

# On the Trade-Offs in Oblivious Execution Techniques

Shruti Tople<sup>(✉)</sup> and Prateek Saxena

National University of Singapore, Singapore, Singapore  
{shruti90,prateeks}@comp.nus.edu.sg

**Abstract.** To enable privacy-preserving computation on encrypted data, a class of techniques for input-oblivious execution have surfaced. The property of input-oblivious execution guarantees that an adversary observing the interaction of a program with the underlying system learns nothing about the sensitive input. To highlight the importance of oblivious execution, we demonstrate a concrete practical attack—called a logic-reuse attack—that leaks every byte of encrypted input if oblivious techniques are not used. Next, we study the efficacy of oblivious execution techniques and understand their limitations from a practical perspective. We manually transform 30 common Linux utilities by applying known oblivious execution techniques. As a positive result, we show that 6 utilities perform input-oblivious execution without modification, 11 utilities can be transformed with  $O(1)$  performance overhead and 11 other show  $O(N)$  overhead. As a negative result, we show that theoretical limitations of oblivious execution techniques do manifest in 2 real applications in our case studies incurring a performance cost of  $O(2^N)$  over non-oblivious execution.

## 1 Introduction

Many emerging techniques provide privacy preserving computation on encrypted data. These techniques can be categorized into two lines of work—secure computation and enclaved execution. Secure computation techniques enable operations on encrypted data without decrypting them. Examples of such techniques include fully homomorphic encryption [24–26], partially homomorphic encryption [20, 31, 50, 52, 61], garbled circuits [34, 36, 68] and so on. A second line of research uses hardware-isolation mechanisms provided by Intel SGX [48], TPM [5], Intel TXT [4], ARM Trustzones [7, 44]. Systems such as Haven [10], XOM [60], Flicker [47] use these mechanisms to provide *enclaved execution*. In enclaved execution the application runs in a hardware-isolated environment in the presence of an untrusted operating system. The sensitive data is decrypted only in the hardware-isolated environment and the computation result is encrypted before it exits the enclaved execution. Enclaved execution can be achieved via hypervisor-based mechanisms as well (cf. OverShadow [14], Inktag [32]).

One fundamental challenge in privacy preserving computation is to make the program execution *input-oblivious*. Input-oblivious execution guarantees that the

execution profile of a program observed by an adversary reveals nothing about the sensitive input. This challenge goes beyond the mechanism of enabling individual operations on encrypted inputs, whether done in enclaved execution environments or via cryptographic techniques for secure computation. In concept, it is easy to show that making all programs oblivious may be undecidable; such a result is neither surprising nor particularly interesting to practice. We study this problem from a practical perspective—whether it is feasible to make existing commodity applications execute obliviously without unreasonable loss in performance. If so, to what extent is this feasible and whether any theoretical limitations manifest themselves in relevant applications.

We explain the problem conceptually, considering various channels of leakage in the scenario of enclaved execution. To highlight the importance of oblivious execution, we show that enclaved execution is highly vulnerable to leakage of sensitive data via a concrete attack—called a logic-reuse attack. Specifically, we show that chaining execution of commonly used utilities can leak every byte of an encrypted input to the enclaved application.

Next, we study how existing oblivious execution techniques such as padding of dummy instructions [46], hiding message length [19] or hiding address accesses using Oblivious RAM [28] proposed in different contexts can be used to block the leakage in enclaved execution. Our work explains the symmetry among these lines of research, systematizing their capabilities and explaining the limits of these techniques in practical applications. Specifically, we manually transform 30 applications from the standard CoreUtils package available on Linux operating system. As a positive result, we show that 6 utilities perform input-oblivious execution without modification, 11 utilities can be transformed with  $O(1)$  performance overhead and 11 other show linear performance overhead of  $O(N)$ .

As a negative result, we show that theoretical limitations of oblivious execution techniques do manifest in 2 utilities which incur an exponential performance overhead of  $O(2^N)$ . Of course, they can be made oblivious conceptually, since everything on a digital computer is finite—in practice, this is hard to do without prohibitive loss in performance.

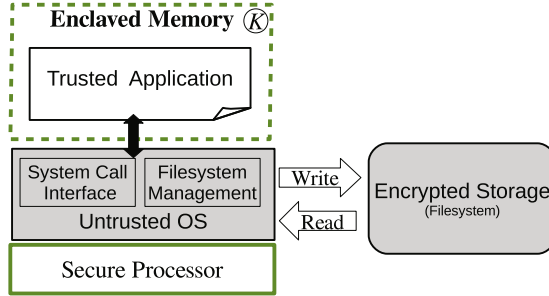
**Contribution.** We summarize our contributions as follows:

- 1. *Logic-reuse attack:* We demonstrate a concrete attack in the enclaved execution setting that leaks every byte of encrypted input by chaining execution of common applications.
- 2. *Systematization of oblivious execution techniques:* We systematize existing defenses for oblivious execution and show new limitations for enclaved execution of practical applications.
- 3. *Study of practical applications:* To study an empirical datapoint, we manually transform 30 applications from CoreUtils package to make them input-oblivious using existing defenses and find that 28 applications can be transformed with acceptable overhead. The limitations of oblivious execution techniques manifest in 2 applications which cannot be transformed without prohibitive loss in performance.

## 2 The Problem

**Baseline Setting.** Various existing solutions such as OverShadow [14], SecureMe [15], Inktag [32], SGX [48], Haven [10] and Panoply [57] support enclaved execution of applications. Here, the OS is untrusted whereas the underlying processor is trusted and secure. The file system is encrypted under a secret key  $K$  to protect the data on the untrusted storage. The trusted application executes in an enclaved memory which is inaccessible to the untrusted OS. The secret key  $K$  is available to the enclaved memory for decrypting the sensitive data. This system guarantees confidentiality and integrity of sensitive content using authenticated encryption. However, the application still relies on the OS to interact with the untrusted storage using *read-write channels* such as file system calls, memory page management and others.

Our baseline setting (shown in Fig. 1) is a system (such as Panoply [57]) where the read-write channels correspond to the read and write system calls. Although our discussion here is for the system call interface, our attack and defenses are applicable to other read-write operations that expose information at the granularity of blocks or memory pages, when caching or swapping out pages (for eg. in Haven [10], OverShadow [14]).



**Fig. 1.** Baseline setting for enclaved execution with untrusted read-write channels

**Attack Model.** In our model, the untrusted (or compromised) OS acts as a “honest-but-curious” adversary that honestly interacts with the application and the underlying encrypted storage. It passively observes the *input/output (I/O) profile* of the execution, but hopes to infer sensitive encrypted data. The I/O profile of an application is the “trace” of read-write file system calls made during the execution. The execution of an application  $A$  with sensitive input  $I$  and output  $O$  generates an I/O profile  $P = (P_1, P_2, \dots, P_n)$ . Each  $P_i$  is a read/write operation of the form  $[type, size, address, time]$  requested by the application  $A$ . Each  $P_i$  consists of four parameters:

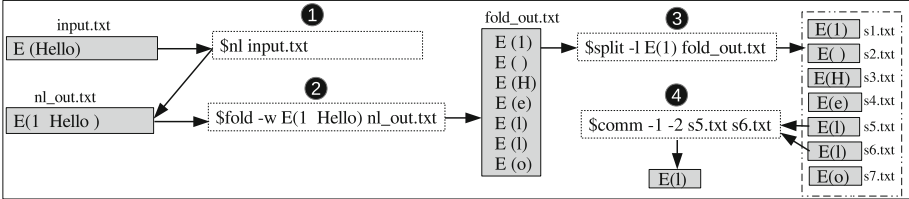
- (C1) type of operation (read or write)
- (C2) size of bytes read or written
- (C3) address (e.g. file name/descriptor) to read or write the content
- (C4) time interval between current and previous operation.

We assume the application  $A$  is publicly available and known to the adversary. Thus, the attacker’s knowledge set consists of  $\psi = \{A, |I|, |O|, P\}$  where  $|I|$  and  $|O|$  are the total input and output size, and  $A$  is the application logic. We assume the OS is capable of initiating the execution of any pre-installed application on encrypted inputs, in any order.

**Goal.** The goal is to make a benign enclaved application input-oblivious. An application that exhibits I/O profile  $P$  which is independent of the sensitive inputs exhibits the above security property. This security property guarantees that an adversary cannot distinguish between any two encrypted inputs of the same size when executed with the same application, leaking nothing beyond what is implied by knowledge of  $\psi$ .

## 2.1 Logic-Reuse Attack

To emphasize the importance of input-oblivious execution in the enclaved execution scenario, we demonstrate a concrete attack called the *logic-reuse attack*. In this attack, the adversary chains the execution of permitted applications to do its bidding (as shown in Fig. 2). Specifically, we show the use of four applications: `n1`, `fold`, `split` and `comm` from the CoreUtils package commonly available in commodity Linux systems [3]. These applications accept sensitive user arguments and file inputs in encrypted form. The attack exploits the execution I/O profile to eventually learn the comparison value of any two characters in the input encrypted file. The result is that the adversary infers the frequency and position of every byte in the target encrypted file. The 4 attack steps are:



**Fig. 2.** Attack example that leaks the frequency and position of characters in an encrypted file

**Step 1 - Get a known ciphertext value:** The `n1` command in CoreUtils adds a line number to each line in the input and writes the modified line to the output. The attacker executes this `n1` program with the target encrypted file (`input.txt` in Fig. 2) as its input. Every ciphertext is of 16 bytes given the use of AES encryption. Thus, the adversary learns that the first ciphertext of each write call contains the encryption of a number along with other characters (see `nl_out.txt` in Fig. 2).

**Step 2 - Generate the ciphertext for individual characters:** This step uses the `fold` program that folds input lines according to the given width size. The adversary runs this command on the output of the previous step. The ciphertext for encryption of number “1” (along with other characters) learned in Step 1 is used as encrypted input argument to the width size. This step folds the input file such that every line contains the ciphertext of a single character and makes a separate call to write it. After this step, the ciphertext for every individual character in the file is available to the adversary (as shown in `fold_out.txt` in Fig. 2).

**Step 3 - Save each ciphertext in a separate file:** In this step, the adversary uses the `split` program that splits an input file either line-wise or byte-wise and writes the output to different files. The command is run on the output of Step 2. The ciphertext of the character “1” learned in Step 2 is passed as an option to it. It generates separate files as output each having encryption of a single character (`s1.txt` - `s6.txt` in Fig. 2). Thus, the adversary learns the total number of characters and their positions in the input file.

**Step 4 - Compare the characters in each file:** Finally, the adversary executes the `comm` program that takes two files as input and writes the lines present in both the files as output. Any two files generated as output in the previous step can be provided as input to this command. The program does not perform a write call if there are no common lines in the input files. Thus, the I/O profile leaks whether two lines (or characters) in the input files are the same.

**Result:** In the end, this allows the attacker to infer a histogram of encrypted bytes. Once the histogram is recovered, it can be compared to standard frequency distribution of (say) English characters [1]. Using the values in the histogram and the positions learned in Step 3, the adversary learns the value of every byte in the encrypted file!

**Remarks.** Note that, the adversary neither tampers the integrity of the sensitive input nor disrupts the execution process in any manner throughout the attack. It simply invokes the applications on controlled arguments and honestly executes the read-write operations from the application without tampering any results. The adversary only passively observes the input-dependent I/O profile of the execution. Thus, we establish that it is practical to completely leak every byte in an encrypted file system in the absence of input-oblivious execution, when program logic running in enclave is sufficiently expressive.

### 3 Analysis of Information Leakage Channels

Recall that in our model, parameters in I/O profile  $P$  form the four channels **C1** to **C4** discussed in Sect. 2. The *type* parameter is either **R** (read) or **W** (write) call, *size* is the bytes read or written to the untrusted storage, and the *address*

signifies the file descriptor (fd) in use. Let *time* be the difference in the timestamp<sup>1</sup> for the occurrence of present and previous call. This section analyses the channels **C1** to **C4** in  $P$  for information leakage and their role in expanding attackers knowledge set  $\psi$ .

---

```

1  rsize = read(infile, inbuffer, 1, infilesize);
2  line1 = getline(inbuffer);
3  while((line2 = getline(inbuffer)) != NULL)
4    if ((linecompare(line1, line2)) == true)
5      match = true;
6    else
7      if(match == true)
8        wsize = write(repeat_out, line1, 1, strlen(line1));
9        match = false;
10   else
11     wsize = write(uniq_out, line1, 1, strlen(line1));
12   line1 = line2;

```

---

**Fig. 3.** Sample program which writes repeated lines in the input to repeat\_out file and the non-repeated lines to uniq\_out file.

Throughout the rest of the paper, we consider a running example similar to the **uniq** Unix utility (refer to Fig. 3) that has 4 information leakage channels. The example reads the data from an input file (line 1) and writes out consecutive repeated lines to **repeat\_out** file (line 8) and non-repeated lines to **uniq\_out** file (line 11). The code performs a character-by-character comparison (line 4) to check whether two lines are equal. Figure 4 shows the I/O profile that this program generates for two different inputs of the same size and the overall information learned about each input file. The I/O profile leaks the total number of input lines, output lines, repeated and non-repeated lines in the encrypted file.

**Sequence of Calls (C1).** The sequence of calls is an *input-dependent* parameter that depends on the **if** and loop terminating conditions in the application. In particular, the sequence of calls in the example are control-dependent on the bits from the sensitive input used in branch conditions.

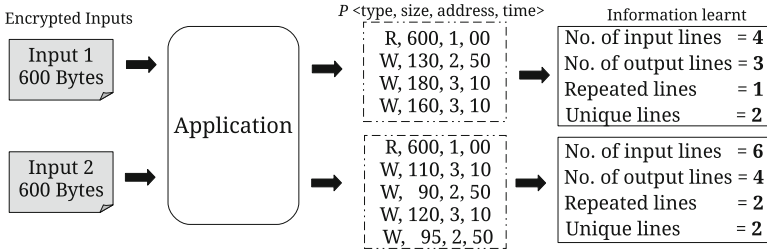
*Example.* The program in Fig. 3 uses a separate write call (highlighted) to output a new line<sup>2</sup>. Every time the adversary observes a **write** call in the I/O profile, it learns that a newline is written to the output file. This is beyond the allowed set  $\psi$  because it leaks the total number of lines in the output. From the I/O profiles in Fig. 4, the adversary learns that input 1 and 2 yield total of 3 and 4 lines as output respectively.

<sup>1</sup> The granularity of the clock is units of measurement as small as what the attacker can measure (e.g., ms, ns or even finer).

<sup>2</sup> This is a common programming practice observed in legacy applications such as CoreUtils as shown later in Sect. 6.

**Difference in Size of Bytes (C2).** The return values of the read and write system calls act as the *size* channel for information leakage. As the *size* parameter in the I/O profile  $P$  shows a direct data dependency on the input values, any difference in the value of this parameter leaks information about the encrypted inputs.

*Example.* In Fig. 4, the adversary observes that the difference in the size of total read and write bytes for input 1 is 130 bytes, inferring that 1 line is repeated. For input 2 the difference is 185 (90 + 95) bytes. Observing the size values in the profile for input 2, the adversary can infer that it has 2 lines repeated since no other combination of sizes result in a difference of 185 bytes.



**Fig. 4.** I/O profiles generated for two different inputs Input1 and Input2 of size 600 bytes. The numbers 1, 2 and 3 in the I/O profile are the file descriptors for infile, repeat\_out and uniq\_out respectively. The last part shows the information learned by observing the I/O profile.

**Address Access Patterns (C3).** We consider the file descriptor (fd) to the read and write system call as the *address* parameter in the I/O profile  $P$ . This is assuming the OS organizes its underlying storage in files. The untrusted OS infers the input dependent accesses patterns to different files from this parameter, as shown in the example below.

*Example.* In Fig. 4, the *address* parameter in  $P$  leaks that input 1 reads the repeat\_out file (fd = 2) once and input 2 reads it twice leaking that they contain 1 and 2 repeated lines respectively. Similar observation for uniq\_out file (fd = 3) leaks that input 1 and input 2 both have 2 unique lines.

**Side Channels - Time (C4).** There are several well-known side channels such as power consumption [41], cache latency [51], time [11, 42] that could leak information about sensitive inputs. We focus on the computation time difference between any two calls as a representative channel of information leakage. Our discussion applies more broadly to other observed channels too.

*Example.* In Fig. 4, readers can see that the computation time before a call that writes to repeat\_out file is 50 units and uniq\_out file is 10 units. A careful analysis of the time difference between all consecutive calls reveals that input 1 and 2 have 1 and 2 repeated lines respectively. This is because for repeated lines

the character-by-character comparison (line 4 in Fig. 3) proceeds till the end of the line, thus taking more time. However, the comparison fails immediately if the lines are not the same, reducing the time difference.

The above explanation with our running example establishes that every parameter in the I/O profile acts as an independent channel of information leakage. Each channel contributes towards increasing the  $\psi$  of an adversary.

**Table 1.** Systematization of existing defenses to hide leakage through I/O profile and their known limitations. ‘D’ and ‘L’ denote defenses and limitations.

Channel	D/L	Determinising I/O Profile	Randomizing I/O Profile
Type	D	Memory trace obliviousness [46] Ascend [22], CBMC-GC [34]	RandSys [38] RISE [9]
	L	Undecidability of static analysis [21, 43]	Infeasible sequences [33, 64]
Size	D	Rounding [13, 67], BuFLO [19]	Random padding [13], Random MTU padding [19]
	L	Storage Overhead	Assumption about input distribution
Address	D	Linear Scan [30, 40, 63, 71]	ORAM [28], [59]
	L	Access Overhead [30, 63]	$\text{polylog } N$ overhead [56]
Time	D	Normalized timing [11, 37] Language-based Approach [6, 17, 49, 70]	Fuzzy Time [35]
	L	Worst Case Execution Time [65]	Insufficient Entropy [27]

## 4 Defense: Approaches and Limitations

To block the above information channels (C1 to C4), the execution of an application should be input-oblivious i.e., the adversary cannot distinguish between two inputs by observing the I/O profile. We formally define the security property of “input-oblivious execution” as:

**Definition 1 (Input-Oblivious Execution).** *The execution of an application  $A$  is input-oblivious if, for any adversary  $\mathcal{A}$  given encrypted inputs  $E(i), E(j)$  and a query profile  $P$ , the following property holds:*

$$\text{Adv}_{\mathcal{A}} := |Pr[\mathcal{P} = \mathcal{P}[E(i)]] - Pr[\mathcal{P} = \mathcal{P}[E(j)]]| \leq \epsilon \quad (1)$$

where  $\epsilon$  is negligible.

There are two common approaches to achieve input-oblivious execution: (a) determinising the I/O profile and (b) randomizing the I/O profile. We study these existing defenses and show whether their limitations manifest in practical applications. Table 1 systematizes existing defenses and their limitations.

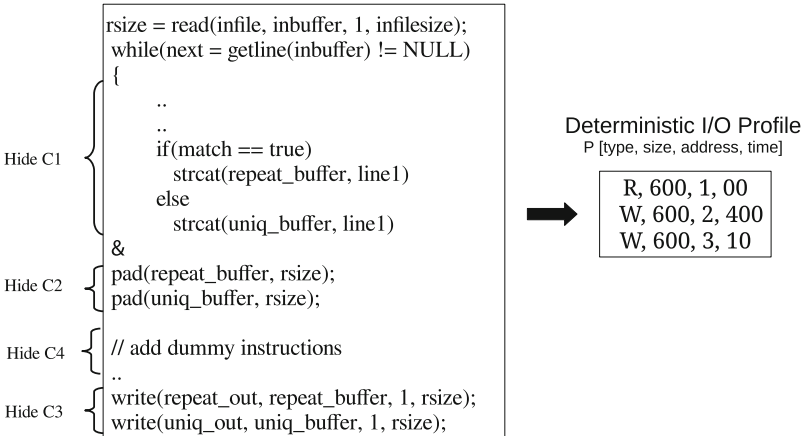


#### 4.1 Approach 1: Determinising the Profile

The idea is to make the execution of an application input-oblivious by determinising the parameters in the I/O profile. This forces a program operating on different inputs of the same size to generate equivalent I/O profiles. Figure 5 shows the modified code for our example (in Fig. 3) and its determinised I/O profile.

**Channel C1 - Type.** To determinise the *type* parameter in  $P$ , a program should have the same sequence of calls for different inputs irrespective of the path it executes. This requires making the execution of read/write calls independent of the sensitive data used in the branches or loops of a program. One way to achieve this is to move the read/write calls outside the conditional branches or the loop statements. This removes their dependence on any sensitive data that decides the execution path. The other method is to apply the idea of adding dummy instructions to both the branches of an `if` condition, as proposed in works on oblivious memory trace execution [22, 46]. This makes the I/O profile input-oblivious with respect to the `if` statements in the program. Loops can be determinised by fixing an upper bound on the number of iterations. Previous work on privacy preserving techniques use this method to remove the input-dependence in loops [34, 46].

*Example:* We show how to apply this idea to our running example. In Fig. 5, we determinise the sequence of calls by moving the write call outside the loop making them data-independent. All the lines are combined into a single buffer and are written outside the loop. This makes the profile  $P$  deterministic with respect to the *type* parameter while retaining the performance.



**Fig. 5.** Modified code with the defense to hide the channels of information leakage in I/O profile and a deterministic I/O profile for input of size 600 bytes.

**Channel C2 - Size.** To hide the leakage through *size* parameter, a straightforward method is to pad the data with dummy bytes up to a certain maximum value. Padding technique is used in several other contexts to hide leakage through message length. Chen et al. and Wright et al. use the idea of rounding messages to fixed length to prevent information leakage in web applications and encrypted VoIP conversations [13, 67]. Dyer et al. proposed the idea of BuFLO (Buffered Fixed Length Obfuscator) as a countermeasure against traffic analysis [19]. Similarly, in program execution, padding can be used to determinise channel **C2** by forcing the same value of *size* parameter in profile *P*.

*Example:* In Fig. 5, we pad the arguments to the write calls upto the size of total read bytes. This is because in our running example, the maximum output size equals the total input size when none of the input lines are repeated.

**Channel C3 - Address.** The pattern of *address* (file descriptor) parameter in profile *P* acts as a channel of information leakage. This is analogous to the memory access patterns observed in RAM memory. A memory address in the RAM model corresponds to a file descriptor in our setting. The simple approach to hide the address access patterns is to replace each access with a linear scan of all addresses [40]. In the context of secure two-party computations, Wang et al. and Gordon et al. show that linear scan approach is efficient for small number of addresses [30, 63]. Privacy preserving compilers such as PICCO use the linear scan approach to access encrypted indexes [71]. Linear scanning approach can be used to determinise the I/O profile with respect to the *address* parameter.

*Example:* In Fig. 5, we modify the program to access both the `repeat_out` and `uniq_out` file for every execution no matter whether the input file contains any repeated or unique lines. This makes the execution oblivious with respect to the *address* parameter.

**Channel C4 - Time.** Even if channels **C1** to **C3** are deterministic, the *time* parameter in the I/O profile leaks information about the sensitive input. Previous work have proposed hiding timing channels by making execution behaviour independent of sensitive values such that the security of program counter is preserved [17, 49]. Other approach is to transform applications to satisfy a specific type system that guarantees to hide the leakage through timing channel [6, 70]. For determinising the *time* parameter in *P*, we can use the idea of adding dummy instructions in the program to make the execution time a constant value as suggested in [11, 37].

*Example:* The input-oblivious version of the program (in Fig. 5) takes a constant time between the read and write calls in the I/O profile. For all inputs of size 600 bytes, the program will always take time of 400 units before it performs the first write call. The second write call follows immediately, thus taking less time.

## 4.2 Limitations of Determinising I/O Profile

Readers will notice that all the defenses to determinise the channels **C1** to **C4** exhibit one common characteristic. Each of the solution modifies their

corresponding parameters in the I/O profile to the worst-case execution time. This introduces a performance trade off in most of the applications. Deterministic approach requires statically deciding the upper bound for the worst-case values of all the profile parameters. This is not always possible due to the theoretical limitations of static analysis [21]. Statically identifying the upper bounds for loops is itself an undecidable problem and notoriously difficult in practice too [43, 54]. To explain the limitations, we use the `split` utility from CoreUtils package (shown in Fig. 6) which reads from an input file (line 1), splits a given input file line-wise (line 3) and writes the maximum  $B$  bytes as output to  $N$  different files (line 8).

---

```

1  n_read = safe_read (STDIN_FILENO, buf, bufsize);
2  while (true)
3      bp = memchr (bp, '\n', eob - bp + 1);
4      if (bp == eob)
5          break;
6      ++bp;
7      if (++n >= n_lines)
8          cwrite (new_file_flag, bp_out, bp - bp_out);
9          bp_out = bp;
10         new_file_flag = true;
11         n = 0;

```

---

**Fig. 6.** `split` program code that splits the lines in input file and writes to different output files

**Type.** In Fig. 6, it is difficult to statically decide a “feasible” upper bound on the number of loop iterations. In the worst case, a file can have a single character on each line in the input file. To explicitly decide an upper bound for a file of size around 1 GB, a determinised profile will execute the loop for  $N = 2^{30}$  times (assuming one byte on each line) which is not a reasonable solution.

**Address.** The simple strategy of linearly accessing all addresses suffers from an overhead proportional to the maximum addresses an application uses during the execution [63]. In `split` program performing linear access incurs a total overhead of  $N^2$  i.e., accessing  $N$  files for each of the  $N$  loop iterations (where  $N = 2^{30}$  in worst case). This is impractical for real usage, unless  $N$  is small.

**Size.** Padding data with dummy bytes up to a maximum output size incurs huge storage overhead as shown in previous work [13, 19]. In our `split` example, for a 1 GB input file, the maximum possible bytes in a line is  $B = 1$  GB, when no newline characters are present in the file. Thus, determinising the `split` program results in total storage overhead of  $N$  GB. It becomes  $N^2$  GB when the I/O profile is determinised with respect to *address* channel.

**Time.** Determinising the *time* channel results in worst case execution time for the application for different inputs of the same size [11, 65]. For a file of 1 GB,

`split` program will take equal time for input file having single character on every line or the whole file having just a single line.

### 4.3 Approach 2: Randomizing the I/O Profile

The second approach to making application execution input-oblivious is transforming the original I/O profile to a randomized profile. Randomizing the I/O profile involves addition of sufficient noise to every parameter in  $P$ . One advantage of randomization over determinising the profile is that it scales better in terms of performance for most of the applications. We explain this paradigm of randomization techniques using the `split` example in Fig. 6.

**Oblivious RAM.** A strategy for randomizing the address access patterns which is the focus of many current research works is to use Oblivious RAM (ORAM) [28, 56, 58, 59, 62]. ORAM technique replaces each read/write operation in the program with many operations and shuffles the mapping of content in the memory to hide the original access patterns [28]. With the best ORAM techniques, the application only needs to perform  $\text{poly log } N$  operations to hide the access pattern where  $N$  is the total address space [56, 58]. This is strictly better as compared to linear overhead of  $N$  operations in the trivial approach. Use of ORAM has been proposed in various areas such as cloud storage [29], file system [66] and so on. Similarly, we can apply ORAM to randomize the file descriptor parameter in the I/O profile during program execution.

*Example:* In Fig. 6, `split` program splits the input file and writes the output to  $N$  files, we can make the I/O profile oblivious by making the program write only to  $\text{poly log } N$  files using ORAM. Thus, the overhead reduces to  $N \cdot \text{polylog } N$  and is strictly better than  $N^2$  in the case of determinising the profile.

**Addition of Noise.** Randomization involves addition of random noise to the parameters in profile  $P$  such that the I/O profiles for two different inputs are indistinguishable. For this to work, we assume the enclaved application has access to a secure source of randomness. We can employ the techniques similar to those used in determinising the profile such as insertion of calls, padding of bytes and addition of dummy instructions to randomize the I/O profile as well. Randomization as a defense is popularly used in Instruction Set Randomization (ISR) and Address Space Layout Randomization (ASLR) techniques to prevent attacks on execution of benign applications [9, 39, 55]. RandSys combines these two techniques and proposes randomization at the system call interface layer [38]. This approach can be used to randomize the sequence of calls in the I/O profile of applications. Hiding of message length using random padding is explored in depth in previous work in the context of web applications [13, 19]. Effects of using same random number for all messages versus different random number for each message was shown in [19]. Recent work has focussed on use of differential privacy techniques [18], to randomly pad the traffic in web application [8]. We can apply similar techniques to randomize the bytes in the I/O profile of an application. Finally, to randomize the *time* channel, we can use existing ideas

that makes all the clocks available to the adversary noisy for reducing the leakage through timing channels [35].

*Example:* For a file size of 1 GB, an efficient random padding technique for `split` program in Fig. 6 writes bytes less than the maximum value for most of the write calls. This requires storage less than the worst case scenario.

#### 4.4 Limitations of Randomizing I/O Profile

Although randomizing I/O profile provides better performance in most applications, it does not imply ease of deployment in real applications.

**Infeasible Sequence.** Randomizing the *type* parameter in the I/O profiles may introduce sequence of system calls which are not possible for a given application. An adversary detecting such infeasible sequences learns about the additional (fake) system calls inserted to make the profile input-oblivious. This is a valid threat as adversary has access to the application logic and hence can notice any irregular sequences. We call this as the “infeasible profile detection” attack. To avoid this, an application needs to guarantee that a randomized sequence is always a subset of feasible sequences. This requires generating a complete set of feasible sequence of calls for a given application which is a theoretical and practical limitation using path-feasibility analyses (eg. symbolic execution) [43].

*Example:* A simple example is the `split` program in Fig. 6 which compulsorily performs a read operation followed by a series of writes to different files. A randomized sequence of calls such as `read`, `write`, `write`, `read`, `write` alarms the adversary that the second `read` call is a fake. This immediately leaks that at most 2 lines are written out by the program before the occurrence of the fake call i.e., the value of variable *n* (at line 7 in Fig. 6) is at most 2. The adversary can iterate the execution sufficient number of times and collect different samples of I/O profile for the same input. With the knowledge of infeasible sequences and identifying the fake calls in each profile, the adversary can recover the original sequences in finite time and learn the actual number of lines in the encrypted input file.

**Assumption About Input Distribution.** The randomization approach often performs better than determinising the profile as it does not always effect the worst case behaviour. However, to reap the performance benefits of randomization, it is necessary to know the input distribution [13].

*Example:* To efficiently pad the *size* channel in the `split` program, the distribution of output bytes (*B*) for an input file with English sentences can be known using possible sentence lengths in English [53]. But the distribution is different for a file that contains numerical recording of weather or genome sequence information. When we compile the application, we may not know this distribution. However, a significant challenge is to know beforehand the appropriate distribution of all possible inputs to an application. It is practically infeasible for common applications such as found it CoreUtils which take variety of input.

**Insufficient Entropy.** With insufficient entropy, the adversary can perform repeated sampling to remove the randomization effect and recover the original profile. Gianvecchio et al. show how entropy can be used to accurately detect covert timing channel [27]. Cock et al. perform empirical analysis to show that although storage channels are possible to eliminate, timing channels are a last mile while thwarting leakage through side channels [16]. Similarly, other channels can be recovered if the source of randomness does not provide sufficient entropy—well-known from other randomization defenses [55]. It is necessary to ensure that the source of random number which the application uses is secure and the amount of entropy is large enough.

## 5 Insufficiency of Hiding Selective Channels

Defenses for both determinising and randomizing the profile have limitations that affect their use in practical applications. One might hope to selectively hide one or more leakage channels so that the transformed applications are still practical to use. This hypothesis assumes that blocking one channel as well hides the leakage through other channels. This section attempts to answer the question: Is it sufficient to hide partial channels to get input-oblivious execution? Or, does hiding one channel affect the amount of leakage through another channel?

**Hiding Only the Address Channel.** In the logic-reuse attack (Sect. 2.1), recall that the `split` application writes the ciphertext of each character in the input to a separate output file (refer Fig. 2). Using ORAM in the `split` program hides the exact file to which the ciphertext is written. It replaces every write call with  $\text{poly log } N$  calls where  $N$  is the total number of characters in the input file. But this is not sufficient to mitigate the attack. The adversary can get the ciphertext for number “1” by brute forcing all the ciphertexts written by the first  $\text{poly log } N$  calls. ORAM just makes it harder for the adversary to get exactly the required ciphertext. With ORAM, the adversary has to try the Step 2 of attack with poly-log input ciphertext. This is expected as ORAM only blocks the leakage through *address* parameter but does not hide the sequence of calls. Recall that the leakage through other parameters like *type*, *size* and *time* are sufficient for the attack to succeed. The adversary can observe the partially oblivious I/O profile and still infer every byte in the encrypted file. Our logic-reuse attacks even works in the presence of ORAM defense for hiding address access patterns.

**Hiding Only the Type Channel.** Let us assume that the *size* parameter for write calls in our running example in Fig. 3 always has the same value. This is possible when all the lines in an input file have the same length. Such an I/O profile is deterministic with respect to the size channel. To hide the leakage through *type* channel, let us move the write calls outside the loop (as shown in Fig. 5). In this case, the *type* channel is determinised but the leakage is not actually blocked. The leakage is shifted to the *size* channel which now has different values depending on the number of repeated lines and unique line. This shows that in determinising the *type* channel the leakage simply gets “morphed” to the

---

```

1 while (( bytes_read = fread (buf, 1, BUFLen, fp)) > 0)
2   unsigned char *cp = buf;
3   length += bytes_read;
4   while (bytes_read--)
5     crc = (crc << 8) ^ crcTab[(( crc >> 24) ^ *cp++) & 0xFF];
6   if (feof (fp))
7     break;
8   printf ("%u %s", (unsigned int) crc, hp);

```

---

**Fig. 7.** cksum code with no channel of information leakage

---

```

1 for (int i = 0; i < n_lines; i++)
2   char *const *p = line + permutation[i];
3   size_t len = p[1] - p[0];
4   if (fwrite (p[0], sizeof *p[0], len, stdout) != len)
5     return -1;

```

---

**Fig. 8.** shuf utility code that leaks the number of lines in input file

*size* channel, not really eliminated. This shows that its often misleading to selectively hide some subset of information channels due to this channel morphism problem.

In summary, transforming an application to input-oblivious execution involves two important steps: (a) correctly identifying all the channels of information leakage in profile  $P$  and (b) applying either deterministic or randomization approach to hide all the channel simultaneously.

## 6 Case Studies

**Selection of Benchmarks.** We select CoreUtils and BusyBox that are commonly available on Unix system as our benchmarks [2,3]. We choose all the 28 text utilities<sup>3</sup> from GNU CoreUtils package, 1 utility (**grep**) from BusyBox and the **file** utility as our case studies. All of them perform text manipulation on input files. With this benchmark, our goal is to answer the following questions.

- (a) Does information leak through I/O profile in practical applications?
- (b) Is it possible to convert practical applications to input-oblivious execution?

### 6.1 Analysis Results

We analyze these 30 applications for read/write channels and manually transform them to perform input-oblivious execution. We use the **strace** utility available

---

<sup>3</sup> The class of utilities that operate on the text of the input files. Other classes in coreutils include file utilities that operate on file metadata and shell utilities.

---

```

1  if ((m = file_is_tar(ms, ubuf, nb)) != 0) /*Tar check*/
2      file_printf(ms, "%s", code_mime);
3
4  if ((m = file_trycdf(ms, fd, ubuf, nb)) != 0) /*CDF check*/
5      file_printf(ms, "%s", code_mime);
6      /*text check*/
7  if ((m = file_ascmagic(ms, ubuf, nb, looks_text)) != 0)
8      file_printf(ms, "%s", code_mime);

```

---

**Fig. 9.** file utility code that leaks the file type through the *time* channel

in Linux system to log all the interactions of the application with the untrusted OS. The “-tt” option of strace gives the time-stamp for every system call made by the application. We categorize each application into one or more channels (discussed in Sect. 3) which need to be blocked for providing input-oblivious execution. The *type*, *size*, *address* and *time* channel leak information in 22, 11, 2, 24 applications respectively. Table 2 summarizes our analysis results.

**No Channels.** Out of the 30 case studies, 6 applications perform nearly input-oblivious execution without modification. These programs include `sum`, `cksum`, `cat`, `base64`, `od` and `md5sum`. Figure 7 shows the code for `cksum` program as a representative to describe the behaviour in these applications. The `while` loop at line 1 uses the input size for termination which is a part of adversary’s knowledge set  $\psi$ . Therefore the program generates the same sequence of calls for different inputs of the same size. As the same computation is performed on every character (line 5), the time interval between the calls is the same for different inputs. Thus, the I/O profile of the program execution does not depend on sensitive input. These 6 applications generate deterministic profiles by default and thereby exhibit the property of input-oblivious execution.

**Type.** Of the remaining 24, 22 generate sensitive input-dependent sequence of calls. We observe that 8 of the 22 applications specifically leak the number of newline characters present in the input file. Figure 8 shows the code for `shuf` utility that shuffles the arrangement of lines in the input file and outputs every line with a separate write call (line 4). These 8 applications include `ptx`, `shuf`, `sort`, `expand`, `unexpand`, `tac`, `nl` and `paste`. Other applications such as `cut`, `fold`, `fmt`, `tr`, `split` and so on leak additional information about the sensitive input depending on the options provided to these applications. Recall that in our logic-reuse attack, the command “`fold -w E(1 Hello)`” leaks the ciphertext for individual characters in the input file.

**Size.** In our case studies, applications that writes as output either partial or complete data from the input file are categorized as leaking channel through *size* parameter. 11 of our 30 case studies fall under this channel namely `tr`, `comm`, `join`, `uniq`, `grep`, `cut`, `head`, `tail`, `split`, `csplit` and `tsort`. All these applications as well leak information through the *type* parameter. This means that none of the 11 utilities leak information exclusively through *size* parameters.



Such a behaviour indicates that even if one of the channels is blocked, information is still leaked and shifted over to another channel (refer Sect. 5).

**Address.** Most of the applications in our case studies read and write to a single file with the exception of two utilities. The `split` and `csplit` programs access different output files during the execution process. Thus, these two application leak information via the address access pattern. From Table 2, readers can see that these are the only two applications that leak information through all the four channels in the I/O profile.

**Time.** All the 24 applications that do not fall in the no channel category leak information through the *time* parameter in the I/O profile. Readers can observe from Table 2 that only two programs i.e., `wc` and `file` leak information explicitly through timing channel. The code snippet of `file` utility in Fig. 9 explains this behaviour. The `file` reads the input and checks it for each file type (line 1, 4 and 7). The I/O profile contains only one read and write call but the time difference between the them leaks information about the input file type.

**Table 2.** Categorization of CoreUtils applications into different leakage channels. ✓ denotes that the channel should be blocked to make the application input-oblivious

	paste	sort	shuf	ptx	expand	unexpand	tac	grep	cut	join	uniq	comm
Type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Size								✓	✓	✓	✓	✓
Address												
Time	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	fold	fmt	nl	pr	split	csplit	tr	head	tail	tsort	file	wc
Type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Size					✓	✓						
Address					✓	✓						
Time	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No Channel - cat , cksum , sum , base64 , md5sum , od												

## 6.2 Can Be Transformed to Input-Oblivious Execution?

To answer our second evaluation goal, we manually transform the applications using the defenses discussed in Sect. 4. Since all the applications leak information through timing channel for a fine grained measurement by adversary, we ignore the timing channel in our manual transformation. We find some positive results where the existing defenses can be directly applied to make commonly used applications input-oblivious. Surprisingly, our findings yield negative results as well. We show that the limitations of oblivious execution techniques do manifest in 2 real applications.

**Transformed with  $O(1)$  Overhead.** We find that 11 applications can execute obliviously by the determinising the profile with respect to the *type* parameter.

These are the applications in Table 2 which fall only under *type* and *time* channels and no others. In all these applications the sequence of call can be made independent of the loops that use sensitive data for termination. The code for `shuf` in Fig. 8 is an example of such an application. Thus, there is no performance overhead due to determinising the *type* channel. We consider this to be a positive result as the applications can be transformed with  $O(1)$  overhead.

**Transformed with  $O(N)$  Overhead.** We find that 11 applications that leak information through both the *type* and *size* channel can be converted to input-oblivious execution by making the sequence of calls loop independent as well as padding the output bytes to the total input size. These transformed applications incur a performance penalty of  $O(N)$  i.e., linear to the size of input file.

**Transformed with Exponential Performance Penalty.** We find that 2 applications namely `split` and `csplit` show the limitation of statically deciding a feasible upper bound for loops. In these programs, the number of loop iteration depends on the number of newline characters present in the input file (line 3 in Fig. 6) which is not known at the compile time. Hence, transforming these applications to input-oblivious execution is not possible without exponential performance overhead of  $O(2^N)$ . We explain this behaviour for `split` program earlier in Sect. 4. The `csplit` application is similar to `split` with additional options to it and therefore exhibits same limitations. This confirms that limitations of existing oblivious execution techniques do manifest in practical applications.

## 7 Related Work

**Attacks on Enclaved Systems.** On a similar setting as this paper, Xu et al. demonstrate controlled-channel attack using page faults that can extract complete text documents in presence of an untrusted OS [69]. This confirms that enclaved execution techniques are vulnerable to information leakage through different channels. Our work specifically focuses on file system calls as the read-write channels in these systems. Iago attacks [12] demonstrate that untrusted OS can corrupt application behaviour and exploit to gain knowledge about sensitive inputs. This attack however assumes the OS is malicious and can tamper the parameter of return values in memory management system calls like `mmap`. In this paper, we have shown that information leakage is possible even with a weaker i.e., semi-honest adversarial model.

**Oblivious Execution Techniques.** A discussion of closely related oblivious execution techniques is summarized in Sect. 4 (see Table 1). Here we discuss a representative set of recent work on these defenses. Liu et al. [46] propose a type system for memory-trace oblivious (MTO) execution in the RAM model. In their solution, they add padding instructions to ‘if’ and ‘else’ branches to achieve memory trace obliviousness. We use this technique to hide the system call sequences in I/O profile. Along with this, they use the ORAM technique to hide address access patterns. GhostRider [45] provides a hardware/software platform for privacy preserving computation in cloud with the guarantees of memory-trace oblivious execution. Along with hiding address access pattern GhostRider

determines the time channel by making the application take worst case execution time. Ascend [22] is a secure processor that uses randomizes access pattern using ORAM and determines the time channel by allowing access to memory at fixed intervals. The fixed interval is a parameter chosen at compile time. It uses the idea of inserting dummy memory access to hide the timing channel. Fletcher et al. have proposed a solution that provides better performance while still hiding the timing channel [23]. However, their solutions leaks a constant amount of information, thus introducing a tradeoff between efficiency and privacy.

## 8 Conclusion

In this paper we demonstrate a concrete attack called—a logic-reuse attack—to highlight the importance of oblivious execution. We systematize the capabilities and limits of existing oblivious execution techniques in the context of enclaved execution. Finally, our study on 30 applications demonstrate that most of the practical applications can be converted to oblivious execution with acceptable performance. However, theoretical limitations of oblivious execution do manifest in practical applications.

**Acknowledgements.** We thank the anonymous reviewers of this paper for their helpful feedback. We also thank Shweta Shinde, Zheng Leong Chua and Loi Luu for useful feedback on an early version of the paper. This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-560-112 and a university research grant from Intel. All opinions expressed in this work are solely those of the authors.

## References

1. <http://letterfrequency.org>
2. BusyBox. <http://www.gnu.org/software/coreutils/>
3. GNU CoreUtils. <http://www.busybox.net/>
4. Intel Trusted Execution Technology: Software Development Guide. [www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf](http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf)
5. Trusted Computing Group. Trusted platform module, July 2007
6. Agat, J.: Transforming out timing leaks. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2000 (2000)
7. ARM: ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper (2013)
8. Azab, T.: Differentially private traffic padding for web applications. Ph.D. thesis, Concordia University Montreal, Quebec (2014)
9. Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the 10th ACM conference on Computer and communications security (2003)
10. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: OSDI (2014)

11. Brumley, D., Boneh, D.: Remote timing attacks are practical. *Comput. Networks* **48**(5), 701–716 (2005)
12. Checkoway, S., Shacham, H.: Iago attacks: why the system call API is a bad untrusted RPC interface. In: *ASPLOS* (2013)
13. Chen, S., Wang, R., Wang, X., Zhang, K.: Side-channel leaks in web applications: a reality today, a challenge tomorrow. In: *IEEE Symposium on Security and Privacy (SP)*, pp. 191–206. IEEE (2010)
14. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dworkin, J., Ports, D.R.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems (2008)
15. Chhabra, S., Rogers, B., Solihin, Y., Prvulovic, M.: SecureME: a hardware-software approach to full system security. In: *ICS* (2011)
16. Cock, D., Ge, Q., Murray, T., Heiser, G.: The last mile: an empirical study of timing channels on sel4. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014* (2014)
17. Coppens, B., Verbaauwhede, I., De Bosschere, K., De Sutter, B.: Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: *30th IEEE Symposium on Security and Privacy*, pp. 45–60. IEEE (2009)
18. Dwork, C., van Tilborg, H.C.A., Jajodia, S.: Differential privacy. *Encyclopedia of Cryptography and Security*, pp. 338–340. Springer, New York (2011)
19. Dyer, K.P., Coull, S.E., Ristenpart, T., Shrimpton, T.: Peek-a-boo, i still see you: why efficient traffic analysis countermeasures fail. In: *IEEE Symposium on Security and Privacy (SP)*, pp. 332–346. IEEE (2012)
20. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: Blakley, G.R., Chaum, D. (eds.) *CRYPTO 1984*. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1985). doi:[10.1007/3-540-39568-7\\_2](https://doi.org/10.1007/3-540-39568-7_2)
21. Fairley, R.E.: Tutorial: static analysis and dynamic testing of computer software. *Computer* (1978)
22. Fletcher, C.W., Dijk, M.V., Devadas, S.: A secure processor architecture for encrypted computation on untrusted programs. In: *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pp. 3–8. ACM (2012)
23. Fletcher, C.W., Ren, L., Yu, X., Van Dijk, M., Khan, O., Devadas, S.: Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 213–224. IEEE (2014)
24. Gentry, C., Halevi, S.: Implementing gentry’s fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) *EUROCRYPT 2011*. LNCS, vol. 6632, pp. 129–148. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-20465-4\\_9](https://doi.org/10.1007/978-3-642-20465-4_9)
25. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: *41st Annual ACM Symposium on Theory of Computing* (2009)
26. Gentry, C., Halevi, S.: A working implementation of fully homomorphic encryption. In: *EUROCRYPT* (2010)
27. Gianvecchio, S., Wang, H.: Detecting covert timing channels: an entropy-based approach. In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM (2007)
28. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. *J. ACM* **43**(3), 431–473 (1996)
29. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012)

30. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012 (2012)
31. Henecka, W., Kogl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: ACM CCS (2010)
32. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: InkTag: secure applications on an untrusted operating system. In: ASPLOS (2013)
33. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. *J. Comput. Secur.* **6**(3), 151–180 (1998)
34. Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012 (2012)
35. Hu, W.M.: Reducing timing channels with fuzzy time. In: IEEE Computer Society Symposium on Research in Security and Privacy, Proceedings, pp. 8–20, May 1991
36. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security Symposium (2011)
37. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: IEEE Symposium on Security and Privacy (SP) (2013)
38. Jiang, X., Wang, H.J., Xu, D., Wang, Y.M.: RandSys: thwarting code injection attacks with system service interface randomization. In: 26th IEEE International Symposium on Reliable Distributed Systems, SRDS 2007, pp. 209–218. IEEE (2007)
39. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 272–280. ACM (2003)
40. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 506–525. Springer, Heidelberg (2014). doi:[10.1007/978-3-662-45608-8\\_27](https://doi.org/10.1007/978-3-662-45608-8_27)
41. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999). doi:[10.1007/3-540-48405-1\\_25](https://doi.org/10.1007/3-540-48405-1_25)
42. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). doi:[10.1007/3-540-68697-5\\_9](https://doi.org/10.1007/3-540-68697-5_9)
43. Landi, W.: Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.* **1**(4), 323–337 (1992)
44. Li, X., Hu, H., Bai, G., Jia, Y., Liang, Z., Saxena, P.: DroidVault: a trusted data vault for android devices. In: 19th International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 29–38. IEEE (2014)
45. Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M., Shi, E.: GhostRider: A hardware-software system for memory trace oblivious computation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 87–101. ACM (2015)
46. Liu, C., Hicks, M., Shi, E.: Memory trace oblivious program execution. In: CSF 2013, pp. 51–65 (2013)
47. McCune, J.M., Parnoy, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for TCB minimization. In: EuroSys (2008)

48. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP (2013)
49. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: automatic detection and removal of control-flow side channel attacks. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 156–168. Springer, Heidelberg (2006). doi:[10.1007/11734727\\_14](https://doi.org/10.1007/11734727_14)
50. Osadchy, M., Pinkas, B., Jarrous, A., Moskovich, B.: SCiFI - a system for secure face identification. In: Security and Privacy (2010)
51. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006). doi:[10.1007/11605805\\_1](https://doi.org/10.1007/11605805_1)
52. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). doi:[10.1007/3-540-48910-X\\_16](https://doi.org/10.1007/3-540-48910-X_16)
53. Quirk, R., Crystal, D., Education, P.: A Comprehensive Grammar of the English Language, vol. 397. Cambridge University Press, Cambridge (1985)
54. Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 225–236. ACM (2009)
55. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 298–307. ACM (2004)
56. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with  $O((\log N)^3)$  worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-25385-0\\_11](https://doi.org/10.1007/978-3-642-25385-0_11)
57. Shinde, S., Le Tien, D., Tople, S., Saxena, P.: Panoply: Low-TCB linux applications with SGX enclaves. In: NDSS (2017)
58. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013 (2013)
59. Stefanov, E., Shi, E., Song, D.: Towards Practical Oblivious RAM. CoRR (2011)
60. Thekkath, D.L.C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX (2000)
61. Tople, S., Shinde, S., Chen, Z., Saxena, P.: AUTOCRYPT: enabling homomorphic computation on servers to protect sensitive web content. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013 (2013)
62. Wang, X.S., Chan, T.H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound (2014)
63. Wang, X.S., Huang, Y., Chan, T., Shelat, A., Shi, E.: SCORAM: Oblivious RAM for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 191–202. ACM (2014)
64. Warrender, C., Forrest, S., Pearlmuter, B.: Detecting intrusions using system calls: alternative data models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp. 133–145. IEEE (1999)

65. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst. (TECS)* **7**(3), 36 (2008)
66. Williams, P., Sion, R., Tomescu, A.: PrivateFS: a parallel oblivious file system. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012*
67. Wright, C.V., Ballard, L., Coull, S.E., Monrose, F., Masson, G.M.: Spot me if you can: uncovering spoken phrases in encrypted VoIP conversations. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP 2008* (2008)
68. Yao, A.C.: Protocols for secure computations. In: *23rd Annual IEEE Symposium on Foundations of Computer Science* (1982)
69. Xu, Y., Cui, W., Peinado, M.: GhostRider: Controlled-channel attacks: deterministic side channels for untrusted operating systems. In: *IEEE Security and Privacy* 2015 (2015)
70. Zhang, D., Askarov, A., Myers, A.C.: Language-based control and mitigation of timing channels. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012* (2012)
71. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013* (2013)

Detection of Intrusions and Malware, and Vulnerability  
Assessment

14th International Conference, DIMVA 2017, Bonn,

Germany, July 6-7, 2017, Proceedings

Polychronakis, M.; Meier, M. (Eds.)

2017, X, 412 p. 114 illus., Softcover

ISBN: 978-3-319-60875-4