Imitating Functional Operations for Mitigating Side-Channel Leakage

Abhijitt Dhavlle, *Graduate Student Member, IEEE*, Setareh Rafatirad, *Senior Member, IEEE*, Khaled Khasawneh, *Member, IEEE*, Houman Homayoun, *Senior Member, IEEE*, and Sai Manoj Pudukotai Dinakarrao, *Member, IEEE*

***Abstract*—Inspired by the idiom, "Mitigation (prevention) is better than cure!", this work presents a random yet cognitive side-channel mitigation technique that is independent of underlying architecture and/or operating system. Unlike malware and other cyber-attacks, side-channel attacks (SCAs) exploit the architectural and design vulnerabilities and obtain sensitive information through the side channels. In contrast to the existing randomization-based side-channel defenses, we introduce a cognitive perturbation-based defense, Covert-Enigma, where the introduced perturbations look legit, but lead to an incorrect observation when interpreted by the attacker. To achieve this, the perturbations are injected at appropriate time instances to introduce additional operations, thereby misleading the attacker making the extracted data futile. To further make the attack more intricate for the attacker, proposed Covert-Enigma offers two modes of operation, chosen by the user, to determine the kind of induced cognitive perturbations—*arbitrary* and *cyclic* modes. Arbitrary mode selects a group of key bits and flips them during every execution of the victim. Cyclic mode exhibits similar behavior, except it selects a new set of bits to flip after "*N*" cycles as chosen by the user. The cognitive perturbations are introduced in the form of a wrapper application to the victim, thus imposing no requirements on architectural level modifications nor soft updates/edits to the operating system. We report rigorous evaluation of the proposed Covert-Enigma protecting RSA cryptosystem attacked by Flush+Reload crypto SCA along with the bit(s) recovered after observing RSA under attack. Compared to traditional randomization-based defenses, proposed cognitive Covert-Enigma leads to 50% less overhead.**

***Index Terms*—Cryptosystems, hardware security, side-channel attack (SCA).**

I. Introduction

MODERN computing systems require designers to embed novel features to satisfy the ever exploding need for high performance and efficient systems. Regardless of evolved features, such as speculative execution, three-level cache architecture, memory sharing/deduplication, etc., the computing systems are vulnerable to security threats, also known as side-channel attacks (SCAs). A plethora of past research works has focused on the vulnerabilities in the systems. Some of the works on the vulnerabilities and their exploits are: malware [2]–[4], reverse engineering of hardware [5], [6]; attacks on machine learning-based malware detectors [7]–[10], cache-based SCAs [1], [11], [12]. SCAs exploit the architectural vulnerabilities, such as timing, power, frequency, etc., in the victim application. By exploiting such vulnerabilities, the SCAs attempt to steal confidential data from sensitive applications. There have been a rapid increase in the cache-targeted SCA [13]–[15]. Computing systems require cache to achieve performance gains. Hence, almost all the applications (sensitive or generic) utilize cache for storing recently accessed memory locations. For instance, cache-targeted SCAs rely cache-access patterns—hit or miss—to determine the recently accessed location(s) [16]–[20]. By studying such patterns that serve as a covert channel leaking sensitive information, the attacker can determine the recently accessed location, hence the secret information. To thwart such emerging threats, our work focuses on defending against cache targeted SCAs.

The unsolved challenges and limitations of the existing defenses can be outlined as follows: 1) suggested hardware or software modifications might not be feasible to adapt and 2) VM[1] (virtual machine) migration-based mitigation are resource hungry strategies, and contribute to a significant timing overheads.

To overcome the limitations of previous works [21]–[23] and thwart SCAs, we introduce Covert-Enigma, a defense for timing-based SCAs. In contrast to the previously mentioned existing works that focus on architectural changes, the proposed Covert-Enigma primarily focuses on maximizing the entropy[2] of the side-channel information obtained by the attacker without interfering with the original functionality of the victim application. In the Covert-Enigma, the original application is coupled with a protective application

Manuscript received October 14, 2020; revised January 25, 2021; accepted March 8, 2021. Date of publication March 31, 2021; date of current version March 21, 2022. The preliminary version of this work is published in ISQED 2020 [1]. This article was recommended by Associate Editor S. Ghosh. *(Corresponding author: Abhijitt Dhavlle.)*

Abhijitt Dhavlle, Khaled Khasawneh, and Sai Manoj Pudukotai Dinakarrao are with the Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA 22030 USA (e-mail: adhavlle@gmu.edu; kkhasawn@gmu.edu; spudukot@gmu.edu).

Setareh Rafatirad is with the Department of Applied Information Technology, George Mason University, Fairfax, VA 22030 USA (e-mail: srafatir@gmu.edu).

Houman Homayoun is with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616 USA (e-mail: hhomayoun@ucdavis.edu).

Digital Object Identifier 10.1109/TCAD.2021.3070243

[1]In a multitenanted cloud environment, the operating system (along with the victim application) is moved and executed on another physical hardware, disallowing the co-location of victim and the attacker OS.

[2]We define Entropy as the amount of randomness in the obtained data. The less entropy information has, the easier it is to decrypt the data.

(wrapper) that induces *cognitive* perturbations in the cache-access information obtained by the attacker.

In contrast to the existing randomization techniques, proposed Covert-Enigma introduces randomization under the constraint that the archived information by attacker looks legit and similar to the normal information, yet leading to a wrong key. In Covert-Enigma, we induce cognitive perturbations in the (security-sensitive) applications' operations by executing dummy instructions that leave the victim's functionality unaltered yet scrambling the sequence observed by the attacker. These induced cognitive perturbations mislead the information retrieved by the attacker, thereby thwarting the attack. Our proposed Covert-Enigma tenders user-tunable parameters, such as the length of successive bits to modify and the cycle frequency, where the number of cycles can be chosen after which the proposed method cognitively selects the other set of bits to perturb. This offers the user to adjust the level of complexity of the injected perturbations. Arbitrary and Cyclic are two operational modes that a user can select, and the details are discussed in other sections. The arbitrary mode offers one or more bits to be cognitively perturbed in the sequence of operations chosen at runtime, whereas the Cyclic mode chooses bit(s) and then keeps perturbing[3] same bits[4] for few executions as determined by the user, post which the position changes. The Cyclic mode is advantageous when the attacker suspects a defense mechanism is in place and tries to repeatedly execute the user application to ascertain the static part of the sequence—as seen by the attacker—which are added perturbation(s). We want to emphasize that, in this work, "entropy-maximization" refers to a reduction in the useful information obtained by an attacker by increasing cognitive randomness over side channels to decrypt the secret key. The proposed Covert-Enigma technique is thoroughly evaluated against active and passive cache-targeted SCAs with victim applications utilizing different keys.

The cardinal contributions of this work are as follows.

1) Contrary to the existing works, the proposed Covert-Enigma enforces security on the covert channel by injecting cognitively crafted perturbations that imitate legit operations yet mislead the attacker.
2) Render the attack more time consuming (in terms of the iterations it takes to break the defense) by providing two modes of operation, Arbitrary and Cyclic, thus offering more flexibility in terms of the defense.
3) Evaluate and compare the benefits of the proposed Covert-Enigma in terms of overhead and performance based on the key size, mode of operation, user-tunable parameters, and the number of bits recovered post-attack on the victim.

The rest of the work is organized as follows. Section II provides the working principle of the flush+reload and flush+flush type SCAs followed by the vulnerability in encryption application. Section III describes the proposed defense, threat model, generation of cognitive perturbations, and modes of operation of the proposed Covert-Enigma. This is followed by Section IV which includes the validation process, recovery of sensitive data under SCA attacks (without the presence of Covert-Enigma), the behavior of Covert-Enigma under attack, the performance of Covert-Enigma and the overhead analysis. Section V describes the motivation supporting the proposed idea as a case study, followed by the state of the art in Section VI. Section VII concludes the work.

[3]Perturbation or cognitive calls refer to dummy cache accesses that leads to meaningful decryption, yet incorrect.

[4]Bit here refers to the bit in the secret information. Bit position refers to the bit in the stream of secret information to be protected as observed by the adversary.

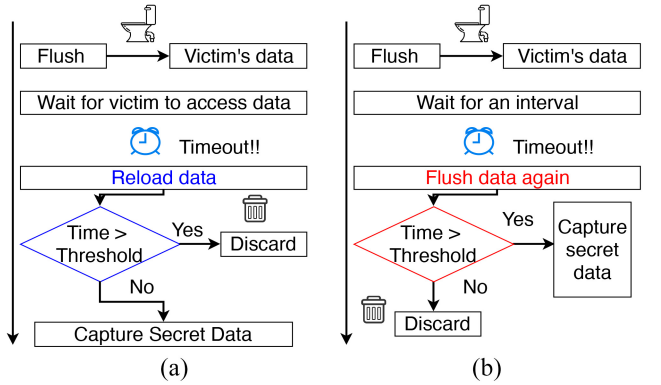


Fig. 1. (a) Flush+Reload attack: the spy (attacker) flushes the data and waits to determine whether victim accessed the flushed line or not. (b) Flush+Flush attack: the spy (attacker) flushes victim data, waits for a short interval and reflushes the same location to observe the time it takes to flush the data, thus, deciding if the data was accessed by the victim.

II. Side-Channel Attacks: Background

This section will briefly introduce the SCAs on which the evaluation of proposed Covert-Enigma is performed along with some previous works.

A. Side-Channel Attacks

1) Flush+Reload Attack: Flush+Reload is a prominent cache targeted SCAs that utilizes the cache-access timing information to retrieve the key. The process of Flush+Reload attack is performed in three steps, as follows.

1) *Step 1:* The attacker (spy) flushes a memory line in the (shared) cache.
2) *Step 2:* Spy waits for a certain amount of time (to let the victim access the cache).
3) *Step 3:* After the timeout, the spy reloads the data into the cache and observes the access time to determine whether the cache line was accessed by the victim or not and in Fig. 1(a).

Thus, the Flush+Reload attack can be inferred as follows: if there was a cache hit for the spy application indicates that the cache line (data) was accessed (and fetched) by the victim application, else the victim does not utilize the data. For instance, the encryption algorithms, such as GnuPG's RSA encryption use a sequence of the square, reduce and multiply operations to calculate the private key's exponent. Utilizing the Flush+Reload attack, depending on the cache hit/miss and the sequence of the Square, Modulo, and Multiply operations, the spy deduces if the bit in key was a logical "1" or

"0". By continuously repeating the above process, the attacker can retrieve the entire private key [16].

2) Flush+Flush Attack: Flush+Flush attack [17] is a relatively advanced cache targeted attack that supersedes the above discussed Flush+Reload SCA both in terms of speed and stealthiness. The Flush+Flush attack is shown in Fig. 1(b). Unlike the Flush+Reload attack, Flush+Flush is passive and works only by executing `clflush` instruction in an infinite loop. Unlike Flush+Reload, Flush+Flush attack does not access any data, the number of cache misses thus created are zero, and hence it becomes difficult to detect. When the `clflush` instruction is issued, data that is cached takes more time to be flushed out of the cache as it has to be evicted out across all cache levels completely as against noncached data, which takes less time. Based on the execution time of the `clflush` the Flush+Flush attack concludes if the data was cached or not cached. The attack does not load any memory line into the cache, and hence if `clflush` takes more time to execute would imply that the victim accessed the data. Based on this strategy, the attack monitors the victim's activities by observing multiple cache lines or data of the victim.

B. GnuPG Encryption

In the previous section, we studied the SCAs, and here we describe briefly the victim application that we utilize as a case study to analyze the impact of proposed Covert-Enigma. GnuPG's public-key encryption (PKE) is a popular way of encrypting the data to maintain confidentiality and integrity. The PKE generates a pair of public and private keys, the width of which is decided by the user. Any document that is encrypted with a public key can only be decrypted with the corresponding private key. Let us say user A wants to send secret data to user B. In such a case, B will have its own public and private key, of which the public key will be made available to user A. User A will use user B's public key to encrypt the secret data and send the encrypted data to user B. To reveal the secret data, user B will decrypt the file with its private key. The way the RSA algorithm is implemented, it is nearly impossible to brute force an encrypted file if the width of the secret keys is large enough and also due to the known fact that users (both legitimate and attackers) have no access to the RSA algorithm directly while it is in the process of encryption and decryption. This might have been true until a few years ago, but not anymore due to the state-of-the-art SCAs that have successfully broken the keys' secrecy, thereby rendering the PKEs vulnerable to attackers. We have discussed the GnuPG's implementation of the RSA and the DSA (with Elgamal) type encryption methods in this work. A series of complex calculations compute the private keys, and the exponent is what the attackers try to target. Once the exponent is captured over the covert channel, the algorithm can be easily broken. Work in [16] vividly describes how SCA can be used to spy on victim's (RSA) operations and thus steal secret data.

```
function Square(){.......
        Probe 1 //  Address 0x086f0
        .......... }

function Multiply(){.......
        Probe 2 // Address 0x08628
        .......... }

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

Listing 1. Spy inserts probes to monitor targeted vulnerable functions in the victim.

III. Design and Implementation of Covert-Enigma

In this section, we will first discuss the challenges that need to be addressed to deploy a successful SCA defense. Further, we present the attack model used in many of the existing works.

Some of the cardinal challenges designers face while incubating any defense in place to protect the victim are: the defense mechanism should serve as a transparent shield and does not alter the victim application's functionality. Second, the attacker executes the application for a large number of times to reduce noise in the channel while trying to capture the desired secret information. In such a scenario, the defense mechanism must ensure that the victim application is guarded against such attack methodology while reducing useful information leaked to the attacker. Finally, but crucial, the defense mechanism must not significantly add overhead to the system while trying to protect the victim application. Covert-Enigma draws inspiration from adversarial learning [24] where we introduce pixel-level perturbation for forcing misclassifications on the attacker's end. In Covert-Enigma we perturb the cache-access sequence by using cognitive operations to mislead the attacker.

A. Attack Model

We assume an adversary whose intention is to steal the confidential data that the victim is processing. For the Flush+Reload and Flush+Flush attack to work, sharing the cache space with the victim is a prerequisite. The spy does not need access to privileged execution mode; instructions such as *clflush* are allowed for user-level processes. The Covert-Enigma does not require superuser privileges as well. It is realistic to assume that the spy knows the addresses to monitor the victim. The Covert-Enigma has similar knowledge of the same addresses of interest to shield the victim against the attacker [16]. The spy can execute on any core as the last-level cache (LLC) is shared across all the cores. Given the attack happens in a real-world setting, we assume that the adversary does not have the right/control to execute the victim at the same time as the adversary, but rather, the adversary can observe a part of the victim's execution during each run. Also, referring to [16], the adversary cannot capture successive victim cache accesses that happen before the next probe (monitored addresses) monitoring cycle.

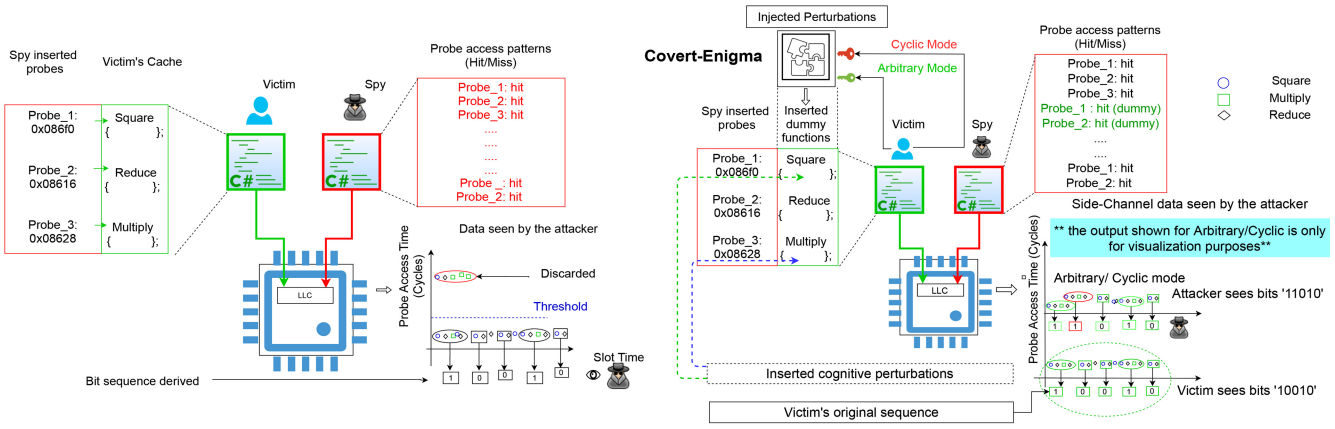


Fig. 2. (a) Traditional SCA on encryption algorithm where the data leaked via covert channel is accessible to the attacker. (b) Victim wrapped with Covert-Enigma that injects perturbation during run-time to perturb the sensitive information leaked thereby making SCA time consuming. *the output shown is only for visualization purpose*.

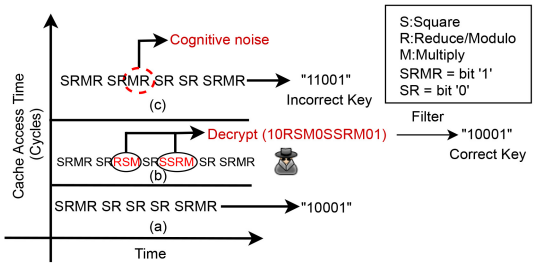


Fig. 3. (a) Sequence of operations in RSA implementation that leaks secret data. (b) Random cache accesses [21], [25], [26]. (c) Cognitive perturbation injected in the observed side-channel data to secure the information.

B. Side-Channel Attack Without Covert-Enigma

Fig. 2(a) shows the working methodology of traditional Flush+Reload attack to spy on a victim (encryption) application to reveal the secret key. The spy inserts probes at the function addresses of nontrivial functions, such as square, modulo, and the multiply operations as these are repetitive and their sequence determine the data flow and reveals the secret key bits—this is how the existing cache-targeted SCAs [16], [17] function. In the case of Flush+Reload attack, the spy (attacker) constantly flushes the addresses at probed locations and monitors if the victim accesses the flushed lines. The process of probing the square, multiply, and modulo/reduce encryption functions by the attacker is shown in Listing 1. Referring to Fig. 2(a), the attacker is able to retrieve the secret information.

C. Covert-Enigma: Injecting Cognitive Perturbations

Introducing random operations[5] as in existing works to induce perturbations is not efficient as the attacker can filter out portion that does not contribute to the construction of secret data [21], [25], [26]. Fig. 3 shows an example of cognitive perturbations injected during the victim's execution.

[5]By random, we mean that random injection of any fake/dummy cache access does not help much to mislead the attacker.

Part (a) presents a sequence of operations that decodes to "10001." Part (b) shows random sequences injected, but these random calls do not make sense in the context of the secret data revealed by the victim. For example, adding a reduce-square-multiply operation, as shown in 3(b), does make the secret data retrieval difficult, yet it does not force the attacker to translate a "0" bit to "1" or *vice versa*. In other words, randomization does not lead to misinterpretation of the secret data and can be filtered out by the attacker. Hence, it is crucial to introduce perturbations cognitively that seem legit.

In this work, we consider RSA implementation [27] as the victim. For RSA, to induce cognitive perturbations, the Covert-Enigma makes cognitive calls to the code within the functions square, reduce and multiply, responsible for crypto-operation. For instance, a sequence of square-reduce operations corresponds to bit "0," whereas a sequence of square-reduce-multiply-reduce will correspond to bit "1" [16]. These operations are implemented as function calls in the GnuPG's encryption suite [27], [28]. Hence, by making a dummy function call to the Multiply followed by the reduce function, the defense can pose as if the sequence corresponds to bit "1" whereas the actual secret bit was "0." We term this phenomenon as the elevation of entropy. It is to be noted that the reverse operation holds true as well, injecting perturbations such that sequences corresponding to bit "1" are observed as a "0." After cognitive perturbation, the series contains additional "multiply" and "reduce" operation, as shown in part (c). With these additional accesses, the sequence is deduced as "11001." The implementation of this technique does not modify the victim's original functionality. Referring to Fig. 2(b), the attacker observes "11010" instead of the original sequence of "10010," given the cognitive perturbations.

The cognitive perturbations are dummy calls as they are not a part of the victim's original operations. Referring to Listing 2, the functions function_1 and function_2 are victim's original operations. In the function_main, the result of either function_1 or function_2 is fetched from the cache. We say the function call is a dummy call when the function's

```
1  function_1 {
2      a = cache_location(100);
3      return a; }
4
5  function_2 {
6      b = cache_location(170);
7      return b; }
8
9  function_main {
10     dummy_flag = 0;
11     if(bit==0)
12         result = function_1;
13
14     else if(bit==1)
15       result = function_2;
16
17     dummy_flag = 1;
18     if(bit==0)
19         result = function_1;
20         discard(result)
21     else if(bit==1)
22         result = function_2;
23         discard(result)
24                 }
```

Listing 2. Example of a dummy operation.

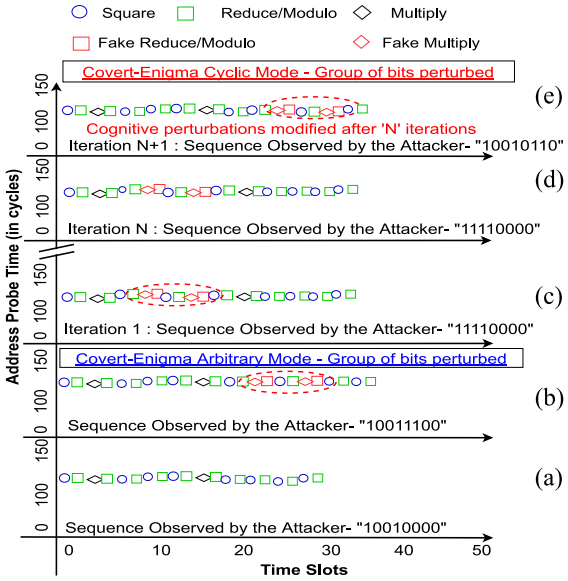


Fig. 4. (a) Part of secret seen by both adversary and victim without Covert-Enigma. (b) Sequence of bits seen by attacker when victim application is protected by Covert-Enigma arbitrary mode, where positions of the perturbed bits change each run. (c) Sequence seen by adversary with Covert-Enigma cyclic mode where position of group of perturbed bits remains same until iteration "*N*." (d) Bit positions from previous run remain same. (e) Bit positions have shifted randomly during new "cycle" of same execution.

obtained result is discarded, meaning that the victim did not use the result. The cache access is only made to perturb the sequence of operations or cache accesses. The dummy calls are injected by Covert-Enigma as a part of the defense mechanism to trick the attacker into observing the sequence of operations the victim performs, including the injected dummy calls. The attacker depends on the cache access patterns, indicated by probe hit/miss, to steal secret information. Hence, by injecting dummy operations, the attacker observes the victim's original operations perturbed by dummy operations. These injected perturbations translate to misleading secret information different than the original information without injected perturbations.

D. Generation of Cognitive Perturbations

The generation of cognitive perturbation is intended to render the observance of sensitive information (over the side channel) during the attack a time-consuming task. With cognitive perturbations, it becomes not only a time-consuming task for the attacker, but it also becomes difficult on the attacker's part to differentiate between the cache accesses caused by the defense in place versus those caused by the victim application. The adversary monitors the probed addresses. Therefore, the adversary is aware of the pattern of the victim's accesses when it executes the probed code lines. Our motivation in introducing cognitive perturbation is that if we can carefully craft cache accesses such that they would be considered legit by the adversary, it would be dummy operations that the victim makes to increase the entropy in the side channel. Because these operations, though dummy in nature, make real cache accesses, they are considered by the adversary as an operation made by the victim while processing sensitive data. These cognitive operations need not be the replica of the functions found in a victim. Still, they could be simple lines of code that reload the same addresses[6] as are flushed by the adversary.

We build cognitive operations that execute (and access cache) similar to what the victim's original functions would do by simply reloading addresses in the memory corresponding to the lines of code in the victim's original operations. These cognitive operations are considered legit by the adversary application, as will be evident in Section IV, where we present and analyze the experimental results.

Addition of the cognitive perturbations might present the notion of additional power consumption, which may be used for other forms of SCAs. However, the dummy function calls are limited in number, and the workload of these functions is miniature in nature. Thus, the amount of additional power or latency introduced by the Covert-Enigma is small. In addition, to perform a power-based SCA, the basic assumption would be that the attacker has power signatures for all the victim application(s) and can reliably compare the same with the golden power traces. However, the power trace collection involves uncertainties from different system components, which could be similar to additional power consumption by the introduced dummy operation of Covert-Enigma. Furthermore, the power SCAs are beyond the scope of this work.

E. Covert-Enigma Modes of Operation

To enhance the robustness of the Covert-Enigma, the Covert-Enigma is equipped with two modes of operation-arbitrary and cyclic. Each mode can be set by the user to inject the corresponding level of perturbations. The tuning of the parameters in the mode refers to the reconfigurable aspect of Covert-Enigma. The reconfigurable parameters are "total bits perturbed" for the Arbitrary mode; and "total bits perturbed" and "Cycle Iterations (N)" for the Cyclic mode. A brief view of the Arbitrary and Cyclic modes is presented in Table I. The results highlight the injected cognitive perturbations to mislead the attacker/adversary. Table I shows groups

[6]The vulnerability in the victim application is known to the adversary.

```
1  Attack {
2      Loop 1: clflush (Probe 1);
3      clflush (Probe 2);
4      clflush (Probe 3);
5      Reload Probe 1, Probe 2 and Probe 3;
6      wait for time = t_wait;
7      t= Measure Reloading time;
8      jump Loop1 ;
9  cmp (t, threshold time(th));
10 if( t > th) => Cache miss;
11 if( t < th) => Cache hit; }
```

Listing 3. Attack code to capture data.

of bits perturbed; the detailed results with Flush+Flush and Flush+Reload are discussed later.

1) Arbitrary Mode: As in Fig. 4(b), the *arbitrary* mode cognitively perturbs a group of cache operations by calling dummy operations to elevate the randomness. The *arbitrary* mode randomly selects positions to call dummy operations. To avoid keeping the number of successive dummy operations static, arbitrary mode randomly groups cache operations (of the victim) and inserts dummy cache accesses[7] in between two successive victim cache access. The random bit position selection is explained in Section III-E3.

2) Cyclic Mode: As illustrated in Fig. 4(c)–(e), our Covert-Enigma supports *cyclic* mode of operation, where a group of bits is selected at random and cognitively perturbed, and the group selected stays the same for a few cycles (execution runs, in other words) determined by the user. Post the cycle count (details in the next section), a different set of bits is selected to insert dummy operations. In summary, the perturbed bit positions change every few cycles (denoted as "N") selected by the user, and for new executions, i.e., at "$N + 1$" cycle, new bits are selected, the position of which remains the same for another "N" runs, as shown in Fig. 4(d). The duration of the cycle is denoted as "N" where N is an integer. After "N" cycles, a new bit or set of bits are selected to perturb cognitively, and the sequence seen by the attacker changes; this is shown in Fig. 4(e). The positions of these bits are random during every run and reduce the secret bits recovered during an attack by elevating the randomness in the side channel.

3) Generation of Random Bit Positions: We introduced the two modes of operation previously. Both *arbitrary* and *cyclic* modes require random numbers to be generated to select bit position to inject cognitive perturbation(s). In such a case, Intel's *RDRAND* [29] and Linux's */dev/random* [30] can be utilized. The true random number generator (TRNG) generates "true" random numbers based on random, nondeterministic noise generated by the device drivers into an entropy pool, which returns random numbers. The Covert-Enigma utilizes these random numbers to generate cognitive noise.

F. Summary of Covert-Enigma

Algorithm 1 outlines a high-level simplified view of the proposed Covert-Enigma along with a snippet of the victim and the attack code. Covert-Enigma part is presented in Algorithm 1 from lines 15–39. The user needs to feed in

[7]Cache is accessed but does not contribute to the functionality of the victim's operations.

Algorithm 1 Pseudocode Illustrating Generation of Perturbations With Covert-Enigma and the Modes of Operation

```
Require: Private Key
Ensure: Decoded Incorrect Key
 1: Victim Program (Mode = Arbitrary or Cyclic)
    {// Performs secure-critical operations that leak data over covert channel}
 2: func Square()
 3:    { Probe 1 inserted here
 4:    Do Square operation;
 5:    Wait for the Covert-Enigma; }
 6: func Reduce()
 7:    { Probe 3 inserted here
 8:    Do Reduce operation;
 9:    Wait for the Covert-Enigma; }
10: func Multiply()
11:    { Probe 2 inserted here
12:    Do Multiply operation;
13:    Wait for the Covert-Enigma; }
14: Covert-Enigma (){
15:       position key_size = 1024/2048/3076 or 4096;
16:       position_array = true_random_generator();
17:       successive_bits_array = true_random_generator();
18:       tamper_proof_location = {N, position_array, successive_bits_array};
19:       bit_count=0;
20:       if (mode = Arbitrary(total_bits) ) then {
21:          for i in range(0 : sizeof(position_array)):
22:             if (current_position=position_array[i]) {
23:                do { Multiply(dummy);
24:                     Reduce(dummy);
25:                     } while(bit_count!= successive_bits_array[i]) }
26:       else if (mode = Cyclic(total_bits, N)) then {
27:          if (cycle_count != N;) then {
28:             reload tamper_proof_location = (N, position_array,
29:                                             successive_bits_array );
30:          else
31:             {refresh tamper_proof_location = (N, position_array,
32:                                             successive_bits_array );
33:       int N, cycle_count=0, bit_count; #N is selected by user
34:       for i in range(0 : sizeof(position_array)):
35:          if (current_position=position_array[i]) {
36:             do { Multiply(dummy);
37:                  Reduce(dummy);
38:                  tamper_proof_location++; }
39:                       while(bit_count!= successive_bits_array[i]) } }
40: end Victim Program;
```

the value of the size of the key. The *position_array, successive_bits_array* stores the values of the bit positions to inject dummy calls to and the successive bits to perturb, respectively. The values required for driving the *cyclic* mode are saved to a tamper-proof location that stores the current cycle count, along with the two arrays mentioned above. The Covert-Enigma and victim are synchronized using function calls. The arbitrary mode is shown in lines 20–25. The *arbitrary* mode injects the perturbations until the *bit_count* in the *successive_bits_array*. In our implementation, the Covert-Enigma only injects a dummy multiply followed by a dummy reduce to give the notion of a bit "1" instead of a bit "0"—as a square-reduce-multiply-reduce sequence corresponds to bit "1" being processed by the RSA. The *cyclic* mode is shown in lines 26–39. The *cyclic* mode operates similar to another mode. The major difference is that it does not keep injecting dummy operations with every new cycle of the victim application. To enable this, the Covert-Enigma accesses a tamper-proof location that stores the cycle count and the other two arrays mentioned previously. This helps to keep

injected perturbations in the victim's cache access patterns "static" for a user-selected number of cycles, specified by the value "*N*." If the victim has not completed the set number of cycles, the Covert-Enigma ensures that the same positions are selected to inject the dummy operations by reloading from the tamper-proof location. Otherwise, new random positions are generated. For the purpose of brevity, we limit the details of the attack, but interested readers can refer to [16] for details.

The attack code has been shown in Listing 3. The attack code is launched with the victim executing in parallel by the adversary to spy on the side-channel data. The probes 1, 2, and 3 are inserted by the victim in the first few lines of code, which the attacker knows are called iteratively by the victim. These probes from the attack's point of view are simply the addresses of the lines in the victim code. The attack code then flushes these probed lines and waits for time *t_wait* for the victim to execute. If the victim executed and accessed the flushed cache line, the attacker, upon reloading the line, would see it as a cache hit—since the data was available and fetched quickly. Else, the attacker sees a cache miss. The cache hit/miss decisions are based on the threshold[8] value (slightly varies from system to system), which was "120" cycles for our experimental setup.

Entropy Maximization: By the conventional entropy equation, $H = -\log(P_i)$, where P_i is the probability of bit i. As seen previously, Covert-Enigma increases randomization by injecting perturbations in the signal; hence, the probability that the attacker observes the correct/original key bit reduces dramatically. Hence, the lower the probability, the higher the entropy, which means more randomness in the retrieved information. Since a group of bits are selected to perturb the observed side-channel data, and the user can choose the position of these bits and their quantity, the total permutation of such sequence is huge if the attacker tries to observe the side-channel data after iterating the victim for thousands or even more number of times. With a 4096-bit key, the attacker would have to iterate it for ${}^{4096}p\ 2 = 16.77{*}10^6$ for 2 bits perturbed and ${}^{4096}p\ 6 = 4.7 * 10^{21}$ for 6 bits perturbed. Hence, by setting more number of bits to be cognitively perturbed or randomly choosing the number of bits to be cognitively perturbed, the user can render more resilience to SCAs, despite the attacker executing an attack a large number of times.

IV. Experimental Evaluation

A. Validating the Attack and Covert-Enigma

Experimental Setup: We tested the proposed Covert-Enigma[9] on system with Intel-i7 core running Ubuntu 18.04 LTS OS with 16-GB RAM and GnuPG's [28] RSA [27] implementation. Flush+Reload [16] attack code can be found at [31].

Validation of Attack: Here, we evaluate the efficacy of the proposed defense to mitigate side-channel leakage to dissuade the adversary from stealing sensitive data. The cache size (LLC) on our experimental setup was 2 MB, with 16-way cache associativity. The cache map is a representation after one iteration of the victim. We present cache access maps (part of a cache that is of interest to investigate) in Fig. 5 for scenarios where the victim is under attack and when our proposed Covert-Enigma shields the victim. The nuances of the colors shown in the figure demonstrate the relative accesses made to a particular location—darker shade signifies more frequent accesses. In comparison, a lighter shade signifies relatively less frequent accesses. As seen in Fig. 5(a) and (b), probed functions for RSA victim are highlighted. These locations correspond to the cache locations attacked by the adversary.

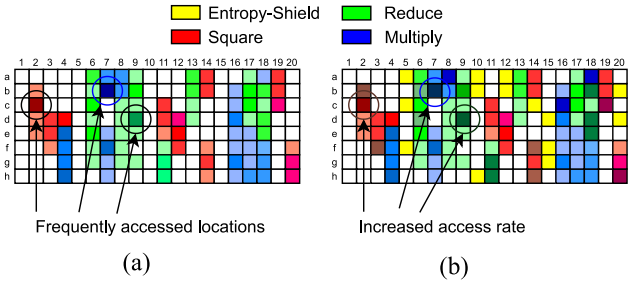


Fig. 5. (a) Cache access map for operations observed when the victim is under attack. (b) Cache access map for operations observed when Covert-Enigma makes cognitive calls to the probed cache lines.

Fig. 5(a) shows access to the cache made by the victim and the adversary. Fig. 5(b) shows the cache accesses made by the defense, adversary, and the victim, as the same cache is shared across. In Fig. 5(a) one can see tightly clustered dense regions of cache access. However, in Fig. 5(b), one can observe the dense areas spread across the whole map. Such a disaggregation leaves the attacker with more ambiguity. Further, some areas have shown an increase in access rate due to additional dummy operations introduced by Covert-Enigma. It is to be noted that Fig. 5 is a simplified illustration of cache accesses obtained from experiments to demonstrate the effectiveness of proposed Covert-Enigma.

B. Recovering Sensitive Data

We also evaluate the effectiveness of the proposed defense in terms of key extraction. In other words, we present the information regarding how many key-bits can be extracted by executing the RSA application under the Flush+Reload SCA with traditional randomization and proposed defense. We follow the procedure described in [16] and [17] for key extraction. We execute the attack on the victim and recover as many bits of the secret data as possible. As described in our threat model, we place our experiments in a real-world setting where the adversary does not have control over the victim and can only observe a part of the victim's execution. This is realistic as the victim only executes for encryption/decryption operations only when required. Hence, it is imperative to mention that the adversary initiates the attack during such instances and observes a portion of the victim's execution. The results of the key recovery are presented in Tables IV and VI. The colored

[8]The threshold is the probe access time in cycles.

[9]https://github.com/hartanonymous3512/Covert-Enigma.git

TABLE I
GROUP OF KEY BITS PERTURBED BY COVERT-ENIGMA

Group of key bits perturbed by Covert-Enigma - Arbitrary mode							
Attack	Victim	Key	Original Key	Victim seen key	Key seen by the adversary		
Flush+Reload	RSA	key_1	100100001110	100100001110	100111001110		
	RSA	key_2	010110000111	010110000111	010110111111		
Group of key bits perturbed by Covert-Enigma - Cyclic mode							
Attack	Victim	Key	Original Key	Victim seen key	Key seen by the adversary		
					Iteration 1	Iteration N^{th}	Iteration $(N+1)^{th}$
Flush+Reload	RSA	key_1	100100001110	100100001110	111100111110	111100111110	100111001111
	RSA	key_2	010110000111	010110000111	110111100111	110111100111	011110011111

TABLE II
SCA ON VICTIM PROTECTED BY TRADITIONAL RANDOMIZATION AND COVERT-ENIGMA. ATTACKER RECOVERED SECRET DATA

Key Size	1024	2048	3072	4096
Bits Recovered (traditional randomization) (%)	88.2	85.1	81.7	77.0
Bits Recovered (Covert-Enigma) (%)	60-70	53-68	48-62	40-52

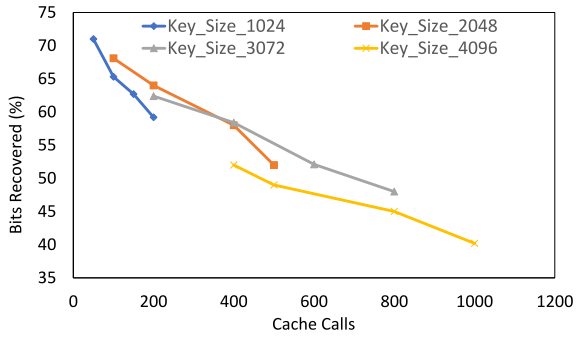


Fig. 6. Bits recovered under SCA shown with different dummy cache accesses and for different key sizes. The victim application is wrapped by Covert-Enigma. The number of bits recovered (secret) reduces with increase in dummy cache calls made by Covert-Enigma.

text highlights the cognitive perturbations that are injected. For Cyclic mode, the selected bit positions remain the same until "N"th round ($N = 25$ in our experiments), followed by new bit positions selected. A part of the observed key is shown for conciseness. We evaluate the defense under different key sizes for crypto-operation.

Table II shows the key extraction for different key sizes with traditional randomization defense. We implement a randomization strategy similar to that in [21]–[23]. We do not claim an accurate replication of the defense in [21]–[23], yet consider a similar approach as a baseline for comparing the proposed methodology versus a similar defense where the cache is accessed randomly to mislead the attacker. The works in [21]–[23] are based on randomization but require hardware or software stack changes. We have replicated them without software stack or hardware architecture changes. Hence, to establish a baseline, we consider the above-mentioned works as a randomization-based defense generically, and compared our work with a similar version. Fig. 6 presents the results for bit recovery with Covert-Enigma for different key sizes and dummy cache accesses. The number of calls made are for one complete execution of the victim. The results in Table II demonstrate that with a defense strategy like that presented by the work in [21]–[23], the recovery of the secret key/data ranges from 88% to 77%. We have presented the recovery rate with when the victim is protected by Covert-Enigma in Fig. 6. We have compared Table II with Fig. 6 indicating that with our proposed defense, the recovery rate reduces to 72%–59% for key size of 1024, 68%–52% for 2048, 62%–48% for 3072, and 52%–40% for a key size of 4096.

TABLE III
PERCENTAGE DIFFERENCE COMPARISON OF VICTIM OPERATIONS WITH AND WITHOUT COVERT-ENIGMA

Amount of injected perturbations	100	300	500	600
Difference observed with perturbations(%)	8	26	48	55

For the *arbitrary* mode, we obtain similar results as in Fig. 6. The cyclic mode's advantage is in scenarios where the user happens to use a crypto operation that uses the same key for deobfuscating different encrypted files on the system and where the adversary can obtain information from multiple executions of the victim. Another evaluation technique we have used in this work is by comparing the observed traces. The spy program is made to print the operations' sequence while the probed locations—probed functions square, reduce, and multiply—are accessed by the victim. These sequences are compared against the victim's operations under Covert-Enigma. Table III presents the number of perturbations (additional cache calls) injected for a 1024-bit key. The table reports the differences seen in percentage. For instance, an 8% difference is observed while comparing the victim's operations without Covert-Enigma and with Covert-Enigma. Theoretically, 8% should have been 10% for 100 additional calls in a 1024-wide key. But, as explained previously, the spy cannot see successive cache operations, and hence, some operations are not observed, as the probe scan time is less than the cache access time.

C. Covert-Enigma With Flush+Reload Attack

We have chosen the Flush+Reload and Flush+Flush attack spying on RSA-RSA and DSA-Elgamal encryption algorithms with a secret key of 4096-bits, as implemented in the GnuPG. We will also present the outcome with different modes of operation—*Arbitrary* and *Cyclic*. We verified our proposed Covert-Enigma by examining the perturbations injected both on the victim and spy end. Fig. 4 presents a pattern of the sequence of operations plotted against time slots versus the probe time as seen by the attacker/victim. Fig. 4(a) shows the secret information observed by the victim and the attacker without the Covert-Enigma. Every square-modulo operation not followed by Multiply is translated as bit "0," and

TABLE IV
GROUP OF KEY BITS PERTURBED BY COVERT-ENIGMA–ARBITRARY MODE

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	100111001110
	DSA-Elgamal	key_2	010110000111	010110000111	010111111111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111011100110
	DSA-Elgamal	key_4	100000110011	100000110011	100111110011

TABLE V
SINGLE KEY BIT PERTURBED BY COVERT-ENIGMA–ARBITRARY MODE

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	100110001110
	DSA-Elgamal	key_2	010110000111	010110000111	010110100111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111001100110
	DSA-Elgamal	key_4	100000110011	100000110011	100001110011

every square-modulo-multiply-modulo operation as bit "1," as in [16]. In this case, the victim and the attacker both see the same information—the spy observes the channel's leaked information. Fig. 4(b) shows the sequence of operations when the victim is being protected by the Covert-Enigma in the arbitrary mode—the victim observes the key as "10010000," the original key, while the attacker sees it as "10011100" since some of the "0" bits are flipped to bit "1." These perturbations are induced irrespective of the key, as shown in Table IV with cognitively selected zeros converted to ones for the *key_1* for the RSA-RSA type encryption—victim sees the key as "100100001110," the attacker observes it as "100111001110." One needs to note that in Fig. 4 all the bits are not shown to avoid congestion in the figure, and also, it was not possible to show all of the 4096-bits. Also, for the Tables IV and VI, a part of the large key has been shown to demonstrate the perturbation rather than the actual position in itself.

Similarly, for *key_2*, DSA-Elgamal type, some other random bits are perturbed, and the attacker observes a different pattern. For the *Cyclic* mode, as shown in Fig. 4 and Table VI, the perturbed key remains the same for "$N = 25$" iterations, post which other random bits are perturbed in the sequence that begins with $(N + 1)$th iteration, which stays static until the end of the cycle which is $(N + N)$th iteration. As seen from Fig. 4, the attacker observes the sequence as "11110000" which remains the same until the end of iteration "N," post which it changes to "10010110" and the results for the same can be confirmed from Table VI.

Tables V and VII demonstrate the results where randomly chosen (similar to the group perturbations) single bits are flipped/perturbed. The *successive_bits_array* value can be modified to choose single bit perturbation instead of grouped perturbation, where successive bits are perturbed. From Table V, the victim sees the value as "100100001110" while the attacker observes it as "10011000110" for RSA type. Similarly, it can be seen from Table VII how the observation is affected using the *Cyclic* mode. The single-bit perturbations can be chosen to perturb bits along the entire sequence of operations of cache accesses. The user can choose a single bit versus a group of bits considering the security-overhead tradeoff.

D. Covert-Enigma With Flush+Flush Attack

We have evaluated our Covert-Enigma against Flush+Flush, whose key extraction results are presented in Tables IV and VI for both the modes. Similar to the Flush+Reload, the induced perturbations can deceive the spy in both *arbitrary* and the *Cyclic* modes. For instance, for the RSA type keys, in the *arbitrary* mode, the key gets translated from "111000100110" to "111011100110" whereas for the Cyclic mode it is observed as "111011111110" and "111000111110" during iteration-1 and iteration $(N+1)$th, respectively. For our proposed defense to work even for Flush+Flush, it needs to ensure that the lines of code within the square, modulo, or multiply functions are cached, and only then the attacker can flush a cache line within the code and consider that the encryption must have accessed the function/operation. Tables V and VII present results for single bit perturbations for both the modes for Flush+Flush.

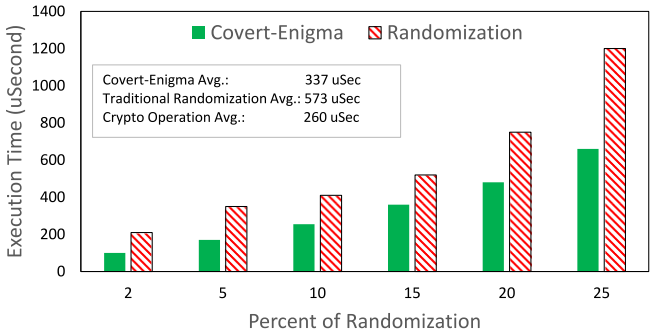


Fig. 7. Overhead analysis for 4096-bit key with different amounts of randomization. Overhead compared with a close replication of random cache policy similar to [21]–[23].

E. Summary of the Implemented Results

Tables IV–VII are ideal cases because while executing them on our machine we reduced the number of background activity. But, in actual scenarios, the OS and other application activity will generate noise in the cache, making the attack more difficult. The attacker might not be able to see the key bits in consecutive order. Hence, as the keys seen by the attacker will be different every time and with such randomness, explained in detail in Section III-F it is very difficult for the attacker to retrieve the key knowing the fact that executing SCAs successfully is nontrivial when it comes to retrieving secret keys amid operating system noise and various cache operations. Our Covert-Enigma enhances security, but there is no single/unified mechanism to evaluate all the corner cases. A single solution does not address all the problems. Our proposed solution holds for the threat model described previously.

F. Overhead Analysis

The overhead analysis graph is shown in Fig. 7. The figure compares the execution times of traditional randomization technique similar to [21]–[23] and Covert-Enigma. We consider the arbitrary mode for presenting the results. The trend is shown for different amounts of randomization added (along the x-axis) and the execution time in microseconds (along y-axis). The average execution times across different percent of randomization is shown in Fig. 7. The trend clearly explains that with our defense, the overhead is 50% less than a randomization technique that tries to insert random calls for every bit of the secret key. Our proposed defense inserts the calls cognitively, hence incurs less overhead—cache is accessed less

TABLE VI
GROUP OF KEY BITS PERTURBED BY COVERT-ENIGMA—CYCLIC MODE

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker		
					Iteration 1	Iteration N^{th}	Iteration $(N+1)^{th}$
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	111100111110	111100111110	100111001111
	DSA-Elgamal	key_2	010110000111	010110000111	101111100111	101111100111	011110011111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111011111110	111011111110	111000111110
	DSA-Elgamal	key_4	100000110011	100000110011	100110111111	100110111111	111000110011

TABLE VII
SINGLE KEY BIT PERTURBED BY COVERT-ENIGMA—CYCLIC MODE

Attack Type	Encryption	Key	Original Key	Victim seen key	Key seen by the attacker		
					Iteration 1	Iteration N^{th}	Iteration $(N+1)^{th}$
Flush+Reload	RSA-RSA	key_1	100100001110	100100001110	101100011110	101100011110	100101001111
	DSA-Elgamal	key_2	010110000111	010110000111	101101000111	101101000111	011110010111
Flush+Flush	RSA-RSA	key_3	111000100110	111000100110	111010110110	111010110110	111000101110
	DSA-Elgamal	key_4	100000110011	100000110011	100100111011	100100111011	101000110011

frequently than traditional randomization techniques. We see this tradeoff as a beam scale balance that weighs security and performance (in terms of execution time/cycles) on each of its scale pans. The user can determine the amount of perturbations by analyzing the overhead-security tradeoff. Again, it is to be noted that the traditional randomization we compare our proposed methodology is not an exact reproduction of the work in [21]–[23], but it is similar in a manner that we allow injecting random perturbations in the cache in software; no hardware modifications are required.

How the cognitive perturbations differ from traditional randomization is explained further referring to Fig. 7. If each bit of the key is perturbed, meaning the cache is accessed in a dummy fashion (randomly), the overhead is significantly higher. From our experiments, if X is the base execution time for the victim, then 2.19X is the overhead with traditional randomization, while it is 1.29X with Covert-Enigma's arbitrary mode. All the overheads are averaged for simplicity. A 4096-bit key is used for crypto operations, and by varying the number of cognitive perturbations, higher security can be offered. This comes at the expense of some overhead. With 10% injected perturbations, the overhead is 25% less with Covert-Enigma compared to randomization only. With 25% injected perturbations, we observe a 50% less overhead against the randomization only method as mentioned above. Hence, perturbing each bit is not a solution owing to large infeasible overhead. With traditional randomization, the overhead can be feasible, but the attack can break the defense much earlier than it can when Covert-Enigma wraps the victim. Moreover, the overhead of Covert-Enigma is less than the other technique.

V. ATTACK PHASE AND THE COVERT-ENIGMA: CASE-STUDY

In this section, we will briefly discuss the motivation supporting the proposed Covert-Enigma which is presented as a case study.

Fig. 8 shows different scenarios of the victim's access to the cache memory along with the attacker's access and how the attacker exploits this information in deducing the secret key.

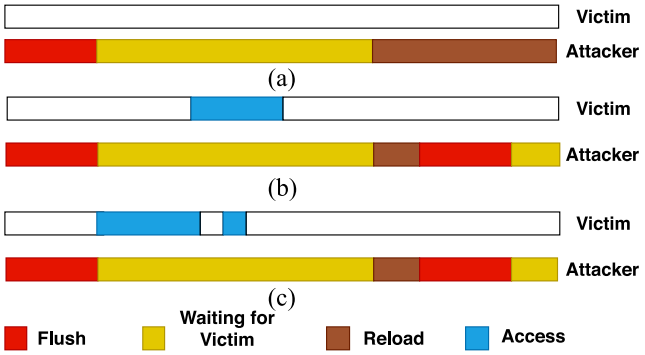


Fig. 8. Timing diagram depicting different scenarios where a victim and/or an attacker may access the cache. (a) Victim does not access. (b) Attack with victim access. (c) Victim multiaccess.

Fig. 8(a) shows a scenario where the attacker tries to flush the victim's data, then waiting for a predefined time before reloading the same data. As can be seen, since the victim did not access the data, the attacker experiences a cache-miss when it tries to reload the data, which is discarded. Fig. 8(b) visually describes the victim's access while the attacker was waiting for the victim to execute. Since the victim accessed the data, the attacker experiences a cache-hit during the reloading phase, thus deducing the data accessed by the victim. Fig. 8(c) presents a scenario when the victim accesses the cache multiple times within the same "wait" window of the attacker. But the attacker can spy on only the recent chunk of data accessed by the victim, and it will never know what locations the victim accessed preceding the recent access. In summary, irrespective of the scenario, the attacker can still spy on the location accessed by the victim.

In addition to this, referring to Fig. 9, we can get an idea of how the attacker spies on the crypto application while executing secure critical operations. As explained in Section II, the GnuPG uses different operations to encode/decode user data or secret data where the sequence of these operations can leak the secret data shown in Fig. 9. Part (a) presents a sequence of operations that decodes to "10001" as discussed previously. If only some noise could be added to these sequence traces, the SCAs could be thwarted with little effort. Let us consider a

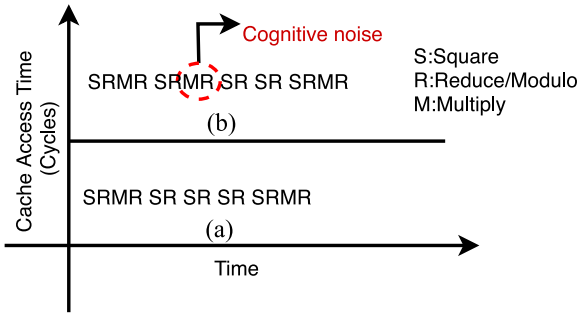


Fig. 9. (a) Sequence of operations in a crypto system that potentially leaks secret data. (b) Cognitive noise injected in the victim's covert channel data to protect the information while tricking the attacker.

defense mechanism that adds noise to the traces observed by the attacker. This might dissuade the attacker from decoding the secret data as due to noise, deducing the key would seem difficult. Still, in case of a persistent attack on the system, the attacker can break the defense wall by observing a large sample of the observed data and filtering the noise. Hence, merely adding noise to the operations' sequence will not suffice and is not a robust solution. We introduced cognitive noise to the sequence of operations that looks legit to the attacker yet leads to an incorrect deduction of the secret key. Fig. 9(b) shows the same sequence of operations as part (a) but with intelligently crafted noise injected in the sequence. This crafted noise concerning the sequences makes sense to the attacker. As can be seen, the multiply and reduce operations are dummy called succeeding the square and reduce operations (original operations called by the victim), which, when observed by the attacker, will translate to bit-1 instead of bit-0 which tricks the attacker. Since the injected operations are dummy, they do not harm the victim's crypto operations. This has been our motivation behind the proposed work. We have discussed different modes of operation of the proposed method to render the defense more robust and resilient to attacks.

Extending to Other Victims: It needs to be noted that our proposed Covert-Enigma could be extended to any victim that repeatedly calls for lines of code, where the sequence of accesses to the cache is important. For example, suppose the victim application is Advances Encryption Standard (AES). In that case, the user can decide to randomize the cache accesses AES makes for reading tables used for the crypto-operations. The adversary times the access to the locations of the tables to conclude which ones were accessed recently. The user can employ Covert-Enigma to introduce cognitive perturbations, which seems legit (cache access), but the sequence is scrambled to camouflage the sequence of cache operations. Thus, it becomes possible to hide the real accesses made by AES, yet leading the attacker to consider cache accesses made by Covert-Enigma.

VI. State of the Art

In order to secure the hardware systems against cache-side channel attacks, various defense techniques have been proposed that use different strategies. To address the challenges of cache-targeted SCAs, techniques, such as static cache partitioning [26], [32], partition locked cache [25], nonmonopolizable (nomo) cache architectures [33] and other defenses [26], [32], [34], [35] are proposed. These techniques can tremendously reduce the interference between the attacker and victim's memory access, thereby providing a better defense. However, adopting such techniques require alterations in the cache design which may not be feasible [26]. To overcome such limitations, techniques such as cache-partitioning, randomization of cache architectures are introduced. Conventional fully associative cache is one of the preliminary randomization-based cache, where 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 [26]. In a similar way, random permutation cache [25], newcache [22], [36], random fill cache [21], and random eviction cache [26] 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 incurs performance penalties. We discuss the most relevant and prominent ones in this section.

A. Cache Partitioning-Based Defenses

These defenses are based on eliminating the cache interference between the running processes. This way, running processes cannot snoop on each others' cache activity. He and Lee [26] proposed to protect sensitive cache access (e.g., coming from sensitive data/operation) by reserving dedicated cache sets for those sensitive accesses. Thus, the sensitive cache access will always index to the dedicated sets and all other cache access, including cache access from other running processes or threads will index to the rest of the cache sets. As the mapping from memory to a cache set involves the physical memory address, the proposed solution utilizes the operating system to organize physical memory into nonoverlapping cache set groups, also called colors, and to enforce isolation policy on these groups. However, this approach leads to inefficient resource utilization and hardware overheads. Vladimir Kiriansky proposed dynamically allocated way guard (DAWG) [37], a generic mechanism for secure way partitioning of set associative structures including memory caches. DAWG endows a set associative structure with a notion of protection domains to provide strong isolation. When applied to a cache, unlike 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 enforces isolation of exclusive protection domains among cache tags and replacement metadata, as long as: 1) victim selection is restricted to the ways allocated to the protection domain (an invariant maintained by system software) and 2) metadata updates as a result of an access in one domain do not affect victim selection in another domain (are requirement on DAWG's cache replacement policy). DAWG protects against attacks that rely on a cache state-based channel, which are commonly referred to

as cache-timing attacks, on speculative execution processors with reasonable overheads. The same policies can be applied to any set associative structure, e.g., TLB or branch history tables. DAWG requires additional techniques to block exfiltration channels different from the cache channel. Nonetheless, cache partitioning-based defenses lead to hardware as well as performance overhead.

SGX Enclave Protection: Furthermore, Oleksenko *et al.* proposed *Varys* [38], 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. This execution environment ensures that neither time-sliced nor concurrent cache timing attacks can succeed. Due to the lack of appropriate hardware support in today's SGX hardware, Varys remains vulnerable to timing attacks on LLC. This article also proposes a set of minor hardware extensions that hold the potential to extend Varys' security guarantees to L3 cache and further improve its performance. But the downside is it requires the application to monitor the SSA (SGX State Save Area) value, thus increasing the overhead and it introduces a window of vulnerability.

3-D Integration: Bao and Srivastava [39] showed that 3-D integration also offers inherent security benefits and enables many new defense mechanisms that would not be practical in 2-D. The work is compatible with the ongoing trend of transition from 2-D to 3-D and enables designers to take security into account when designing future cache using 3-D integration technology. Experimental results show that using their cache design, the side-channel leakage is significantly reduced while still achieving performance gains over a conventional 2-D system.

Intel Cache Allocation Technology: Dong *et al.* present in [40] defenses against page table and LLC SCAs launched by a compromised OS kernel. They prototyped the solution in a system call Apparition, building on an optimized version of Virtual Ghost. To thwart LLC SCAs, it leverage Intel's CAT in concert with techniques that prevent physical memory sharing. Apparition's control over privileged hardware state can partition the LLC to defeat cache SCAs. Their defense combines Intel's CAT feature (which cannot securely partition the cache by itself) with existing memory protections from Virtual Ghost to prevent applications from sharing cache lines with other applications or the OS kernel. Similarly, authors in the paper [41] propose to utilize CAT in Intel processors to provide a system-level protection to defend against SCAs on shared LLC. CAT is a way-based h/w cache-partitioning mechanism for enforcing quality to LLC occupancy. "CATalyst" uses CAT to partition the LLC securely into a hybrid hardware–software managed cache to defend against SCAs.

B. Randomization-Based Defenses

To overcome limitations of hardware oriented approaches, randomizing the memory access is introduced in [25], thus, making the attack much harder, even impossible. For instance, He and Lee [26] used 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. It can also use software-based compiler assisted approach to transform applications to randomize its memory access patterns.

Control Flow Randomization: Crane *et al.* [23] explored software diversity as a defense against SCAs by dynamically and systematically randomizing the control flow of programs. Existing software diversity techniques transform each program trace identically. This diversity-based technique instead transforms programs to make each program trace unique. This approach offers probabilistic protection against both online and off-line SCAs. It extends previous, mostly static software diversification approaches by dynamically randomizing the control flow of the program while it is running. Rather than statically executing a single variant each time a program unit is executed, they created a program consisting of replicated code fragments with randomized control flow to switch between alternative code replicas at runtime.

C. Detection-Based Defenses

Computational Anomaly Detection: Work in [11] give an overview of the attacks on hardware, including the SCAs, and describes the panacea to thwart such attacks and secure the hardware. Shukla *et al.* [3], [10] have proposed a unique methodology in detecting even stealthy malwares with hardware performance counters (HPCs) and image processing along with RNN-based ML to assist the detection process.

SCA Detection in the Cloud: Zhang *et al.* [42] presented an architecture where cores (processors) equipped with specialized signature detection techniques are employed to detect SCAs based on the HPCs these attacks generate in a system. Kim *et al.* presented in [43] a system-level protection mechanism against cache-based SCAs in the cloud named as "STEALTHMEM." This mechanism protects cache from unauthorized access by managing a set of locked cache lines per core that are never evicted from the cache. Thus, any virtual machine (VM) can hide its sensitive information from others. Work in [44] presents "StopWatch" a system that defend against SCAs in cloud environment by triplicating each cloud-resident VM and using the timing of the I/O events at the replicas to determine the timings observed by each replica or the attacker. Shi *et al.* in their work in [45] propose a technique, they name as dynamic cache coloring, which notifies the VM when an application is executing secure-sensitive operations to swap the associated data to a safe an isolated cache line to protect the same against SCA attack by limiting its access. They presented the technique for multitenant-based cloud environment.

VII. Conclusion

In this work, we propose Covert-Enigma which can protect applications from timing-based SCAs by injecting cognitive perturbations and reducing useful information leaked on the

covert channel. The proposed technique is equipped with two modes—arbitrary and cyclic—to render flexibility to the user in terms of robustness. We verified the efficacy of Covert-Enigma on Flush+Reload, and Flush+Flush attack on RSA and Elgamal encryption methods. The results demonstrate that the proposed technique can be utilized to secure victim applications without needing modifications to hardware or the operating system. Our proposed technique's average overhead is 10% compared to without the defense in place due to additional cache accesses. Our approach can easily be modified to suit a variety of victim applications.

REFERENCES

[1] A. Dhavlle, R. Mehta, S. Rafatirad, H. Homayoun, and S. M. P. Dinakarrao, "Entropy-shield: Side-channel entropy maximization for timing-based side-channel attacks," in *Proc. 21st Int. Symp. Qual. Electron. Design (ISQED)*, 2020, pp. 161–166.

[2] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3332–3344, Nov. 2016.

[3] S. Shukla, G. Kolhe, S. M. P. Dinakarrao, and S. Rafatirad, "Stealthy malware detection using RNN-based automated localized feature extraction and classifier," in *Proc. IEEE Int. Conf. Tools Artif. Intell. (ICTAI)*, 2019, pp. 590–597.

[4] S. Shukla, G. Kolhe, P. D. S. Manoj, and S. Rafatirad, "RNN-based classifier to detect stealthy malware using localized features and complex symbolic sequence," in *Proc. Int. Conf. Mach. Learn. Appl.*, 2019, pp. 406–409.

[5] G. Kolhe *et al.*, "Security and complexity analysis of LUT-based obfuscation: From blueprint to reality," in *Proc. Int. Conf. Comput.-Aided Design*, 2019, pp. 1–8.

[6] Z. Chen *et al.*, "Estimating the circuit deobfuscating runtime based on graph deep learning," in *Proc. Design Autom. Test Europe Conf. (DATE)*, 2020, pp. 358–363.

[7] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "EnsembleHMD: Accurate hardware malware detectors with specialized ensemble classifiers," *IEEE Trans. Depend. Secure Comput.*, vol. 17, no. 3, pp. 620–633, May/Jun. 2020.

[8] H. Sayadi *et al.*, "2SMART: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *Proc. DATE*, 2019, pp. 728–733.

[9] S. M. P. Dinakarrao *et al.*, "Adversarial attack on microarchitectural events based malware detectors," in *Proc. Design Autom. Conf.*, 2019, p. 164.

[10] S. Shukla, G. Kolhe, P. D. S. Manoj, and S. Rafatirad, "Microarchitectural events and image processing-based hybrid approach for robust malware detection: Work-in-progress," in *Proc. Embedded Syst. Week*, 2019, pp. 1–10.

[11] F. Brasser *et al.*, "Advances and throwbacks in hardware-assisted security: Special session," in *Proc. Int. Conf. Compilers Architect. Synth. Embedded Syst.*, 2018, pp. 1–10.

[12] A. Dhavlle, S. Bhat, S. Rafatirad, H. Homayoun, and P. D. S. Manoj, "Work-in-progress: Sequence-crafter: Side-channel entropy minimization to thwart timing-based side-channel attacks," in *Proc. Int. Conf. Compliers Architect. Synth. Embedded Syst. (CASES)*, 2019, pp. 1–9.

[13] *Symantec*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.symantec.com/security-center/threat-report

[14] *Lp.Skyboxsecurity*. Accessed: Oct. 10, 2020. [Online]. Available: https://lp.skyboxsecurity.com/rs/440-MPQ-510/images/Skybox_Report_Vulnerability_and_Threat_Trends_2019.pdf

[15] *Sonicwall*. Accessed: Oct. 10, 2020. [Online]. Available: https://blog.sonicwall.com/en-us/2019/03/new-spoiler-side-channel-attack-threatens-processors-mitigated-by-sonicwall-rtdmi/

[16] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, L3 cache side-channel attack," in *Proc. USENIX Conf. Security*, 2014, pp. 719–732.

[17] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proc. Int. Conf. Detect. Intrusions Malware Vulnerability Assessment*, 2016, pp. 279–299.

[18] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2006, pp. 201–205.

[19] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. ACM Conf. CCS*, 2012, pp. 305–316.

[20] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security Privacy*, vol. 8, no. 6, pp. 40–47, Nov./Dec. 2010.

[21] F. Liu and R. B. Lee, "Random fill cache architecture," in *Proc. IEEE/ACM Int. Symp. Microarchitect.*, 2014, pp. 203–215.

[22] F. Liu, H. Wu, K. Mai, and R. B. Lee, "NewCache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[23] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2015, pp. 1–5.

[24] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proc. ACM Workshop Security Artif. Intell.*, 2011, pp. 97–112.

[25] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. ISCA*, 2007, pp. 494–505.

[26] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proc. IEEE/ACM*, 2017, pp. 341–353.

[27] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[28] *gnupg*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.gnupg.org/

[29] *Software*. Accessed: Oct. 10, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

[30] *man7*. Accessed: Oct. 10, 2020. [Online]. Available: http://man7.org/linux/man-pages/man4/random.4.html

[31] T. Hornby. (2016). *Flush+Reload Attack*. Accessed: Jul. 1, 2019. [Online]. Available: https://github.com/defuse/flush-reload-attacks

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

[33] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–35, Jan. 2012.

[34] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities," in *Proc. IACR Cryptol. ePrint Archive*, Jan. 2006, p. 52.

[35] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proc. ACM Workshop Comput. Security Architect.*, 2008, pp. 25–34.

[36] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *Proc. MICRO*, 2008, pp. 83–93.

[37] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. MICRO*, 2018, pp. 974–987.

[38] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *Proc. USENIX*, 2018, pp. 227–240.

[39] C. Bao and A. Srivastava, "3D integration: New opportunities in defense against cache-timing side-channel attacks," in *Proc. IEEE ICCD*, 2015, pp. 273–280.

[40] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *Proc. USENIX Security Symp.*, 2018, pp. 1441–1458.

[41] F. Liu *et al.*, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect. (HPCA)*, Mar. 2016, pp. 406–418.

[42] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Proc. RAID*, 2016, pp. 118–140.

[43] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in presented at the 21st USENIX Security Symp. (USENIX Security), 2012.

[44] P. Li, D. Gao, and M. K. Reiter, "StopWatch: A cloud architecture for timing channel mitigation," *ACM Trans. Inf. Syst. Security*, vol. 17, no. 2, pp. 1–28, Nov. 2014.

[45] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. IEEE/IFIP 41st Int. Conf. Depend. Syst. Netw. Workshops*, 2011, pp. 194–199.

Abhijitt Dhavlle (Graduate Student Member, IEEE) received the master's degree in computer engineering from George Mason University, Fairfax, VA, USA, where he is currently pursuing the Ph.D. degree.

He is currently serving as a Graduate Research Assistant with George Mason University. His research interests are microarchitectural attacks, physical side-channel attacks, and hardware-based malware detectors.

Setareh Rafatirad (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science from the University of California at Irvine, Irvine, CA, USA, in 2009 and 2012, respectively.

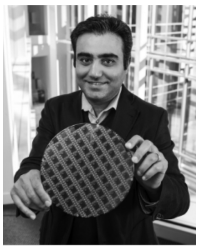
She is an Associate Professor with the Department of Information Sciences and Technology, George Mason University, Fairfax, VA, USA. She received research funding from government agencies, including NSF, DARPA, and AFRL for major projects. She is currently actively supervising multiple research projects focused on applying ML and deep learning techniques on different domains, including house price prediction, malware detection, and emerging big data application benchmarking and characterization on heterogeneous architectures. Her research interest covers several areas, including big data analytics, data mining, knowledge discovery and knowledge representation, IoT security, and applied machine learning.

Dr. Rafatirad received the ICDM 2019 Best Peper Award (9% acceptance rate) and was nominated for ICCAD 2019 Best Paper Award.

Khaled Khasawneh (Member, IEEE) received the B.Sc. degree in computer engineering from the Jordan University of Science and Technology, Irbid, Jordan, in 2012, the M.S. degree in computer science from SUNY Binghamton, Binghamton, NY, USA, in 2014, and the Ph.D. degree from the University of California at Riverside, Riverside, CA, USA, in 2019.

In Fall 2019, he joined George Mason University, Fairfax, VA, USA. He has previously interned with Community Integrity Team, Facebook, Menlo Park, CA, USA. His research interest is in computer systems and specifically in computer architecture support for security, malware detection, adversarial machine learning, and side-channels attacks.

Dr. Khasawneh his 2018 paper in USENIX Workshop on Offensive Technologies Received the Best Paper Award. He is the recipient of the Dissertation Year Program Award and the International Student Recognition Award from UCR. Several of his contributions have been reported on by technical news outlets, including ZDNet, Digital Trends, Tech Republic, the Register, Threat Post, Beta News, and Bleeping Computer.



Houman Homayoun (Senior Member, IEEE) received the B.S. degree in electrical engineering from the Sharif University of Technology, Tehran, Iran, in 2003, the M.S. degree in computer engineering from the University of Victoria, Victoria, BC, Canada, in 2005, and the Ph.D. degree in computer science from the University of California at Irvine, Irvine, CA, USA, in 2010.

He was an Associate Professor with the Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA, USA. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA, USA. From 2010 to 2012, he spent two years with the University of California at San Diego, San Diego, CA, USA, as an NSF Computing Innovation Fellow Awarded by the CRA-CCC. He is currently the Director with the Accelerated, Secure, and Energy-Efficient Computing Laboratory, University of California at Davis, Davis, CA, USA. He has published over 100 technical papers in the prestigious conferences and journals on the subject and directed over $8M in research funding from NSF, DARPA, AFRL, NIST, and various industrial sponsors. His conducts research in hardware security and trust, data-intensive computing, and heterogeneous computing.

Dr. Homayoun was a recipient of the Four-Year University of California, Irvine, Computer Science Department Chair Fellowship. He received several best paper awards and nominations in various conferences, including GLSVLSI in 2016, ICCAD in 2019, and ICDM in 2019. He served as Member of Advisory Committee, Cybersecurity Research and Technology Commercialization working group in the Commonwealth of Virginia in 2018. Since 2017, he has been serving as an Associate Editor of IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION. He was the Technical Program Co-Chair of GLSVLSI 2018 and the General Chair of 2019 conference.

Sai Manoj Pudukotai Dinakarrao (Member, IEEE) received the master's degree in information technology from the International Institute of Information Technology Bangalore, Bangalore, India, in 2012, and the Ph.D. degree in electrical and electronics engineering from Nanyang Technological University, Singapore, in 2015.

He is an Assistant Professor with George Mason University (GMU), Fairfax, VA, USA. Prior joining to GMU as an Assistant Professor, he served as Research Assistant Professor and a Postdoctoral Research Fellow with GMU and was a Postdoctoral Research Scientist with the System-on-Chip group, Institute of Computer Technology, Vienna University of Technology, Wien, Austria. His research interests include on-chip hardware security, neuromorphic computing, adversarial machine learning, self-aware SoC design, image processing and time-series analysis, emerging memory devices, and heterogeneous integration techniques.

Dr. Dinakarrao won best paper award in International Conference on Data Mining in 2019, and his works were nominated for best paper award in prestigious conferences, such as Design Automation & Test in Europe in 2018, the International Conference on Consumer Electronics in 2020, and won Xilinx open hardware contest in 2017 (student category). He is the recipient of the "A. Richard Newton Young Research Fellow" award in Design Automation Conference in 2013.