46], consists of three phases. First, the attacker identifies and flushes cache lines shared with the victim using the *clflush* instruction. Next, the attacker waits some amount of time to allow the victim to access the shared cache lines. Lastly, the attacker times how long it takes to reload the cache lines he previously flushed. A cache line being accessed by the victim will introduce a shorter latency in the reload step.

The *Flush+Flush* [14] technique is similar to *Flush+Reload*, except that the last phrase flushes the shared cache line again. Since the *clflush* instruction's execution time depends on whether the target data is cached, Flush+Flush constructs a cache side channel.

A key limitation of *Flush+Reload* and *Flush+Flush* is that they require a physical page in memory to be shared between the victim and adversary [29, 46]. Although this has plenty of use cases, the attack fails when no memory page is being shared.

The *Prime+Probe* technique [32, 35] works under broader circumstances: it does not assume shared physical page, making it the most practical technique among others. Prime+Probe is also performed in three steps. First, the attacker primes cache sets being monitored, typically by making sufficient accesses to fill a cache set with his data. The attacker then waits for the victim to run. Afterwards, the attacker accesses data filling the cache lines being monitored: a cache line being accessed by the victim evicts attacker's data in the cache, introducing a longer latency in the probe step.

2.1.3 Attack window. All popular attack techniques sketched above require getting the cache into a known state, wait for the victim to execute for some time, and then reveal victim's cache accesses. We call the time interval of performing one round of all those steps an *attack window*.

[1] This work does not report a specific window size. However, the attack requires two clflush instructions to be executed. We note that the execution time of a single clflush instruction varies on different CPUs. We use 200 since [14] reported that a single clflush instruction takes a minimum of 100 CPU cycles on various CPUs to complete.

Since the length of an attack window affects the number of noise cache accesses being injected, we summarize the window sizes reported in various cache attacks in Table 1. Typically, a synchronous attack has a smaller window size because the adversary is assumed to control the beginning and end of a few instructions of interest. For example, victim access times as low as 100 cycles has been reported [50]. But for more realistic settings where the attacker only controls the beginning and end of one encryption, one AES encryption itself takes around 320 cycles [15], making a window size of 500+ cycles, and asymmetric cipher such as RSA requires a much longer victim access time.

For the less restrictive asynchronous attacks, the window size is typically thousands of cycles. Especially, the most realistic attack technique Prime+Probe requires priming an entire cache set (typically, in LLC) instead of individual cache lines; this takes considerable time since modern LLCs are often 12-way to 24-way associative. Moreover, while the victim access time can be very small in theory, Liu et al. [32] observe that smaller window sizes result in higher error rates when recovering data, highlighting that having a minimal window size is not optimal for asynchronous attacks. Hence, for an asynchronous attack, we estimate the minimum attack window to be 2000+ cycles for attacking both symmetric and asymmetric ciphers.

In sum, while the design and implementation of Ghost Thread is independent of the window size, we use 2,000 cycles as a conservative estimation of the window size of a realistic cache attack (i.e., asynchronous attack) in this paper. Although synchronous attacks, which are more difficult to launch in practice, have smaller window size (around 500 cycles), Ghost Thread still offers sufficient security (we treat the window size as an unknown parameter when estimating the attack difficulty in Section 5).

2.2 Attack Model

We consider the scenario where an adversary and victim share the same physical hardware, a common scenario today due to the prevalence of cloud computing. Furthermore, the adversary is able to be co-scheduled on the shared hardware with the victim. The adversary is capable of setting the CPU cache into a known state (e.g., flushing the L3 cache) before the victim process runs and query the cache state after. Hence, the attacks sketched in Section 2.1 are feasible. We assume that the shared cache between adversary and victim process (e.g., LLC cache) is also shared by Ghost Thread, but Ghost Thread works on any level of cache. By default, we refer to LLC when a concrete context is needed.

3 RELATED WORK

Protecting data from cache attacks is a well-studied problem. We compare with the most relevant techniques in this section.

Application-level mitigation. The most related work to Ghost Thread is Cloak [13], an application-level defense mechanism that requires few code changes. To protect an application, Cloak first preloads sensitive memory locations (marked by a user) into the CPU cache and uses HTM to ensure that all accesses to the sensitive locations are cached; otherwise, a failure in HTM happens and the accesses are rolled back. Cloak is shown to have negligible performance overhead on some applications but saw overheads of up to 248% on applications that incur more frequent transaction abortions. Compared with Ghost Thread, Cloak is not applicable to processors without the HTM feature. Moreover, it can only protect a limited memory region since all memory used by a transaction must be cached. Furthermore, as discussed in [13], it may introduce new side channels revealed by transaction aborts.

To remove cache side channels, another approach is to rewrite the source code so that memory accesses are independent of confidential data. A common approach in crypto implementation is to use constant time implementations of specific algorithms [9]. However, these techniques are complicated to get right and introduce overheads; it is very challenging for a non-expert to rewrite the source code in a secure way. Compiler-aided transformation exists [5, 34], but those techniques only remove sensitive branches in the source code; they do not prevent sensitive cache-line accesses, in general, as we do in Ghost Thread.

Noise injection. Ghost Thread draws inspiration from approaches that use randomized memory accesses to thwart cache attacks. There are a few techniques that randomize the cache accesses, but to the best of our knowledge, Ghost Thread is the first approach that works on existing OSes and hardware, and offers strong security.

At the OS/VM level, one related work is Düppel [51], which injects random delays into the memory access time to obfuscate the cache state (i.e., a cache hit may incur a long latency by the injected delay from OS). In contrast, Ghost Thread injects random cache accesses at the application level, making it possible to protect an arbitrary program without modifying OS/VM. Moreover, Ghost Thread introduces almost no overhead for under-utilized systems and moderate overhead for system with heavy loads; Düppel [51] incurs overhead regardless of system load. KeyDrown [39] is a kernel mechanism that prevents timing attacks on key strokes by injecting fake key strokes. However, KeyDrown does not generally mitigate cache side channels, and requires OS modification.

At the application level, diversification [6] and obfuscation [37] also introduce randomness to confuse attackers. Diversification generates a replica that preserves the original program semantics but differ at the level of machine instructions; hence, by selecting a replica to execute at random, it alleviates side channels. Obfuscation confuses the attacker by pretending that additional program paths are executed than what really was. However, these techniques only add low entropy to side channels and they offer no formal security guarantee against cache attacks. Moreover, they incur considerable overheads: 1.75X for protecting AES in libgcrypt [6] and an average overhead of 21.8X the unprotected execution time [37].

At the hardware level, randomizing cache accesses has been explored in prior work. Some system randomizes the cache mapping [31] and others change the cache replacement policy to evict and fill in a randomly picked cache line [7, 30]. Efficient implementation of Oblivious RAM (ORAM) [12, 28], a technique that cryptographically obfuscates the memory access patterns, also typically requires modifying existing architectures. In comparison, Ghost Thread works with commodity hardware.

Isolating shared resources. Another approach of thwarting cache-based side channels is to isolate the victim from the attacker. Previous work has achieved this by soft isolation that reduces resource sharing by better scheduling [42], or hard isolation that partitions resources to disallow sharing between an attacker and a

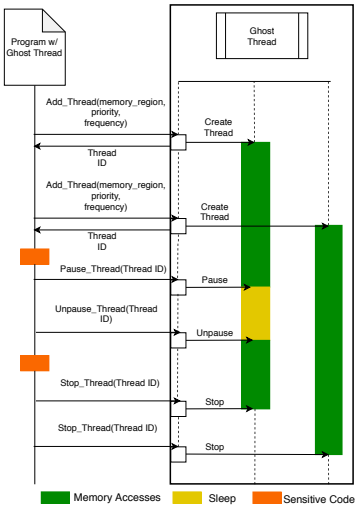

Figure 1: An overview of how Ghost Thread can be used to provide protection to an application.

victim [10, 22, 25, 29, 45, 48]. However, resource isolation is typically very challenging at the application level. Moreover, they all incur overhead even in a system with low resource utilization.

4 SYSTEM DESIGN

Ghost Thread works by inserting phony cache accesses to interfere with an adversary's ability to learn the actual cache accesses from the victim. With Ghost Thread, an adversary now has to determine if the status of the targeting cache line was caused by the legitimate behavior of the victim process or a noise access made by Ghost Thread. Randomly adding data to the CPU cache can significantly limit an adversary's ability to learn useful information. In this section, we discuss the design of Ghost Thread; we provide a formal analysis highlighting the added protection in Section 5.

4.1 Why User-Level Library

While injecting random memory accesses should be effective at thwarting cache attacks, where it is implemented impacts the amount of incurred performance overhead, as well as the compatibility of the approach. One way is to rewrite the victim code to inline those random memory accesses. However, this would incur a significant performance overhead. For instance, we observed over 60% overhead on execution time when we inlined 16 memory accesses in each round of encryption (160 total accesses) to protect accesses to the S-Box used by AES. One key insight of Ghost Thread is that we can take advantage of multiple cores in modern systems to inject phony cache accesses in a more efficient way. This results in virtually no overhead on systems that are not being fully utilized.

Another potential software-level solution we considered was to add noise using an operating system, in a similar manner as [51]. While this option may have resulted in more consistent results in some cases thanks to more control over process scheduling, it requires substantial modification of an OS. Ghost Thread is instead designed as a user-space library, allowing it to be easily integrated into an existing application on an unmodified OS and hardware.

```
tid=GT_add(&sbox,256,sizeof(int));
plaintext = AES_decrypt(ciphertext,key);
GT_pause(tid);
process(plaintext);
GT_unpause(tid);
ciphertext = AES_encrypt(plaintext,key);
GT_stop(tid)
```

Figure 2: An example showing how a user can modify an application to provide protection. The highlighted lines of code would be added by the user. Green indicates necessary code modification, yellow indicates code added to reduce overhead.

4.2 System Overview

Figure 1 illustrates how a user interacts with Ghost Thread to protect a vulnerable application. In the first step, the user identifies what memory regions in the application need protection against cache attacks. These are called *sensitive memory regions*; if an attacker can directly observe what locations in these regions are accessed by the application, secret information can be inferred. In the second step, the user modifies the application's source code with calls to the Ghost Thread library. Typically, she would insert calls to create protection threads before code regions that access sensitive data; these code regions are shown in orange in Figure 1[1]. Depending on the security requirements of the application, there can be as many protection threads as needed; they can protect the same memory region or each can protect a separate memory region. For each protection thread, the user can decide to keep it running (shown in green), pause it (shown in yellow), unpause it, or stop it.

To provide a more concrete example of how Ghost Thread can be used, Figure 2 shows a typical use case using AES encryption as an example. In the example, the lines in green indicate necessary code changes to get protection from Ghost Thread. The lines in yellow indicate optional features that reduce Ghost Thread impact on the application's performance. This program starts a protection thread to protect an encryption routine's S-Box (Substitution box). This is performed by providing the base address of the S-Box and the size.

To facilitate better performance, the thread is paused after the calling of `AES_decrypt`. This frees the system's processor that was running the protection thread. After performing some work on the plaintext and prior to performing the next AES encryption, the protection is unpaused. After the application is finished using AES routines, the protection thread can be stopped to free all resources associated with that thread.

4.3 Library Interface

We first introduce the interface of Ghost Thread and its basic functionality. We then describe advanced features in Section 4.4.

Creating a Ghost Thread. In order to create a protection thread, a user should call the following function provided by Ghost Thread:

```
u32 GT_add(void *base_addr,u32 num_elmts,
           u32 element_size,
           bool priority, u32 freq)
```

[1]We show in Section 4.5 how program locations and memory regions can be automatically identified to help users accurately use Ghost Thread's interface.

This function takes five parameters and returns a thread ID used internally to manage threads by Ghost Thread. The first three determine the sensitive memory region to be protected. Ghost Thread uses the base address of the memory region, the number of elements, and the elements' size to determine the sensitive memory region. The other two parameters (priority and freq) will be discussed shortly. An example use of `GT_add` is shown in Figure 2. By default, the frequency will be set to max ensuring optimal security. The priority of the thread is set to the maximum available to the application; for example if run as the root user on Linux the thread would have real-time priority by default.

Each protection thread in Ghost Thread is given a contiguous range of memory locations to protect. This design decision facilitates fast injection of noise memory accesses as it benefits from hardware prefetchers, which bring in more than one protected cache lines for each injected noise access.

Pausing and Unpausing Protection. It is often the case that the majority of an application does not need protection against side-channel attacks. As shown in Figure 2, there is likely going to be time when protection is unnecessary. To accommodate this, Ghost Thread can be paused and unpaused throughout an application's execution. Ghost Thread offers the following two functions to pause and unpause ghost threads:

```
GT_pause(u32 tid)
GT_unpause(u32 tid)
```

Each function takes the thread ID of the protection thread to be paused/unpaused. Semantically, pausing a protection thread puts it to sleep. This allows the OS to schedule other tasks without having to schedule the protection thread. With the function call, Ghost Thread looks up the thread internally and puts the desired thread to sleep, as illustrated in Figure 2 on line 3. Similarly, the unpausing function wakes up the thread with thread ID `tid` and resumes injecting noise.

In order to ensure that noise is currently being injected by a protection thread, the unpausing function also checks if the protection thread has completed all of its initialization and is actively adding noise to the cache. Doing so ensures that sensitive memory accesses after the unpausing function are properly protected by Ghost Thread.

Stopping Protection. When there is no longer a need to protect a specific memory region, a user should call the following function provided by Ghost Thread:

```
GT_stop(u32 tid)
```

This function allows the thread to terminate and return its resources to the OS. An example can be found on line 7 of Figure 2 after the program has completed all of its accesses to the S-Box.

To summarize, Ghost Thread's main functionality is composed of 4 methods: GT_add, GT_pause, GT_unpause, and GT_stop. To reap the protection benefits of Ghost Thread, a user only needs to add *one* line of code to their existing application for each region that needs protection. The remaining features are optional, but can significantly enhance application performance by freeing system resources.

4.4 Tuning Ghost Thread

For advanced developers, Ghost Thread provides many features that allow them to tune the protection for desired levels of security and performance. The remainder of this section will highlight these features and explain how and why they should be used.

4.4.1 Multiple Protection Threads. For some applications, a single protection thread may not sufficiently address their security requirements. For instance, when an application has a large sensitive memory region, a single thread may not be able to inject sufficient noise memory accesses. As a second example, an application may have multiple non-contiguous sensitive memory regions. In both situations, Ghost Thread allows users to create multiple protection threads, which inject noise accesses independently and concurrently.

Multiple protection threads are simply created by multiple calls to the `GT_add` function. To protect multiple non-contiguous memory regions, multiple protection threads should be created with non-overlapping memory regions. Additional threads with the same memory regions can also be created to increase the injection rate of noise; this provides additional security if required, as shown in Section 6.

4.4.2 Ensuring Thread Scheduling. Ghost Thread is implemented as a user-space library without any modification to the OS including its scheduler. Modern OSes use schedulers that attempt to fairly allocate CPU time to all processes waiting to run. This means the threads that Ghost Thread uses to protect an application may not be scheduled when the protected application is running. The situation becomes more likely when a system is more utilized since the scheduler becomes more relied upon to allocate CPU time.

To overcome this issue, Ghost Thread provides an option to set the priority of ghost threads. This is the fourth parameter, `priority`, of the `GT_add` function. By default, Ghost Thread sets the priority of a protection thread to be real time. In this case, the priority parameter to `GT_add` is set to `true`. Line 1 of Figure 2 shows how the priority of a protection thread is set to the default value of `true`. When a real-time protection thread is created, Ghost Thread requests the OS to create a thread with corresponding scheduling priority.

Most modern OSes support some notion of real-time scheduling. On Linux, real-time tasks can be preempted only when a task with higher priority is ready to run. By giving a ghost thread the maximum priority (99 in Linux), no other tasks can preempt it. By further setting the CPU core affinity of the thread to a specific core, Ghost Thread prevents the Linux scheduler from migrating the real-time thread to another core. After these steps, protection threads using real-time priority are scheduled on specific cores until they voluntarily give up their CPU time. This guarantees that ghost threads are running at the same time as the application they need to protect.

We empirically verified that real-time protection threads are always scheduled. We used the Linux Performance (Perf) Tools to monitor what tasks were scheduled on each core. Perf Tools sample what is running on every processor at fixed units of time. In our experiments, we set the priority of a thread to be real time and checked the result of the Perf Tools. We found that once a real-time thread is scheduled, it was never descheduled until it terminated.

Optionally, a ghost thread can be set to regular priority by setting the priority parameter of `GT_add` to `false`. Although changing the priority of a thread to be regular priority weakens the protection, it could still be desirable in the scenarios where the root privilege

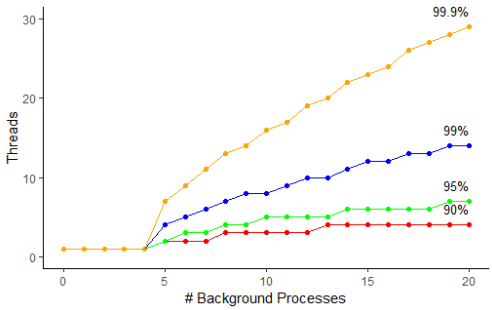

Figure 3: The number of threads required to obtain 99.9%, 99%, 95%, or 90% overlap with a protected thread, when various background tasks are running at the same time on a 6 core machine. These values assume that the scheduler assigns every task with the same probability of being scheduled.

is unavailable, or when the user is willing to sacrifice security for better performance. Moreover, adding multiple protection threads with regular priority can potentially offer similar level of protection as one real-time ghost thread.

Assuming a fair scheduler that schedules every thread equally, we can compute how likely at least one of the protection threads runs concurrently with an application that has just one thread. Figure 3 shows how many threads are required to obtain either 99.9%, 99%, 95%, or 90% overlap with the application thread, when the system is under various amounts of strain on a machine with 6 cores. The results show that on a system with a light load, just one protection thread is typically sufficient, even if the ghost thread is not real-time. As more background tasks are competing for the processors, the number of protection threads needed for a target overlap increases. The higher the percentage of overlap, the faster the number of threads required increases. But still, to offer 90% overlap, only a few regular priority threads are needed with heavy system load. We show in Section 6 how we can use these percentages to compute more accurately the protection added by Ghost Thread.

In sum, ensuring at least one protection thread is actively running when the protected application is running is a critical aspect of Ghost Thread. Ghost Thread offers two possible solutions to this problem. The ideal (and default) solution is to assign the real-time priority to the protection threads to ensure they are always running. Since this may not always be possible due to requiring root privileges, we also provide some guidance for users to obtain a target percentage of overlap by introducing multiple protection threads to ensure at least one of them is running when the application is running.

4.4.3 Varying Noise Injection Rate. By default, a protection thread injects memory accesses as quickly as possible in a tight loop. To better balance security and performance, Ghost Thread allows a user to configure the rate at which noise is injected.

Ghost Thread provides an option of setting the noise injection rate: the fifth parameter `freq` of the `GT_add` function. An example is shown at line 1 of Figure 2, where `freq` is set to the default value 0, specifying that the Ghost Thread injects noise memory accesses at the maximum rate; the maximum frequency we observed on our machine is about 470 million noise accesses per second. For other target frequency numbers, Ghost Thread calculates appropriate n (the length of a sequence of noise memory accesses in an iteration) and t (the length of sleep). In order to ensure consistent noise, our implementation always sets n to be a constant (usually 1,000,000). Based on that gap, Ghost Thread computes the amount of sleep (t) per iteration needed in order to achieve the desired frequency. With computed n and t, a ghost thread (1) injects a sequence of n noise memory accesses, and (2) puts the thread to sleep for time t.

	Precomputed	rand	PCG	RDRAND
Noise/Window	467	32	124	5

Table 2: The average number of memory accesses that a ghost thread can inject in a window of 2,000 clock cycles.

4.4.4 Memory Access Randomization. An important goal of Ghost Thread is that its protection threads inject noise accesses randomly to sensitive memory regions. Currently, Ghost Thread supports four different methods for obtaining randomness. Each of these methods can be selected by passing compiler options when building the Ghost Thread library.

The first method is to precompute the random numbers before the Ghost Thread's protection threads begin injecting noise. This can either be performed at run time during thread initialization, using one of our supported random number generators, or added by the user using their preferred random number generator. We also allow these numbers to be added at compile time. Adding the numbers at compile time can be significantly faster since there would be no overhead incurred during the execution of the application.

The three other supported methods generate random numbers as needed using different random number generators. Generating random numbers along with noise injection can be more versatile since it can be done on demand and takes up significantly less memory compared to precomputing. This versatility comes at a cost to security, since it takes longer to generate random numbers compared to looking up from a precomputed random-number table. Table 2 shows the security impact of random number generation techniques; for each method, it shows the maximum number of memory accesses per window, where a window is 2,000 CPU cycles.

In sum, we suggest using a secure random number generator to create a table prior to using Ghost Thread (the default setting of Ghost Thread). This provides the best security and performance. With a strong attacker model where the adversary may observe the cache status in every 2,000 clock cycles (i.e., when the window size is about 2,000 cycles), the security of other random number generators could be insufficient due to the few number of noise memory accesses being injected.

4.5 Usability of Ghost Thread

While the default configurations offer the best security of Ghost Thread, the advanced configurations such as priority, noise frequency and randomness mechanism are designed for more capable users to tailor Ghost Thread to their needs.

To better facilitate a non-expert user to adopt Ghost Thread, Ghost Thread uses a static taint analysis to mark the sensitive memory regions as well as insert pausing and unpausing statements.[2]

[2]In the future, we might also leverage existing static analysis tools [4, 8, 9, 43, 44] to do so in a more precise way.

The taint analysis only requires users mark the sensitive data in their application; it then automatically identifies two vulnerable code patterns: tainted branches and tainted memory accesses, following the constant-time programming principle [2].

When those patterns are detected, the taint analysis reports the corresponding source code line numbers, which directly lead to the insertion of pausing and unpausing statements. Tainted memory accesses also directly lead to memory regions to be protected. The only subtle case is tainted branches, which to be sound, require any memory region being accessed under the branches to be protected. Ghost Thread currently requires manual inspection to identify memory regions under tainted branches to be protected; in our experience, such manually work is typically straightforward.

We use an implementation of AES from libgcrypt 1.8.3 as a concrete example. With private key marked as tainted, the taint analysis marks all tainted memory accesses to the S-box; the marked code corresponds to the following pseudo code:

$ciphertext = s\_box[key \oplus plaintext]$;

Thus, the taint analysis automatically identifies both the code region (the encryption routine that performs the sensitive accesses) and memory regions (the S-box table) to be protected.

5 SECURITY ANALYSIS

Injecting noise typically does not eliminate a cache side channel. In this section, we show both in theory and practice that Ghost Thread effectively makes realistic side channel attacks impractical. We break our analysis up into two components: (1) How much harder does Ghost Thread make the adversary's task? (2) Does Ghost Thread thwart existing side-channel attacks?

5.1 Normalized Samples

We measure the attack difficulty with Ghost Thread by how many more samples are needed to successfully launch a cache attack on a vulnerable program protected by Ghost Thread. To accomplish this, we measure *normalized samples*, the ratio of samples needed to achieve the same level of confidence of retrieving the secret correctly.

Existing attacks typically observe the *existence* of accesses to sensitive memory regions to reveal private keys. This is done via exploiting the access time of a cache line being probed: a cache line being accessed in a window likely introduces a smaller latency compared with the case where it is absent. Intuitively, Ghost Thread works by shifting the access time when an access is absent to the case where it is accessed in a window. In this section, we follow an analytical model used in prior work [30, 33, 41] to quantify the number of samples needed to launch a successful attack.

We first abstract the execution time under the condition of *at least one access* and *no access* in a window as μ_1 and μ_2 respectively:

$$\mu_1 = P_1\ t_{hit} + (1 - P_1)\ t_{miss}$$
$$\mu_2 = P_2\ t_{hit} + (1 - P_2)\ t_{miss}$$

where t_{hit} (resp. t_{miss}) is the (expected) probing times when the data of interest is present (resp. absent) in the cache; P_1 (resp. P_2) is the probability that data remains in the cache at the probing time when the cache line is accessed (resp. absent) in a window. Following prior analytical models [33, 41], the number of measurements required for a successful attack (i.e., one that distinguishes samples from either μ_1 or μ_2 with high confidence) can be estimated as

$$N \approx \frac{2Z_\alpha^2\ \sigma^2}{(P_1 - P_2)^2 (t_{miss} - t_{hit})^2}$$

where Z_α is the quantile of the standard normal distribution for α, the desired success rate, and σ is the variance of probing time.

M \ δ	100	200	300	400	467
AES (M=16)	4.03E5	1.63E11	6.56E16	2.64E22	1.51E26
RSA (M=64)	2.33E1	5.44E2	1.27E4	2.96E5	2.44E6

Table 3: Normalized samples with various amounts of noise accesses for AES and RSA.

Samples needed without Ghost Thread. Due to background noise in a window, there is a small chance that an accessed cache line is evicted or an unaccessed one is cached. To simplify the model, we ignore such noise and estimate P_1 as 1 and P_2 as 0. Note that this is justified since it makes our estimation more conservative (i.e., it is in favor of the attack). Hence, we have $N_{w/o} = \frac{2Z_\alpha^2\ \sigma^2}{(t_{miss} - t_{hit})^2}$.

Samples needed with Ghost Thread. With Ghost Thread that injects δ random accesses in one window, P_1 remains 1, but P_2 significantly changes compared with the previous case: there is a $1 - (\frac{M-1}{M})^\delta$ chance that the cache line is brought into the cache even though it is absent from the original program. Hence, we have: $N_{w/} = \frac{2Z_\alpha^2\ \sigma^2}{(\frac{M-1}{M})^{2\delta} (t_{miss} - t_{hit})^2}$.

Normalized Samples needed with Ghost Thread. With Ghost Thread, it is more meaningful to measure $\frac{N_{w/}}{N_{w/o}}$ for the added security to the protected program. We call this ratio *Normalized Samples*:

$$\frac{N_{w/}}{N_{w/o}} \approx (\frac{M}{M-1})^{2\delta}$$

Figure 3 shows the normalized samples needed for table-based implementation of AES (with 16 sensitive cache lines) and RSA (with 64 sensitive cache lines). We emphasize that the noise added by Ghost Thread significantly improves the security of the vulnerable programs: the number of samples needed for a successful attack grows exponentially. In a conservative setting of windows with a size of 2,000 cycles for asynchronous attacks, Ghost Thread injects 467 noise cache accesses (Table 2). Hence, both attacks quickly become infeasible: over 100,000X more samples are needed to launch a successful attack. Even for a synchronous attack on AES with a small window size of 520 cycles [15], Ghost Thread can inject over 100 noise cache accesses, requiring over 100,000X more samples to launch the attack on AES. This is for just a single Ghost Thread active, multiple threads can be used to increase security even further.

Security against cross-processor attacks. We note that Ghost Thread requires that the cache being observed by the adversary be shared by the protected application and the protection threads spawned by Ghost Thread for this analysis to be applicable. In the common scenario when the LLC is being observed, the protection thread and protected application simply need to be running on the same processor since the LLC is usually shared among all cores. For

```
count = [0 for x in range(256)]
for i in range(numPlaintexts):
 plainTexts[i] = genRandPlaintext()
 usedCachelines[i] =
        observeEncrypt(key,plainTexts[i])
for keyByteGuess in range(256):
 for i in range(numPlaintexts):
  if guessCacheLine(keyByteGuess,plainTexts[i])
        in usedCachelines[i]:
   count[keyByteGuess]++
return maxIdx(count)

def guessCacheLine(guess,plaintext):
 return (guess ^ plaintext) / 16
```

Figure 4: Pseudo code of the AES attack.

cross-processor attacks, such as those shown in [19], we can apply the same analysis described in this section when either the processor interconnects all have the same memory access latency, or that Ghost Thread's protection thread is always scheduled on the same processor as the protected application. This is the case because cross-processor attacks apply the same principles of an intra-processor attack, but also leverage the high-speed processor interconnects between processors to detect the memory access time differences between cache hits and misses.

5.2 Empirical Study

Previously, we have shown the theoretical attack difficulty with Ghost Thread. Next, we use Ghost Thread to protect a vulnerable cryptographic library against real cache attacks. We show that Ghost Thread completely prevents those attacks even in the face of strong attack models. The attack, following [35], is a synchronous attack targeting the first round of AES in two commonly used implementations (mbedTLS and libgcrypt).

5.2.1 AES Attack. We follow the attack that targets the first round of AES in [35]. The pseudo code of the attack for revealing one key byte is shown in Figure 4. Note that this is a synchronous attack where the attacker is able to collect the cache state right after the first round of encryption; doing so makes the attack feasible even if only a very small number of samples are collected. Nevertheless, we show that Ghost Thread successfully thwarts such attacks.

In this attack, the attacker first encrypts a number of known plaintexts (`numPlaintexts`) and collects the resulting cache states right after the first round (`usedCacheLines`). After that, the attacker enumerates over 256 possible values of a certain key byte, and sees which values best "matches" the resulting cache states.

To do that, recall that the index into the S-Box used by AES in the first round is $p_i \oplus k_i$ where p_i is the i^{th} byte of plaintext and k_i is the i^{th} byte of the key. Hence, for each sample i, the attacker can successfully compute the index by using the known plaintext `plainTexts[i]` and the guessed key value. Hence, the key value that resulted in the most correct cache line guesses (stored in array `count`) is the key byte used in the encryption.

We performed this attack on three different machines running different Intel chips (i3, i5, and i7). For brevity, we present the results on the Intel i7 chip since all 3 experiments are very similar. The result for the unprotected program is shown at the top of Figure 5. Here, each blue dot corresponds to the percentage of correct cache line guesses for the corresponding guessed key byte (the x-axis);

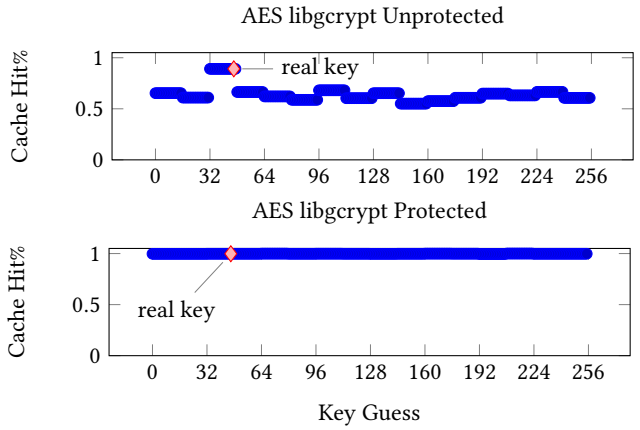

Figure 5: AES Attack without and with Ghost Thread. We see that without Ghost Thread, there is a strong correlation between key values and percentages of cache hits, while with Ghost Thread, virtually every key guess results in a cache hit, disrupting the correlation.

the red triangle identifies the actual value of the key byte used in these experiments (which is set to be 46). Without Ghost Thread, we observe that the cache line containing the real key value is easily distinguishable from other cache lines.[3]

In order to see how effective Ghost Thread is, we ran the attack again with the AES program protected. The results are shown at the bottom of Figure 5. With Ghost Thread, the real value of the key byte can no longer be identified. This is due to the fact that Ghost Thread injects hundreds of extra memory accesses, making all cache lines being cached in the experiments.

6 EVALUATION

In this section, we answer the following questions: (1) How much overhead does Ghost Thread have on a protected application? (2) What is the maximum overhead an application can incur when being protected by Ghost Thread? (3) What are the normalized samples needed for a successful attack after applying Ghost Thread (with various settings) to an application? (4) What is the impact of using Ghost Thread on other applications running on the machine?

To evaluate Ghost Thread, we use a machine with 32 GB of RAM on an Intel Core i7-5820K CPU at 3.3 GHZ using Ubuntu 14.04. For consistent results, we disabled hyper-threading and turbo boost.

6.1 Application Overhead

A key aspect of our evaluation is understanding how much overhead Ghost Thread adds to applications. To explore this, we applied Ghost Thread to an Apache web server.

Setup. In our experiments, we analyze the throughput (i.e. the number of requests per second) of an Apache web server with various numbers of concurrent users. We use an HTTP server benchmarking tool called Siege [11] to load a 2KB static web page using HTTPS for differing numbers of concurrent users; each configuration is repeated 100 times to reduce noise. We follow the

[3]The attack targeting the first round of AES cannot distinguish the exact key value since 16 key values are stored on the same cache line. However, this attack is much easier to launch compared to the complete attack in [35]; thwarting this version also prevents real attacks with more system noise and more bits to learn.

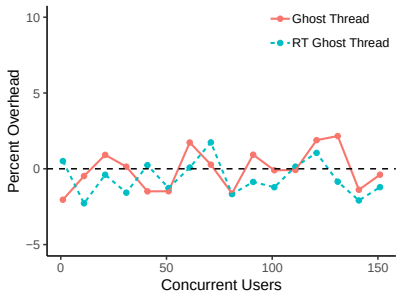

Figure 6: The throughput overheads of concurrent users accessing a webpage hosted on an Apache server with both the AES and RSA routines protected by Ghost Thread.

	Best Case	Worst Case
Cloak [13]	-0.8%	248%
Ghost Thread	-1.25%	20.1%

Table 4: Overheads of Cloak [13] and one single Ghost Thread.

experimental setup in [16]. There are two security-sensitive components of the Apache web server: an RSA routine is invoked to establish connections, and an AES routine is used to encrypt the transmitted data.

We apply Ghost Thread to protect the T-table implementation of the AES routine, requiring about 4KB of memory to be protected. Ghost Thread is also applied to the precomputed table (1KB) used by the RSA routine, which was shown to be vulnerable to cache attacks [47]. We use the taint analysis described in Section 4.5 to determine appropriate points to add the instrumentation.

Results. We measured the throughput overheads with various concurrent users ranging from 1 to 150 where each user requests a webpage of size 3KB. The results are shown in Figure 6 with two possible configurations: using real-time ghost threads, or normal-priority threads. The results demonstrate that in either case, Ghost Thread adds little to no overhead to the Apache web server regardless of the priority Ghost Thread used. This is because Ghost Thread only needs to be active during critical sections of the application where secrete data is being processed.

When Apache serves webpages, it spends 0.19% of the execution time on AES encryption, 0.23% on RSA encryption, for a combination of 0.42% of execution time that requires protection. This is consistent with our belief that programs used in practice likely require Ghost Thread to be active for small portions of their execution. In Section 6.2, we zoom in on the overhead incurred by applying Ghost Thread to the encryption routines, which provides insight regarding how Ghost Thread would perform on a smaller micro benchmark where it is active most of the time .

Comparison with Cloak [13]. A comparison between state-of-the-art user-space mitigation mechanism Cloak [13] and a single non-real-time Ghost Thread is shown in Table 4. Since our system does not support TSX, we use the results reported in [13] for comparison. Although the experimental environments are not completely identical, we note that both best cases reflect light system workloads. Compared to Cloak [13], Ghost Thread preforms comparably. Cloak reports a 0.8% speed up when protecting AES from

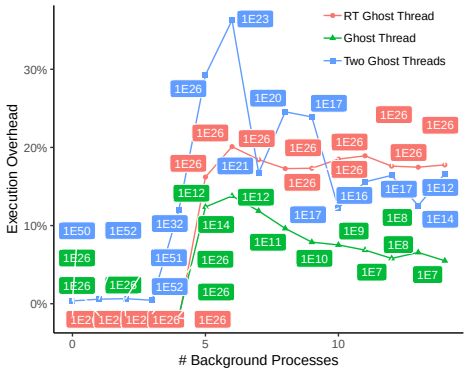

Figure 7: The latency overheads of Ghost Thread on an AES implementation from libgcrypt 1.8.3 with various configurations. The boxes show the additional normalized samples needed.

side channel attacks due to a higher cache hit rate and few memory accesses. We observe the same in Figure 6: Ghost Thread also has near zero overhead due to a higher cache hit rate and limited required resources of Ghost Thread since the region that requires protection in Apache is not large.

6.2 Worst Case Analysis

To protect a vulnerable application, Ghost Thread typically does not need to be active throughout the entire execution of an application. However, it is also useful to measure the maximum overhead an application can incur: both to understand the limit of Ghost Thread as well as how parameters such as thread priority and noise rate affect performance and security. Next, we conducted experiments on AES and RSA where Ghost Thread is active 100% of the time.

Setup. We ran AES using an implementation in libgcrypt 1.8.3 with 128-bit keys protecting the S-boxes (1 KB). We also applied Ghost Thread to the precomputed tables (4 KB) in the modular exponentiation routine of GnuPG 1.4.23 used by the RSA algorithm. We again used the taint analysis presented in Section 4.5 to identify where to insert Ghost Thread instrumentation. We used the corresponding AES and RSA implementations without Ghost Thread protection as the baseline. To get consistent results, we recorded the execution time of encrypting 5 million messages.

To fairly evaluate Ghost Thread under various system loads, we considered the percent increase in execution time of an AES application with and without Ghost Thread while other background processes are competing for resources. For the latter, we used two programs from SPEC CPU 2006: `bzip2` and `gobmk`. We selected these two programs since `bzip2` represents a memory-intensive program and `gobmk` represents a CPU-intensive program. Another scenario we used in our evaluation is when there is a mix of different kinds of programs on the system. To do this, we ran all of the integer benchmarks from SPEC CPU 2006 written in C.

The various background applications resulted in similar results, within measurement error. Thus we show only the results using `bzip2` as background programs hereafter. We presume this is the case since Ghost Thread only protects a small region of memory: the background application is unlikely to change the victim program's cache hit rate significantly.

6.2.1 Performance Results. The latency overheads added by Ghost Thread to AES with various system loads and Ghost Thread configurations are shown in Figure 7. We also performed this experiment on RSA and obtained similar overhead to what is found for AES, but omitted the details of these performance results due to space constraints. The results indicate that adding memory accesses to protect against cache side channels does not add noticeable overheads when a system is being under utilized (i.e., having fewer tasks than cores.).

We note that when Ghost Thread is active 100% of the time and all cores are occupied, the performance overhead is non-negligible both for AES and RSA. This is expected given that Ghost Thread competes for computation resources. However, we note that the *worst-case* overhead ranges between 13% to 35%, depending on the configuration of Ghost Thread. Moreover, as we show in Section 6.1, the overhead could be much smaller in more common scenarios where only part of an application requires protection.

Figure 7 further shows the impact of different configurations of Ghost Thread: a normal-priority protection thread, a real-time priority protection thread, and two normal-priority protection threads. All protection threads inject noise to the memory region that holds the S-Box for AES or the precomputed tables used in modular exponentiation for RSA.

When the priority of the thread is set to real time, the execution time is similar to a thread with normal priority until the system becomes fully utilized. This is due to the fact that the OS is unable to schedule anything on the core the real time thread gets assigned to. This causes a larger amount of overhead (about 20%) compared to the case of using a normal-priority thread (about 10%). We also observe that creating two normal-priority threads incurs the largest latency overhead in both cases.

Comparison with Cloak [13]. While the overhead of Ghost Thread is non-negligible in the worst-case scenario, we note that it is still reasonable compared with the state-of-the-art of user-level defence mechanism: Cloak was shown to have a worst case overhead of 248% on average in a heavily loaded system [13]. As shown in Table 4, the worst-case overhead of Ghost Thread (20.1%) is still reasonable and in fact, an order of magnitude better.

6.2.2 Security Results. While Ghost Thread is able to obtain negligible (when the system is under loaded) to modest (when the system is fully loaded) runtime overheads, it provides strong security for all system loads. We find that Ghost Thread is able to make side-channel attacks infeasible to launch.

We use normalized samples presented in Section 5 as a metric for measuring security. During experiments, we recorded the number of injected cache accesses per window and the percentage of the time when the protected program and Ghost Thread execute simultaneously. To handle non-real-time protection, we extend the analysis in Section 5 in the following way. For protection threads that are not real time, P_2 in the analysis is multiplied by the percentage of time Ghost Thread and the program co-execute. The order of magnitude of the normalized samples (e.g., 1E26 represents 10^{26}) is shown in the boxes near data points in Figure 7.

The results indicate that while Ghost Thread cannot completely eliminate cache side channels, it is infeasible to attack protected applications when we use real-time ghost threads, the default configuration of Ghost Thread. This is true regardless of the system load: it will require $10^{26}\times$ samples to perform the same attack on AES. For RSA, $10^{6}\times$ samples are needed to perform the same attack. To better connect normalized samples to attack difficulty (in terms of computation time), we collected the time to generate sufficient number of samples to successfully launch state-of-the-art attacks, as summarized in Table 5, under column "Time w/o GT". In order to compute the time needed to launch the same attacks with Ghost Thread, we use normalized samples to compute the corresponding time (assuming the same computation resource), shown in Table 5, under column "Time w/ RT GT". We see that for AES attacks, it will take over 1E17 years to complete, that is longer than the current age of the universe. For a more realistic attack scenario spanning across VM boundaries, we see the expected computational time is even worse, requiring at least 1E19 years of time to compile enough samples. Although the normalized samples is much smaller in the RSA case, due to the longer attack time on RSA, it still requires over 3,000 years to obtain enough samples.

Algorithm	Time w/o GT	Time w/ RT GT
AES [36]	0.064s	1E17 years
AES (Cross-VM) [18]	4.5s	1E19 years
RSA [47]	30 min	3170 years

Table 5: A summary of how long it takes to collect sufficient number of samples on our test machine to launch the corresponding cache attacks without and with Ghost Thread.

When one normal-priority ghost thread is used, the overall security is also affected by system load: when the system is under utilized, we observe the same level of protection as real-time ghost threads. But the normalized samples drop when the system is fully utilized. This is due to the fact that a normal-priority thread injects less noise due to scheduling, but still requires over 1 million times more samples (resp. 100 times more samples) to have the same success rate in the AES case (resp. RSA case). For AES, it takes approximately 3,000 years to collect a sufficient number of samples. While normal-priority ghost threads offer weaker security compared with real-time threads, they may be preferable when the reduced security is acceptable considering the gained performance.

Ghost Thread also provides the capability to add multiple threads to improve security. By using multiple protection threads more noise can be added per window and also increases the likelihood at least one protection thread is concurrently executing with the protected application. Our data indicates that adding one additional protection thread requires $10^{50}\times$ ($10^{12}\times$ for RSA) more samples compared to not having protection. This is an enormous amount of samples. Even when the system is loaded the additional protection thread performs between a single default priority thread Ghost Thread and a real-time thread requiring $10^{14}\times$ ($10^{4}\times$ for RSA) more samples. This many more samples will take over 200,000 (0.6 for RSA) years to collect using the same method used to compute the data in Table 5.

6.2.3 Adjusting Noise Frequency. Next, we evaluate the impact of noise injection rate on performance and security. The motivation is that for users who can tolerate weaker security, Ghost Thread can inject fewer noise accesses to reduce the performance cost. Users can also strengthen security by adding additional protection threads.

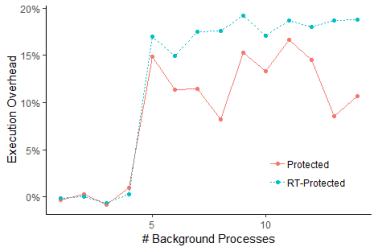

Figure 8: Percent overhead on bzip2 when a program protected by Ghost Thread is running concurrently.

Setup. The configuration is the same as in Section 6.2 with the addition of samples showing the performance and security of AES protected with Ghost Thread configured for various noise injection frequencies. Each of the Ghost Threads used have normal priority.

Results. To summarize our results, we find that lowering the number of memory accesses tends to lower the overhead of Ghost Thread. For instance, with 50 million accesses per second by Ghost Thread, the overhead with 6 background processes is 11% better than when the frequency is set to max (around 471 million accesses per second).

However, lowering the frequency also can significantly reduce the security improvement. When the protection thread is set to its maximum frequency, we observe normalized samples needed increase by 26 orders of magnitude on an under-utilized system. The required normalized samples decrease to 1E5 when the frequency is lowered to 150 million accesses per second and the system is being under utilized. Additionally, we find normalized samples drop from 1E10 at maximum frequency to 1E3 at 150 accesses per second when the system is fully utilized. Although tested on other configurations, we find that the rate of 150 Million offers a reasonable balance between security and performance and omit other results due to space limitations.

6.3 System Impact

Another concern to users is the impact of Ghost Thread on other programs running at the same time. There are two main reasons Ghost Thread could cause an increase in execution time for other applications. The first is increased contention for the cache, and the second is contention for CPU cores.

To evaluate the impact on other applications, we ran bzip2 and gobmk from SPEC2006 while running AES encryption in a loop with and without Ghost Thread. Ghost Thread is active for 100% of the duration of the execution of bzip2 and gobmk to demonstrate the worst-case overhead to the system. We selected those two programs since bzip2 represents a memory-intensive program and gobmk represents a CPU-intensive program. Due to the similarity between the two results, for space we present our findings for bzip2.

The results with bzip2 is shown in Figure 8. Figure 8 indicates that the overhead on other applications is again not significant until all of the CPU cores are being used. After all of the cores are being utilized, the overhead increases to its peak of about 16%, and then slowly declines as more processes are added. This experiment also shows that there is not a significant impact caused by Ghost Thread's added memory accesses since on an under-loaded system, there is no noticeable increase to the overhead. This is likely due to the fact Ghost Thread only accesses a small region of memory addresses to provide protection.

Note that since Ghost Thread is protecting the encryption the entire time, this experiment shows the worst-case overhead of Ghost Thread. As shown at the beginning of this section, applying Ghost Thread to applications in practice only requires protection for a fraction of the program's total lifetime. This will significantly reduce its impact on other applications running concurrently.

7 DISCUSSION

The experiments in Section 6 are conducted on a machine with a processor containing 6 cores. We note that since Ghost Thread utilizes additional CPU resources, the overhead added is proportional to the number of CPU cores available on a system. Thus, less powerful machines with less cores will see a higher overhead; a more powerful machine with more cores (e.g., cloud servers, a very important target of cache attacks) will see less overhead.

We note that although Ghost Thread is implemented as a shared library, the only data flowing to Ghost Thread from the application is the memory region to be protected along with the frequency and priority of the thread. Since the memory accesses made to the memory region by Ghost Thread are random, an adversary cannot gain any additional information by virtue of Ghost Thread being a shared library that is not already assumed to be public (e.g., the memory location of the S-Boxes for AES).

8 CONCLUSION

In this paper, we present a novel cache side channel defense Ghost Thread. We demonstrate its effectiveness at thwarting cache based side channel attacks by computing the increased difficulty in launching side-channel attacks. The overhead of applying Ghost Thread is negligible when a system is being under utilized and reasonable even when a system is being fully utilized. Ghost Thread is a readily deployable, software only, cache side channel protection system which delivers strong security guarantees with low overhead cost making it ideal for developers to get general cache-based side channel protection without costly specialized hardware.

9 ACKNOWLEDGEMENT

We thank the reviewers for their helpful feedback which we used improve our work. This research was supported by NSF grants CNS-1956032, CNS-1816282, CCF-1723571, and a gift from Intel.

REFERENCES

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. 2003. The EM Side—Channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002*, Burton S. Kaliski, çetin K. Koç, and Christof Paar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–45.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.

[3] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. cr.yp.to/papers.html#cachetiming.

[4] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. 364–380.

[5] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proc. 30th IEEE Symp. on Security and Privacy (S&P)*. 45–60.

[6] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.. In *NDSS*. 8–11.

[7] John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel Vulnerability Factor: A Metric for Measuring Information Leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 106–117.

[8] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proc. the 22nd USENIX Security Symposium (USENIX Security)*. 431–446.

[9] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 406–421.

[10] Úlfar Erlingsson and Martín Abadi. 2007. *Operating system protection against side-channel attacks that exploit memory latency.* Technical Report MSR-TR-2007-117. Microsoft Research. 7 pages.

[11] Jeff Fulme. 2018. siege. https://github.com/JoeDog/siege.

[12] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 182–194.

[13] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*. 217–233.

[14] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. 2016. Flush+Flush: A Stealthier Last-Level Cache Attack. In *DIMVA*.

[15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, Berkeley, CA, USA, 897–912.

[16] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. 2015. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. In *2015 IEEE Symposium on Security and Privacy*. 3–19.

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games—Bringing Access-Based Cache Attacks on AES to Practice. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 490–505.

[18] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *Proc. IEEE Symp. on Security and Privacy (S&P)*. 591–604.

[19] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross Processor Cache Attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 353–364.

[20] Gorka Irazoqui, MehmetSinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses*, Angelos Stavrou, Herbert Bos, and Georgios Portokalidis (Eds.). Lecture Notes in Computer Science, Vol. 8688. 299–319.

[21] Emilia Käsper and Peter Schwabe. 2009. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 1–17.

[22] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium*. 189–204.

[23] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[24] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–397.

[25] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A Language for Hardware-level Security Policy Enforcement. In *Proc. 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 97–112.

[26] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, Berkeley, CA, USA, 549–564.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[28] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 87–101.

[29] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 406–418.

[30] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *Proc. 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 203–215.

[31] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (Sept. 2016), 8–16.

[32] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*. 605–622.

[33] Stefan Mangard. 2004. Hardware countermeasures against DPA–a statistical analysis of their effectiveness. In *Cryptographers' Track at the RSA Conference*. Springer, 222–235.

[34] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Proc. 8th International Conference on Information Security and Cryptology*. 156–168.

[35] Dag A. Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006), 1–20.

[36] Dag A. Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006* (Jan. 2006), 1–20.

[37] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 431–446.

[38] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *16th ACM Conference on Computer and Communications Security (CCS)*. 199–212.

[39] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[40] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*.

[41] Kris Tiri, Onur Acıiçmez, Michael Neve, and Flemming Andersen. 2007. An analytical model for time-driven cache attacks. In *International Workshop on Fast Software Encryption*. Springer, 399–413.

[42] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. 2014. Scheduler-based Defenses against Cross-VM Side-channels.. In *USENIX Security Symposium*. 687–702.

[43] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 657–674.

[44] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Proc. the 26th USENIX Security Symposium (USENIX Security)*. 235–252.

[45] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proc. Annual International Symp. on Computer Architecture (ISCA)*. 494–505.

[46] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. 719–732.

[47] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA.. In *CHES (Lecture Notes in Computer Science)*, Benedikt Gierlichs and Axel Y. Poschmann (Eds.), Vol. 9813. Springer, 346–367.

[48] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proc. 20th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–516.

[49] Tianwei Zhang, Si Chen, Fangfei Liu, and Ruby Lee. 2013. Side Channel Vulnerability Metrics: The Promise and the Pitfalls. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. ACM, New York, NY, USA, Article 2, 8 pages.

[50] Tianwei Zhang, Fangfei Liu, Si Chen, and Ruby B. Lee. 2013. Side Channel Vulnerability Metrics: The Promise and the Pitfalls. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. ACM, New York, NY, USA, Article 2, 8 pages.

[51] Yinqian Zhang and Michael K. Reiter. 2013. DüPpel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*. 827–838.