Entropy-Shield:Side-Channel Entropy Maximization for Timing-based Side-Channel Attacks

Abhijitt Dhavlle
Dept. of Electrical and Computer Engineering
George Mason University
Fairfax, USA.
adhavlle@gmu.edu

Raj Mehta
Dept. of ECE
George Mason University
Fairfax, USA.
rmehta21@gmu.edu

Setareh Rafatirad
Dept. of Information Sciences and Technology
George Mason University
Fairfax, USA.
srafatir@gmu.edu

Houman Homayoun
Dept. of Electrical and Computer Engineering
University of California
Davis, USA.
hhomyoun@ucdavis.edu

Sai Manoj Pudukotai Dinakarrao
Dept. of Electrical and Computer Engineering
George Mason University
Fairfax, USA.
spudukot@gmu.edu

***Abstract*—The hardware systems have experienced a plethora of side-channel attacks (SCAs) in recent years with cache-based SCAs being one of the dominant threats. The SCAs exploit the architectural caveats, which invariably leak essential information during an application's execution. Shutting down the side-channels is not a feasible approach due to various restrictions, such as architectural changes and complexity. To overcome such concerns and protect the data integrity, we introduce *Entropy-Shield* in this work. The proposed Entropy-Shield aims to maximize the entropy in the leaked side-channel information rather than attempting to close the side-channels. To achieve this, the proposed Entropy-Shield introduces carefully and sensibly crafted perturbations into the victim application, thereby increasing the entropy of the information obtained by the attacker to deduce the secret key, while the information being observed looks legit yet futile. This methodology has been successfully tested on cache targeted SCAs such as Flush+Reload and Flush+Flush and the key information retrieved by the attacker is shown to be ultimately futile, indicating the success of proposed Entropy-Shield.**

***Index Terms*—Side-Channel Attack (SCA), hardware-security**

I. Introduction

Advancements in the design and complexity of modern computing systems facilitate encompassing a plethora of functional features to enhance performance and efficiency. Albeit advancements with evolved features such as cache-sharing and speculative execution that led to enhanced performance, they have been exploited for crafting security attacks, termed as side-channel attacks (SCAs). A wide variety of attacks have threatened the hardware security domain and SCAs are one branch of this domain. There have been a variety of previous works to address threats to the hardware like those posed by reverse engineering of hardware [1], attacks on machine learning based malware detectors [2], [3], cache based side-channel attacks [4], [5], etc. In this work, we address the issues with SCAs, propose a solution for such attacks , and discuss the past works. SCAs exploit the architectural vulnerabilities rather than the caveats in the application and utilize the side-channels or covert channels to extract the secret information from the system and are passive.

A rapid increase in the cache targeted SCAs are reported in recent times. To thwart such threats, our work focuses on defending against cache targeted SCAs. A plethora of cache targeted SCAs rely on the timing information to determine the cache-access (hit or miss) patterns to obtain the accessed addresses and eventually the secret key from the cache [6]–[10]. For instance, Flush+Reload SCA [6] depends on the assumption that the victim and the attacker share the same memory space and utilizes the cache-access timing information to retrieve the secret key from the system. Attacks such as Prime+Probe [11] supersedes the Flush+Reload attack by not requiring any shared memory space with the victim to extract sensitive information.

To address the challenges of cache targeted SCAs, techniques such as static cache partitioning [11], partition locked cache [12], non-monopolizable (nomo) cache architectures [13] and other works [14]–[16] are proposed. These techniques can tremendously reduce the interference between the attacker and the victim's memory access, thus providing a better defense. However, adopting such techniques require alterations in the cache design and also leads to performance degradation [11]. To overcome the limitations of the existing works such as cache-partitioning, randomization of cache architectures are introduced. The conventional fully associative cache is one of the preliminary randomization based methods, in which a memory line can be mapped to any of the existing cache lines. Similarly, any of the cache lines can be evicted in random, thus, preventing the leakage of cache-access information. Despite its security benefits, this technique incurs large delays and is power hungry [11]. In a similar way, random permutation cache [12], newcache [17], [18], random fill cache [19], and random eviction cache [11] strategies are implemented. Compared to the cache-partitioning, the randomization based solutions have shown higher robustness, yet the above-mentioned methods require modifications to the hardware and/or software and incur performance penalties. However, previously proposed defenses are confined to the specific attack, which makes it difficult to defend against an emerging wide range of attacks.

As a summary, the unsolved challenges and limitations of the existing defenses can be outlined as follows: a) side-channels are inevitable; b) hardware or software modifications can lead to enhanced security, but might not be practical to adapt; c) solutions such as VM migrations or switching leads to performance degradation. To overcome the limitations of previous works and thwart SCAs, here, we introduce Entropy-Shield, a defense for timing-based side-channel attacks. In contrast to the existing works that focus on architectural changes, the proposed Entropy-Shield primarily focuses on maximizing the entropy[1] of the side-channel information obtained by the attacker without interfering with the original functionality of the victim application. In the Entropy-Shield the original application is coupled with a protective application that is able to facilitate to introduce intelligent perturbations in the cache-access timing information obtained by the attacker. In contrast to existing randomization techniques, proposed Entropy-Shield introduces randomization under the constraint that the archived information by attacker looks legit and similar to the normal timing information, yet leading to the wrong key. The proposed Entropy-Shield introduces perturbations in the sequence by executing dummy functions that do not affect the functionality for the victim, but scrambling patterns observed by the attacker, thereby reducing the entropy and dissuading the attack. Proposed Entropy-Shield also offers two different modes of operation: uniform and deceptive modes, where the user can determine the mode to inject different types of perturbations. We would like to emphasize that, in this work 'entropy-maximization' refers to a reduction in the useful information obtained by an attacker over side-channels to decrypt the secret key, or in other words increasing the randomness of the data. The proposed Entropy-Shield technique is thoroughly evaluated against both active and passive cache targeted SCAs with victim utilizing different keys.

The primary contributions of this work are:

- In contrast to existing randomization techniques, crafted randomization in Entropy-Shield forces the wrong sequence to envisage as a legit pattern, thereby augmenting the entropy in the obtained information.
- Offer different modes of operation of the proposed shield, thereby giving liberty to the user to determine the level of the induced perturbations.
- Evaluate the security offered by the proposed defense on different encryption methods using different SCAs and secret keys.

[1]We define Entropy as the amount of randomness in the obtained data that tricks the attacker

The rest of the paper is organized as follows. Section II discusses the previous related works. Section III discusses the proposed Entropy-Shield. Experimental evaluation of the proposed Entropy-Shield against different SCAs are presented in Section IV. Section V concludes with the inferences and the contributions made.

II. Defenses against SCAs: State-of-the-Art

In order to secure the hardware systems against cache targeted SCAs, various defense techniques have been proposed that use different strategies. We discuss the most relevant and prominent ones in this section.

a) Isolation by Cache Partitioning [11]: Two processes that do not share a cache cannot snoop on each others cache activity. Thus, the idea of this approach is to assign to a sensitive operation its own cache set, and not to let any other programs share that part. As the mapping from memory to a cache set involves the physical memory address, this can be done by the operating system by organizing physical memory into non-overlapping cache set groups, also called colors, and enforcing an isolation policy. However, this leads to inefficient resource utilization and hardware overheads.

b) Access Randomization [12]: To overcome limitations of hardware-oriented approaches, randomizing the memory access is introduced, thus, making the attack much harder, even impossible. For instance, [11] uses random memory-to-cache mappings. There is a permutation table for each process, which enables a dynamic memory address to cache set mappings. This makes the attacker hard to evict a specific memory line of the victim process. However, maintaining the mapping and updating mapping tables penalizes performance.

In addition to these general defense techniques, there are many recent works in progress to minimize the cache SCAs. For example, Vladimir Kiriansky proposed a dynamically allocated way guard (DAWG) [20], a generic mechanism for secure way partitioning of set-associative structures, including memory caches. When applied to a cache, unlike the existing quality of service mechanisms such as Intel's Cache Allocation Technology (CAT), DAWG fully isolates hits, misses, and metadata updates across protection domains. DAWG requires additional techniques to block exfiltration channels different from the cache channel.

Similarly, Oleksenko proposed *Varys* [21], a system that protects unmodified programs running in SGX enclaves from cache timing and page table SCAs. The *Varys* takes a pragmatic approach of strict reservation of physical cores to security-sensitive threads, thereby preventing the attacker from accessing shared CPU resources during enclave execution. But the downside is it requires the application to monitor the SSA (SGX State Save Area) value, thus increasing the overhead. Stephen Crane in [22] explores software diversity as a defense against side-channel attacks by dynamically and systematically randomizing the control flow of programs. This diversity based technique instead transforms programs to make each program trace unique. This approach offers probabilistic protection against both online and off-line side-channel attacks. Chongxi Bao's work in [23] shows that 3D integration also offers inherent security benefits and enables many new defense mechanisms that would not be practical in 2D. Experimental results show that using their cache design, side-channel leakage is significantly reduced while still achieving performance gains over a conventional 2D system. Xiaowan Dong presents in [24] defenses against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel.

As seen, the present works either require hardware or software-stack modifications and/or incurs substantial performance penalties. In contrast, proposed Entropy-Shield works on the principle of maximizing the entropy through crafted perturbations with less performance loss.

III. Proposed Entropy-Shield

Though it seems the attacker can obtain the secret key in one iteration, it is nearly impossible to obtain in real scenarios due to the system noise and other system operations. Hence, the attacker needs to repeatedly execute the attack to extract the complete secret information, thereby filtering the system noise and other impacts. Unlike existing works, considering these factors, we propose Entropy-Shield, that protects the victim application by reducing the entropy of the side-channel, despite attacker executing victim application multiple times.

Listing 1. Spy inserts probes to monitor victim's cache lines

```
func Square(){.......
     Probe 1 —  Address 0x086f0 }

func Multiply(){.......
     Probe 3 — Address 0x08628 }

func Modulo/Reduce(){.......
     Probe 3 — Address 0x08616 }
```

Similar to all existing works [6] [7], the underlying assumption in successfully probing and eventually capturing secret data is that the attacker knows the addresses of the functions that perform sensitive operations. Shield has similar knowledge as the attacker, where it knows which sections of the victim code need to be monitored and protected. The implemented Entropy-Shield is shown in Figure 1, where the outcome of an encryption algorithm under SCA with and without our proposed has been presented in (b) and (a) respectively. We describe and illustrate the system with and without Entropy-Shield when SCA is launched below.

A. Side-Channel Attack without Entropy-Shield

Figure 1(a) shows how traditional Flush+Reload attack is able to spy on an (encryption) application to reveal the secret key. The spy inserts probes at the function addresses of non-trivial functions such as square, modulo, and the multiply operations as these are repetitive, and their sequence reveals the secret key bits. The spy constantly flushes the addresses at probed locations and monitors it again if it was accessed by the victim as shown in Listing 1. The spy does not insert anything into the victim's code for the probes; it means that the addresses are monitored for the victim's access, and when the probed location is accessed, the spy takes a note of it.

B. Crafted Perturbations in Side-Channel Information Through Entropy-Shield

For the ease of understanding of Entropy-Shield, lets assume the width of the secret key is only 16-bits of which after a large number of attempts the spy is able to capture only 4-bits of it, as shown in Figure 1(a). In such a case, the part of the key that both the spy (attacker) and victim (user) were able to deduce was "1001", meaning that the attacker was successful in stealing the secret data over the covert channel. If one can increase the entropy of the side-channel by introducing perturbations it is nearly impossible for the spy to steal the secret keys. Though one can assume inserting function calls randomly can introduce the perturbations, as done in previous works [19], it is not efficient. The reason for inefficiency is that the attacker can determine the presence of randomness or uniformity in the measured data through observed meaningless operation sequences and can perform post-processing or filtering to remove the noise and retrieve the key. In contrast, the Shield induces the perturbations that seem legit, yet deceptive. To induce such intelligent perturbations Entropy-Shield invokes the functions in an order from which the attacker can deduce a key, i.e., for instance, the victim calls the functions that would be executed if the secret key is '1', though the secret key bit is '0', thereby inducing additional noise through which entropy in the leaked information increases. Figure 1(b) shows how by reducing useful information in the covert channel and introducing crafted perturbations in the sequence, user (victim) observes the original key "1001" while the spy (attacker) observes it as "1111" for uniform and "1011" for deceptive mode. One shortcoming of such straightforward flipping(all 0's to 1's) is that the attacker can detect it and flip the bits. To thwart such a scenario, we introduce multiple modes of execution shown in Figure 1(b) using key symbols, discussed in the later section. The introduction of dummy operations is shown in Algorithm 1.

Listing 2. Uniform and Deceptive mode of operation

```
M1 = Uniform; M2=Deceptive
{func Modulo(fake, mode, multiply_done){
If (M1 and !multiply_done){
    Multiply(arguments= fake, uniform)}
elseif(M2 and !multiply_done){
    random call Multiply(argument=fake,random)}
elseif(M1 and multiply_done){
    do fake Modulo and disregard results;}
elsif(M2 and multiply_done){
    random call fake Modulo and disregard results;}
elseif(!fake) {do Modulo operation on bits;}}

func Multiply(fake, mode){
if (fake and mode=uniform){
    discard Multiply results;
    Modulo(fake, uniform, multiply_done)}
elseif (fake and mode=random){
    discard Multiply results;
    random call Modulo();}
elseif(!fake) {
    do Multiply operation on bits;}}}
```

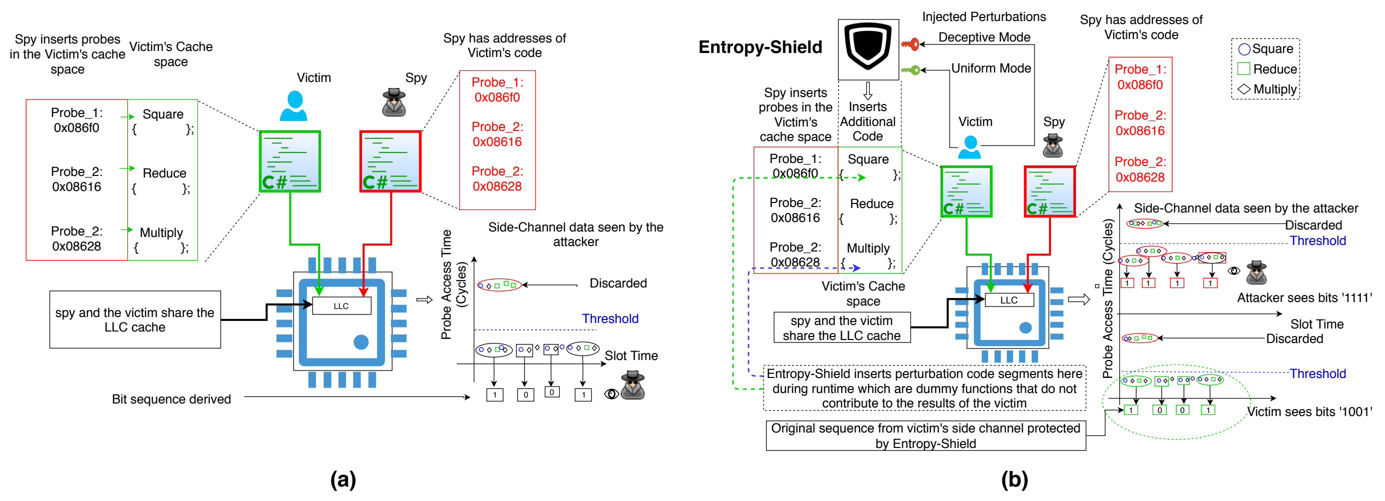
Spy inserts probes in the Victim's cache space
Victim's Cache space
Probe_1: 0x086f0
Probe_2: 0x08616
Probe_2: 0x08628
Square
Reduce
Multiply
Victim
Spy
Spy has addresses of Victim's code
Probe_1: 0x086f0
Probe_2: 0x08616
Probe_2: 0x08628
spy and the victim share the LLC cache
LLC
Side-Channel data seen by the attacker
Probe Access Time (Cycles)
Discarded
Threshold
Slot Time
Bit sequence derived
1
0
0
1
(a)
Entropy-Shield
Injected Perturbations
Deceptive Mode
Uniform Mode
Inserts Additional Code
Spy inserts probes in the Victim's cache space
Victim
Spy
Spy has addresses of Victim's code
Square
Reduce
Multiply
Victim's Cache space
spy and the victim share the LLC cache
Side-Channel data seen by the attacker
Discarded
Threshold
Attacker sees bits '1111'
Slot Time
Discarded
Threshold
Victim sees bits '1001'
Entropy-Shield inserts perturbation code segments here during runtime which are dummy functions that do not contribute to the results of the victim
Original sequence from victim's side channel protected by Entropy-Shield
(b)

Fig. 1. (a) Traditional side-channel attack on encryption algorithm where the data leaked via covert channel is accessible to the attacker; (b) Victim wrapped with Entropy-Shield that injects perturbation during run-time to perturb the sensitive information leaked thereby making SCAs laborious and time-consuming. *Only Uniform mode results have been shown*

C. *Modes of Operation in Entropy-Shield*

With the basic mode of operation, the Entropy-Shield perturbs the sequence of operations such that all the zeros(0's) in the secret key are converted to ones (1's) from the viewpoint of the attacker. It is beneficial when the user wants to add maximum noise to the sequence. However, to enhance the robustness of the Entropy-Shield and make it laborious for the attacker, our Shield is equipped with the capability to switch between two modes of perturbation: uniform and deceptive. The Uniform mode flips all the 0's in the sequence to all 1's. In the 'deceptive' mode, the shield randomly perturbs the sequence, which generates a sequence where only randomly selected 0's are converted to 1's. Effectively, this makes it difficult for the attacker to differentiate which bits belong to the original sequence generated due to the victim's operations and which ones were not. Moreover, in deceptive mode, the sequence perturbed or generated by the Shield is different in each iteration. Listing 2 shows the part of the code where either of the modes are selected. Each function call, Multiply or Modulo is given arguments, which helps it to recognize if the victim or the Shield made the call. For flipping 0's to 1's for the Uniform or the Deceptive mode, it requires that the Modulo call a fake Multiply function, and then the same Multiply function calls Modulo function again, so the sequence becomes Square-Modulo-Multiply-Modulo translating to bit '1'. Hence, the algorithm first checks if a fake call was made and it is the first call to Reduce in the sequence using "*if (M1 and !multiply_done)*"

After this is verified using the line shown above, the program proceeds to the Multiply function with the "fake" argument, which helps it to drop the results of the Multiply function and make a fake call. The program control then returns to Modulo, where this time, it knows that the call to itself is repeated, and it directly executes a fake call to itself and ignores the result. Across all the code, the Shield checks for the mode of operation and repetitively injects perturbations or randomly does it. Thus the cache is accessed, but at the same time, it does not affect the results of the victim. The victim code does not need to be modified because the results and/or the algorithm is not modified; it keeps running itself until it encrypts/decrypts the data with the secret key. Hence, in any mode of operation, the victim does not get affected or interrupted.

D. *Summary of Proposed Entropy-Shield*

Algorithm 1 consists of pseudo-code for the Entropy-Shield. Lines 2-19 belong to the victim's code encapsulated by Entropy-Shield with code for dummy functions. Line 9 checks for the mode of operation and then sets the Call flag accordingly. Lines 10 and 11 perform static perturbations and random perturbations based on the Uniform and Deceptive modes respectively. For the Multiply function which calls for Modulo function again, Lines 16, 17 and 18 perform similar functions. Probes at Lines 4, 8 and 15 are monitored by the spy and are not inserted by the Entropy-Shield. Lines 20-25 belong to the spy/attacker, which flushes a particular location (function addresses in this case) and reloads them to see if the victim accessed them. Line 26-28 compare the reloaded address's access time, and if it happens to be less than the threshold, then the address was accessed by the victim and not otherwise. With Flush+Flush, Line 25 would be absent. The perturbations added modify the sequence of executed operations, thus giving a notion of actual cache accesses made by the victim. Hence the attacker observes an incorrect key.

IV. Implementation Results

A. *Experimental Setup*

We tested the Entropy-Shield on a PC with Intel-i7 processor running Ubuntu 18.04.2 LTS OS with 16 GB RAM and GnuPG version 1.4.13. The Flush+Reload [6] and Flush+Flush [7] attack codes are deployed, which can be found at [25] and [26] respectively.

Algorithm 1 Pseudocode illustrating generation of perturbations with Entropy-Shield

```
Require: Private Encryption Key
Ensure: Decoded Incorrect Encryption Key
 1: Victim Program(Mode = Uniform or Deceptive) {Performs secure-critical
    operations that leak data over covert channel}
 2: func Square()
 3: { - - - - - - - - - - -
 4:     Probe 1 address
 5:     - - - - - - - - - }
 6: func Modulo()
 7: { - - - - - - - - - - -
 8:     Probe 3 address
 9:     Call = Uniform or Deceptive
10:     If (Call==Uniform) then {dummy call Multiply(); discard Modulo
    results}
11:     else if (Call==Deceptive) then {dummy call Multiply() at random
    intervals; discard Modulo results }
12:     - - - - - - - - - }
13: func Multiply()
14: { - - - - - - - - - - -
15:     Probe 2 address
16:     Call = Uniform or Deceptive
17:     If (Call==Uniform) then {dummy call Modulo(); discard Multiply
    results}
18:     else if (Call==Deceptive) then {dummy call Modulo() at random
    intervals; discard Multiply results}
19:     - - - - - - - - - }
20: Attack Program{Sample pseudo code that decodes the secret key}
21: Loop 1:
22:     clflush (Probe 1); clflush (Probe 2); clflush (Probe 3)
23:     wait for an interval;
24:     Reload Probe 1; reload Probe 2; reload Probe 3
25:     Measure reloading time(t)
26:     compare t ,# threshold time(th)
27:     if( t > th) => Cache miss
28:     if( t < th) => Cache hit
29: jump Loop1
30:   Based on perturbed sequence of Cache hit operation, Incorrect Secret
    Key is Deduced
```

B. *Entropy-Shield with Flush+Reload Attack*

In this section, we present the results of our proposed Shield. We have chosen the Flush+Reload and Flush+Flush attack spying on RSA-RSA and DSA-Elgamal encryption algorithms with the secret key of 4096-bits, as implemented in the GnuPG. We have evaluated two different secret keys. We also present the outcome of the Shield with different modes of operation - uniform and deceptive.

TABLE I
KEY AS VISIBLE TO THE ATTACKER AND THE VICTIM WITH ENTROPY-SHIELD - UNIFORM MODE OF OPERATION

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker
Flush+Reload	RSA-RSA	key_1	0FCFFF	0FCFFF	FFFFFF
	DSA-Elgamal	key_2	587BFA	587BFA	FFFFFF
Flush+Flush	RSA-RSA	key_3	54FF0B	54FF0B	FFFFFF
	DSA-Elgamal	key_4	89DE00	89DE00	FFFFFF

TABLE II
KEY AS VISIBLE TO THE ATTACKER AND THE VICTIM WITH ENTROPY-SHIELD - DECEPTIVE MODE OF OPERATION

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker	
					Iteration 1	Iteration 100th
Flush+Reload	RSA-RSA	key_1	0FCFFF	0FCFFF	5FDFFF	0FEFFF
	DSA-Elgamal	key_2	587BFA	587BFA	59FBFB	78FFFE
Flush+Flush	RSA-RSA	key_3	54FF0B	54FF0B	75FF1B	55FF1F
	DSA-Elgamal	key_4	89DE00	89DE00	CBDF02	8BDF01

We verified the efficiency of our proposed Entropy-Shield by examining the perturbations injected both on the victim and spy end. We modified the GnuPG's code to output the injected perturbations along with the sequence of square, modulo, and multiply operations. Figure 2 presents a graph of the sequence of operations plotted against time slots versus the probe time, as seen by the attacker/victim. Figure 2(a) shows the secret information observed by the victim and the attacker without the Entropy-Shield. Every Square-Modulo operation not followed by Multiply is translated as bit '0' and every Square-Modulo-Multiply-Modulo operation as a bit '1' [6]. The probe time in cycles has to be less than the threshold (value depends on the system, 125 in our work) value to be considered as accessed by the victim. For simplicity, we have not shown the operations that took higher than the threshold. In this case, the victim and the attacker both see the same information, meaning the victim continues to operate on encryption/decryption, and the attacker sees the same operations on the covert channel. Figure 2(b) shows the sequence of operations when the Entropy-Shield is protecting the victim in uniform mode. Hence, after perturbations are injected into the sequence, the victim observes the key as "011100000111" while the attacker sees it as "111111111111" since all the '0' bits are flipped to bit '1'. These perturbations are induced irrespective of the key, as shown in Table I with all zeros converted to ones. Figure 2(c) presents the operations as observed by the attacker with the deceptive mode of operation. The red-colored operations are injected perturbations and do not belong to the original sequence of the victim's activities. Unlike the uniform mode, the deceptive mode injects perturbations randomly and the bit positions that are perturbed change during every iteration of the victim as seen in Table II where the key "587BFA" is translated to "59FBFB" during iteration 1 and "78FFFE" during iteration 100. Parts of the sequence where perturbations were injected are highlighted in the tables.

C. *Entropy-Shield with Flush+Flush Attack*

We have evaluated our Entropy-Shield against a passive attack such as Flush+Flush, whose key extraction results are presented in Table I and II for both the modes. Similar to the Flush+Reload, the induced perturbations can deceive the spy in both uniform and the deceptive modes. For instance, in the uniform mode, the key gets translated from "54FF0B" to "FFFFFF" whereas for the deceptive mode, it is observed as "75FF1B" and "55FF1F" during iteration-1 and iteration 100, respectively. For our proposed defense to work even for Flush+Flush, it basically needs to ensure that the lines of code within the square, modulo or multiply functions is cached and only then it is possible for the attacker to flush a cache line within the code and consider that the function/operation must have been accessed by the encryption algorithm. Tables I and II are ideal case because while executing them on our machine we reduced the number of background activity. But, in actual scenarios, the OS and other application activity generate noise in the cache, making the attack more difficult and owing to which the attacker might not be able to see the key bits in consecutive order. Hence, as the keys observed by the attacker are different every time, and with such randomness, it is challenging for the attacker to retrieve the secret key. Given the fact that executing SCAs is non-trivial when it comes to retrieving secret keys amid operating system noise and various cache operations. The working principle remains the same for both the RSA and DSA encryption.

D. *Performance of Entropy-Shield*

Since Entropy-Shield includes additional functional calls, this would incur overheads in terms of execution time. To analyze the overheads, we have executed the RSA and DSA algorithms with and without Entropy-Shield for 1000 times and averaged the execution time to remove the noise impacts in measurements. We executed our proposed Entropy-Shield for the encryption methods mentioned in Table II and with different keys for over 1000 times. It has been observed that with the proposed Entropy-Shield, the execution time increases by 8% on average, which is significantly small and can also happen in the presence of system noise, thus can be ignored, and also an attacker cannot detect the Entropy-Shield based on runtime.

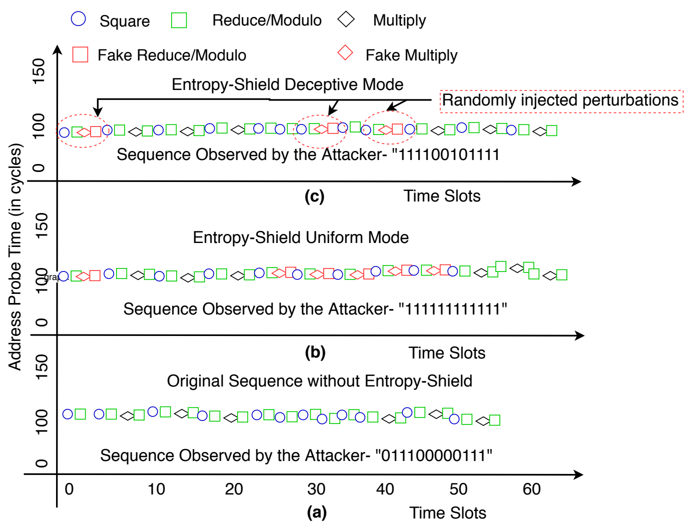

Fig. 2. Plot of sequence of operations (a) Original sequence of operations without Entropy-Shield as seen by both the victim and the attacker; (b) Sequence of operations with Entropy-Shield in Uniform mode are seen differently by the attacker and victim; (c) Sequence of operations with Entropy-Shield in deceptive as seen differently by the attacker and victim

V. Conclusion

In this work, we discussed the threats side-channel attacks (SCAs) posed to the computing systems and delineated the available defense mechanisms proposed in the past. The downsides of the previous works are that they require significant modifications to the hardware or software architectures to safeguard cache subsystems. Hence, we proposed Entropy-Shield which can protect applications from SCAs by reducing the amount of useful information leaked on the covert channel. We verified the efficacy of Entropy-Shield on Flush+Reload and Flush+Flush attack with RSA and Elgamal encryption methods as victims and found it to be successful. The average overhead with our proposed shield is 8% compared to without the defense in place. Our approach can easily be modified to suit a variety of applications.

References

[1] G. Kolhe and et.al., "Security and complexity analysis of lut-based obfuscation: From blueprint to reality," in *Int. Conference On Computer Aided Design*, 2019.

[2] S. M. P. Dinakarrao *et al.*, "Adversarial attack on microarchitectural events based malware detectors," in *Design Automation Conf.*, 2019.

[3] S. Shukla and et.al., "Microarchitectural events and image processing-based hybrid approach for robust malware detection: work-in-progress," in *Embedded Systems Week*, 2019.

[4] F. Brasser *et al.*, "Advances and throwbacks in hardware-assisted security: Special session," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2018.

[5] A. Dhavlle *et al.*, "Work-in-progress: Sequence-crafter: Side-channel entropy minimization to thwart timing-based side-channel attacks," in *International Conference on Compliers, Architectures and Synthesis for Embedded Systems (CASES)*, 2019.

[6] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Conference on Security*, 2014.

[7] D. Gruss *et al.*, "Flush+flush: A fast and stealthy cache attack," in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016.

[8] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*, 2006.

[9] Y. Zhang *et al.*, "Cross-VM side channels and their use to extract private keys," in *ACM Conf. on CCS*, 2012.

[10] D. Harnik *et al.*, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security Privacy*, vol. 8, no. 6, pp. 40–47, Nov 2010.

[11] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the IEEE/ACM*, 2017.

[12] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the ISCA*, 2007.

[13] L. Domnitser *et al.*, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 35:1–35:21, Jan. 2012.

[14] E. Brickell *et al.*, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities." *IACR Cryptology ePrint Archive*, vol. 2006, p. 52, 01 2006.

[15] J. Kong *et al.*, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *ACM Workshop on Computer Security Architectures*, 2008.

[16] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint Archive*, vol. 2005, p. 280, 2005.

[17] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008.

[18] F. Liu *et al.*, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[19] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[20] V. Kiriansky *et al.*, "DAWG: A defense against cache timing attacks in speculative execution processors," in *MICRO*, 2018.

[21] O. Oleksenko *et al.*, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *USENIX*, 2018.

[22] S. Crane *et al.*, "Thwarting cache side-channel attacks through dynamic software diversity," in *In Network and Distributed System Security Symposium*, 2015.

[23] C. Bao and A. Srivastava, "3d integration: New opportunities in defense against cache-timing side-channel attacks," *IEEE (ICCD)*, 2015.

[24] X. Dong *et al.*, "Shielding software from privileged side-channel attacks," in *USENIX Security Symposium*, 2018.

[25] T. Hornby. (2016) Flush+reload attack. Last accessed: 15-July-2019. [Online]. Available: https://github.com/defuse/flush-reload-attacks

[26] (2017) Flush+flush attack. Last accessed: 15-July-2019. [Online]. Available: https://github.com/IAIK/flush_flush/tree/master/sc/ff