Preventing Page Faults from Telling Your Secrets

Shweta Shinde   Zheng Leong Chua   Viswesh Narayanan   Prateek Saxena

National University of Singapore

{shweta24, chuazl, visweshn, prateeks} @comp.nus.edu.sg

ABSTRACT

New hardware primitives such as Intel SGX secure a user-level process in presence of an untrusted or compromised OS. Such "enclaved execution" systems are vulnerable to several side-channels, one of which is the page fault channel. In this paper, we show that the page fault side-channel has sufficient channel capacity to extract bits of encryption keys from commodity implementations of cryptographic routines in OpenSSL and Libgcrypt— leaking 27% on average and up to 100% of the secret bits in many case-studies. To mitigate this, we propose a software-only defense that masks page fault patterns by determinising the program's memory access behavior. We show that such a technique can be built into a compiler, and implement it for a subset of C which is sufficient to handle the cryptographic routines we study. This defense when implemented generically can have significant overhead of up to $4000\times$, but with help of developer-assisted compiler optimizations, the overhead reduces to at most 29.22% in our case studies. Finally, we discuss scope for hardware-assisted defenses, and show one solution that can reduce overheads to 6.77% with support from hardware changes.

1. INTRODUCTION

Operating systems are designed to execute at higher privileges than applications on commodity systems. Recently, this model of assuming a trusted OS has come under question, with the rise of vulnerabilities targeting privileged software [24]. Consequently, new hardware primitives have emerged to safeguard applications from untrusted OSes [36,37,47]. One such primitive is Intel SGX's enclaved execution which supports secure execution of sensitive applications on an untrusted OS. The SGX hardware guarantees that all the application memory is secured and the OS cannot access the application content. During execution, applications rely on the OS for memory management, scheduling and other system services. Intel SGX holds the promise of affording a private virtual address space for a trusted process that is immune to active probing attacks from the hostile OS. However, side-channels such as the page-fault channel have been recently discovered [51]. Since the OS manages the virtual-to-physical page translation tables for the

sensitive application, it can observe all page faults and the faulting page addresses, which leaks information. These attacks show that mere memory access control and encryption is not enough to defend against the OS, which motivates a systematic study of defense solutions to mitigate this channel.

In this paper, we first show that the channel capacity of the page-fault channel is sufficient to extract secret key information in existing implementations of cryptographic routines (OpenSSL and Libgcrypt). Cryptographic routines are vital to reducing the TCB and enclaved applications are expected to critically rely on them to establish secure channel with the I/O, filesystem and network sub-systems [10, 27, 41]. To perform an attack, the adversarial OS allocates a minimum number of physical pages to the sensitive enclave process, such that memory accesses spill out of the allocated set as much as possible, incurring page faults. We call such attacks as *pigeonhole attacks*[1] because they force the victim process to spill outside the allocated physical pages, thereby maximizing the channel capacity of the observed side-channel. They affect a long line of systems such as Intel SGX [37], InkTag [28], PodArch [44], and OverShadow [18] which protect application memory.

The page fault channel is much easier for the OS to exploit as compared to other side-channels. For example, in case of cache side-channel, the hardware resources such as size, number of data entries, eviction algorithm and so on are often fixed. The adversary has a limited control on these factors and the observations are mainly local to small fragments of program logic. On the contrary, in case of pigeonhole attacks, adversary is much stronger, adaptive, and controls the underlying physical resource (the number of physical pages). Moreover, it can make far more granular clock measurements (both global and local) by invoking and inducing a fault in the enclave. To defend applications against this unaddressed threat, we seek a security property that allows an application to execute on any input data while being agnostic to changes in the number of pages allocated. The property assures that the OS cannot glean any sensitive information by observing page faults. We call this property as *page-fault obliviousness* (or PF-obliviousness).

In this work, we propose a purely software-based defense against pigeonhole attacks to achieve PF-obliviousness. We point out that defenses against time and cache side-channels do not directly prevent pigeonhole attacks, and achieving PF-obliviousness has been an open problem [51]. Our goal is to guarantee that even if the OS observes the page faults, it cannot distinguish the enclaved execution under any values for the secret input variables. Our propose approach is called *deterministic multiplexing*, wherein the enclave application exhibits the same page fault pattern under all values possible for the secret input variables. Specifically, we modify the

---

[1] These attacks were also referred to as controlled-channel attacks in previous work.

program to pro-actively access all its input-dependent data and code pages in the same sequence irrespective of the input. In our empirical case studies, the naive implementation of deterministic multiplexing results in an overhead of about $705\times$ on an average and maximum $4000\times$! Therefore, we propose several optimizations techniques which exploit specific program structure and makes the overhead statistically insignificant in 8 cases, while the worst-case performance is 29.22%. All our defenses are implemented as an extension to the LLVM compiler, presently handling a subset of C/C++ sufficient to handle the cryptographic case studies. Finally, we discuss alternative solutions for efficient defenses, and suggest a new defense which requires hardware support, but yields an acceptable worst-case overhead of 6.67% for our case studies.

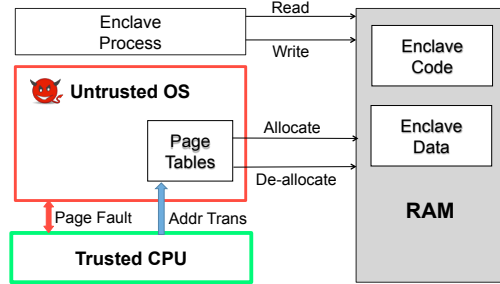**Contributions.** We make the following contributions:

- *Pigeonhole attacks on real cryptographic routines.* We demonstrate that the page-fault channel has sufficient capacity to extract significant secret information in widely-used basic cryptographic implementations (e.g., AES, EdDSA, RSA).
- *Defense.* We propose PF-obliviousness and design deterministic multiplexing approach that eliminates information leakage via page fault channel.
- *Optimizations & System Evaluation.* We apply our defense to the vulnerable cryptographic utilities from Libgcrypt and OpenSSL, and devise sound optimizations. In our experiments, deterministic multiplexing amounts to an average of $705\times$ overhead without optimization, and is reduced to an acceptable average and worst case overhead of 29.22% after optimization.

2. PIGEONHOLE ATTACKS

In a non-enclaved environment, the OS is responsible for managing the process memory. Specifically, when launching the process, the OS creates the page tables and populates empty entries for virtual addresses specified in the application binary. When a process begins its execution, none of its virtual pages are mapped to the physical memory. When the process tries to access a virtual address, the CPU incurs a page fault. The CPU reports information such as the faulting address, type of page access, and so on to the OS on behalf of the faulting process, and the OS swaps in the content from the disk. Similarly, the OS deletes the virtual-to-physical mappings when it reclaims the process physical memory as and when requested or when necessary. Thus, a benign OS makes sure that the process has sufficient memory for execution, typically, at least 20 pages in Linux systems [13].

2.1 Benign Enclaved Execution

The aim of enclave-like systems is to safeguard all the sensitive process (called as an enclave) memory during the execution. These systems use memory encryption [18, 44] and / or memory access controls [28, 37] to preserve the confidentiality of the sensitive content. The process memory is protected such that the hardware allows access in ring-3 only when a legitimate owner process requests to access its content [18]. When the OS in ring-0 or any other process in ring-3 tries to access the memory, the hardware either encrypts the content on-demand or denies the access. This guarantees that neither the OS nor other malicious processes can access the physical memory of an enclave. In enclaved execution, the OS memory management functions are unchanged. The onus still lies with the OS to decide which process gets how much physical memory, and which pages should be loaded at which addresses to maintain the process-OS semantics. The OS controls the page table entries and is also notified on a page fault. This CPU design allows the OS to transparently do its management while the

Figure 1: Problem Setting. Process executing on untrusted OS.

hardware preserves the confidentiality and integrity of the process memory content. For example, if there are not many concurrent processes executing, the OS may scale up the memory allocation to a process. Later, the OS may decrease the process memory when it becomes loaded with memory requests from other processes. Further, the CPU reports all the interrupts (such as page fault, general protection fault) directly to the OS. Figure 1 shows the scenario in enclaved execution, wherein the untrusted OS can use 2 interfaces: `allocate` and `de-allocate` to directly change the page table for allocating or deallocating process pages respectively. Many systems guarantee secure execution of processes in presence of untrusted OSes, either at the hardware or software level. Execution of processes in such isolated environments is referred to as cloaked execution [18], enclaved execution [37], shielded execution [10], and so on depending on the underlying system. For simplicity, we refer to all of them as enclaved execution in this paper. SGX-specific details have been outlined in other works [7, 8].
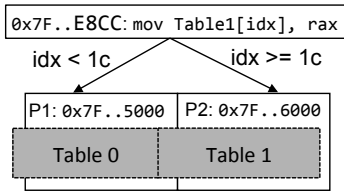
2.2 Pigeonhole Attack via Page Faults

In enclaved execution, the OS sees all the virtual addresses where the process faults [2]. This forms the primary basis of the page fault side-channel. Each page fault in the enclaved execution leaks which specific page is the process accessing at a specific point in execution time. Since the OS knows the internal structure of the program such as the layout of the binary, `mmap`-ed pages, stack, heap, library addresses and so on, the OS can profile the execution of the program and observe the page fault pattern. In fact it can invoke and execute the enclave application for a large number of inputs in offline mode to record the corresponding page fault patterns. At runtime, the OS can observe the page fault pattern for the user input and map it to its pre-computed database, thus learning the sensitive input. The remaining question is, what degree of control does the OS have on the channel capacity?
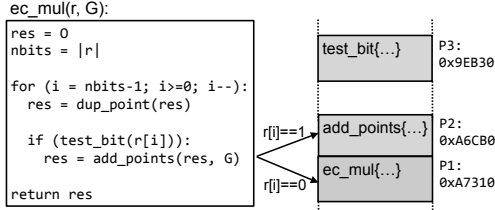
An adversarial OS that is actively misusing this side-channel always aims to maximize the page faults and extract information for a given input. On the other hand, applications often follow temporal and spatial locality of reference and thus do not incur many page faults during benign execution. Thus, the information leaked via the benign page faults from the enclave is not significant. However, note that the adversarial OS controls the process page tables and decides which virtual pages are to be loaded in the physical memory at a given point. To perpetrate the pigeonhole attack, the OS allocates only three pages at most to the program at a particular moment — the code page, the source address and the destination address [3]. Lets call this as a pigeonhole set. Thus, any subsequent instructions that access any other page (either code or data) will fall

---

[2] In our model, the trusted CPU or hypervisor only reports the base address of the faulting page while masking the offset within the page (unlike in InkTag [28]).

[3] An x86 instruction accesses at most 3 address locations.

Figure 2: Attack via input dependent data page access in AES. The data lookup to either P_1 or P_2 is decided by secret.

Figure 3: Attack via input dependent control page access in EdDSA. The control to either P_1 or P_2 is dependent on secret.

out of the pigeonhole set resulting in a page fault [4]. The faulting address of this instruction reveals what the process is trying to access. In most applications, a large fraction of memory accesses patterns are defined by the input. To extract the information about this input, the OS can pre-empt the process by inducing a page fault on nearly every instruction. Our analysis shows that empirically, every 10th code / data access crosses page boundaries on an average in standard Linux binaries [5]. This implies that the OS can single step the enclaved execution at the granularity of 10 instructions by forcing a page fault and make observations about the virtual address access patterns. Thus, by resorting to this extremity the OS achieves the maximum leakage possible via the page fault channel.

2.3 Attack Examples

A pigeonhole attack can manifest in any application running in an enclave. In this work, we limit our examples to cryptographic implementations for two reasons. First, even a minimalistic enclave will at least execute these routines for network handshake, session establishment and so on. For example, SGX applications such as one-time password generators, secure enterprise rights management and secure video conferencing use an enclave for the TLS connections and other cryptographic functions on sensitive data [27]. Second, the previous work does not study the leakage via page faults in cryptographic routines since they are assumed to be already hardened against other side-channel attacks such as timing and power consumption. On the contrary, we show that cache hardening and memory encryption is not enough. This is because caches are accessed by lower address bits while pages are accessed by higher order bits. Only masking lower order bits does not necessarily mask the page access order. Let us take a look at two representative examples to demonstrate real pigeonhole attacks.

Input Dependent Data Page Access. We choose a real example of AES from the Libgcrypt v1.6.3 compiled with `gcc` v4.8.2 on Linux system. In this example, the adversary can learn information equivalent to 25 bits of entropy about the input secret key. Note that the best known purely cryptanalytic attack for AES leak information equivalent to 2-3 bits about the key [12]. Any leakage beyond that is a serious amount of leakage. A typical AES encryption routine involves multiple S-Box lookups. This step is used to map an

input index to a non-linear value, followed by the MixColumn step. In the Libgcrypt implementation of AES, the lookup tables are designed to contain both S-box values as well as pre-computed values for MixColumns transform for optimization [2]. There are four such tables ($Table_0$ to $Table_3$) which are used in table look-ups at various rounds of encryption process. All the lookup operations in the first round take in a byte of the secret input key, XOR it with the plain text (which can be set to 0s) and emit a corresponding value in the table. Each of these tables comprise of 256 entries and are statically loaded based on the compiler-generated layout. In our example, $Table_1$ and $Table_3$ cross page boundaries. Specifically, indexes below `0x1C` are in first page (P_1) while the indexes from `0x1C` to `0xFF` are in second page (P_2). Figure 2 shows the snapshot of the virtual address space of AES, where $Table_1$ is loaded. During an enclaved execution, the process will exhibit page access profile depending on the input secret key and the plain text. The adversary adaptively selects the plain text and observes the page faults to learn the secret key. For example, lets say the key is `0x1A3E0946` and the adversary choses the plain text to be `0x00000000`. Then the resulting XOR is `0x1A3E0946`, and the page access profile will be $[P_1 P_2 P_1 P_2]$. An adversarial OS observing these page faults knows if the enclave is accessing page P_1 or P_2. Thus, for each access, this information reduces the OSes uncertainty from 256 choices to either 28 or 228 choices. In case of AES, these two portions of the table are accessed 4 times each in every round for a 128 / 196 / 256-bit key. The OS can adaptively execute the process for different known plain texts and observe the access page access profile across multiple runs. This amounts to an information theoretic leakage of 25 bits in just the first of the total 10 / 12 / 14 rounds of AES [6].

Input Dependent Code Page Access. As a second example, consider EdDSA which is an elliptic curve using with twisted Edward curve and is used in GnuPG and SSL. In EdDSA signing algorithm [11], the main ingredient is a randomly chosen scalar value r which forms the session key. The value of r is private and if leaked it can be used to forge a signature for any arbitrary message. We show how the adversary can use pigeonhole attacks to completely leak the private value r. Figure 3 shows a code snippet and the page layout for the scalar point multiplication routine of Libgcrypt implementation compiled with `gcc` v4.8.2. It takes in an integer scalar (r in this case), a point (G), and sets the result to the resulting point. The multiplication is implemented by repeated addition — for each bit in the scalar, the routine checks the value and decides if it needs to perform an addition or not. The main routine (`ec_mul`), the sub-routines for duplication (`dup_point`) and testing the bit (`test_bit`) are located in three different pages denoted as P_1, P_2, P_3 respectively. Interestingly, the addition subroutine (`add_points`) is located in pages P_1 and P_2. A page profile satisfying a regular expression $[P_1 \ P_2 \ P_1 \ P_3 \ P_1 \ (P_1 P_2)^*]$ implies a bit value 1 and $[P_1 \ P_2 \ P_1 \ P_3 \ P_1]$ implies a 0 bit value. Essentially, the OS can learn the exact value of the random integer scalar r picked by the process. This amounts to a total leakage of the secret, and in fact enables the OS to forge signatures on behalf of the enclave.

We have experimentally confirmed both of the above attacks. We demonstrate more attacks on cryptographic implementations of Libgcrypt and OpenSSL in Section 6.1. These attacks may apply to cloud server platforms [10, 15, 21, 22, 41, 49, 54, 55].

---

[4]Note that the process does not suffer denial of service, only its progress is slowed down due to excessive page faults.

[5]We tested CoreUtils utilities in malicious runs under random inputs.

[6]AES first round uses the first 128 bits of a 128 / 196 / 256-bit key. Initial uncertainty of OS = 2^{128}. With pigeonhole attack, the OS knows for the 64 bits if the index is less than `0x1c`. So, final uncertainty = $2^{64} \times 28^8$. Information leakage (in bits) = $log_2(2^{128} - (2^{64} \times 28^8)) = 25.54 \sim 25$ bits [45].

3. OVERVIEW

The malicious OS can use pigeonhole attacks to observe input-dependent memory accesses and learn information about input program secrets. We now discuss our approach to prevent this leakage.

3.1 Security Definitions & Assumptions

Lets represent an enclave program P that computes on inputs I to produce output O as $(P, I) \mapsto O$, such that both I and O are secret and are encrypted in RAM. In case of enclaved execution, the adversary can observe the sequence of page faults. We term this knowledge of the adversary as the *page access profile*. Note that each observed profile is specific to an input to the program, and is defined as:

Definition (Page Access Profile.) *For a given program P and a single input I, the page access profile $\overrightarrow{VP}_I$ is a vector of tuples $\langle VP_i \rangle$, where VP_i is the virtual page number of the ith page fault observed by the OS.*

To model the security, we compare the execution of a program on a real enclaved system with its execution on an "ideal" system. The ideal system is one which has infinite private memory and therefore the program execution doesn't raise faults. On the other hand, the real system has limited memory and the enclave will incur page faults during its execution. Specifically, we define these two models as follows:

- ∞-memory Enclave Model ($M_{\infty-model}$). The enclaved execution of program on a system with an unbounded physical memory such that the page access profile is $\emptyset$.
- Bounded-memory Enclave Model ($M_{B-model}$). Enclaved Execution of program such that for any instruction in the program, the enclave has the least number of pages required for executing that instructions [7].

Definition (Page Access Profile Distinguishability) *Given a program $(P, I) \rightarrow O$, we say P exhibits page access profile distinguishability if there exists an efficient adversary $\mathcal{A}$ such that $\exists I_0, I_1 \in I$ and $b \in \{0, 1\}$, for which the advantage:*
$$Adv(\mathcal{A}) = |Pr[Exp(\overrightarrow{VP}_{I_{b=0}}) = 1] - Pr[Exp(\overrightarrow{VP}_{I_{b=1}}) = 1]|$$
is non-negligible.

If a probabilistic polynomial time-bounded adversary can distinguish the execution of the program for two different inputs by purely observing the page access profile, then the program exhibits page access profile distinguishability. A safe program exhibits no leakage via the page fault channels; we define page-fault obliviousness as a security property of a program as follows:

Definition (PF-obliviousness) *Given a program P w.r.t. inputs I, the PF-obliviousness states that if there exists an efficient adversary $\mathcal{A}$ which can distinguish $(\overrightarrow{VP}_{I_0}, \overrightarrow{VP}_{I_1})$ for $\exists I_0, I_1 \in \vec{I}$ in the $M_{B-model}$, then there exists an adversary $\mathcal{A}'$ which can distinguish I_0, I_1 in the $M_{\infty-model}$.*

Our definition is a relative guarantee — it states that any information that the adversary learns by observing the execution of program on a bounded private memory, can always be learned by observing the execution even on an unbounded memory (for e.g., the total runtime of the program). Such information leaked can be gleaned even without the page fault channel. Our defense does not

---

[7] In our case it is at most three pages, which is the maximum number of pages required to execute any Intel x86 instruction.

---

provide any absolute guarantees against all possible side-channels. If there are additional side channels in a PF-oblivious program, they can be eliminated with orthogonal defenses.

Scope and Assumptions. Our work considers a software-based adversary running at ring-0; all hardware is assumed to be trusted. Further, the following challenges are beyond the goals of this work:

- **A1.** Our attacks and defenses are independent of other side-channels such as time, power consumption, cache latencies, and minor execution time differences between two different memory access instructions that raise no faults. If such a difference is discernible, then we can show that they provides a source of advantage even in an execution with no page faults (∞-model). Application developers can deploy orthogonal defenses to prevent against these side-channels [53]. Our defenses do not prevent information leakage via untrusted I/O, system-call, and filesystem channels [17].
- **A2.** Once a page has been allocated to the enclave, the OS can take it away only on a memory fault. We do not consider the case where the OS removes enclave pages via a timer-based pre-emption, since the adversary's clock granularity is much coarser in this case and likely yields a negligible advantage.

3.2 Problem & Approach Overview

Problem Statement. Given a program P and set of secret inputs I, we seek a program transformation $T: P \mapsto P'$ such that the transformed program P' satisfies PF-obliviousness with respect to all possible values of I.

Consider a program executing on sensitive input. The execution path of such a program can be defined by the sequence of true and false branches taken at the conditional statements encountered during the execution. Each set of straight-line instructions executed and corresponding data accessed between the branching condition statements can be viewed as an *execution block*. Let us assume that each execution block has the same number of memory accesses and by assumption A1 each memory access takes approximately same amount of time. Then, all such paths of a program can be represented using a tree, say the *execution tree* such that each node in the tree is an execution block connected by branch edges. For example, the function foo() in Figure 4 (a) has 3 execution paths in the execution tree shown in Figure 4 (b). Each of the paths a, b, c can be executed by running the program on the inputs (x = 4, y = 2), (x = 8, y = 9) and (x = 6, y = 5) respectively.
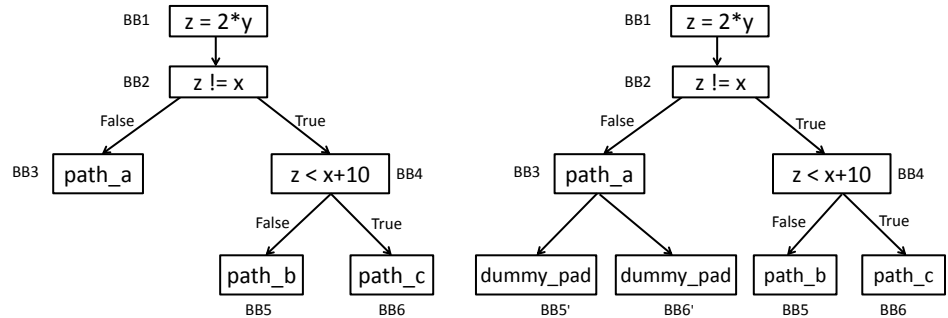
Page access profile is inherently input dependent, so anyone who observes the page access profile can extract bits of information about the input. However, if the page access profile remains the same irrespective of the input, then the leakage via page fault channel will drop to zero [35]. We call this transformation strategy as determinising the page access profile. We adopt this strategy and enforce a deterministic page access profile for possible paths in the program execution. The enclaved execution always sequentially accesses all the code and data pages that can be used at a particular memory-bound instruction for each execution. In our example, Figure 4, we will access both BB3 as well as BB4 irrespective of the branching condition. Similarly, we also apply it at level 4, so that the complete program path is BB1, BB2, BB3, BB4, BB5', BB6', BB5, BB6 for all inputs. Thus, deterministic execution makes one real access and several fake accesses to determinise the page access profile. It is easy to see that under any input the execution exhibits the same page access profile.

The challenge that remains is: how to execute such fake accesses while still doing the actual intended computations. We present a simple mechanism to achieve this. First we use the program's exe-

```
1   foo (int x, int y)
2   {
3     z = 2 * y
4     if (z != x)
5     {
6       if (z < x + 10)
7         path_c()
8       else
9         path_b()
10    }
11    else
12      path_a()
13  }
```

Figure 4: (a) Code snippet for example function `foo` where `x` and `y` are secret. (b) Unbalanced execution tree. (c) Corresponding balanced execution tree.

cution tree to identify what are all the code and data pages that are used at each level of the tree for all possible inputs (BB3, BB4 at level 3 in our example). This gives us the set of pages for replicated-access. Next, we use a multiplexing mechanism to load-and-execute the correct execution block. To achieve this, we break each code block execution into a fetch step and an execute step. In the fetch step, all the execution blocks at the same level in the execution tree are fetched from memory sequentially. In the execute step the multiplexer will select the real block and execute it as-is. In our example, for (x = 4, y = 2), the multiplexer will fetch all blocks but execute only BB3 at level 3, and for (x = 8, y = 9) or (x = 6, y = 5), the multiplexer will execute BB4.
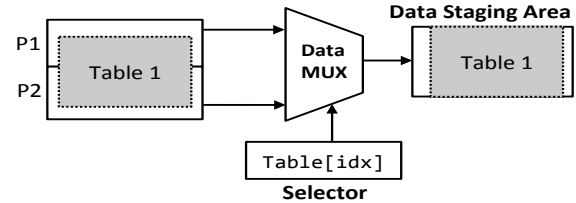
4. DESIGN

There can be several ways for determinising the page access profile; selecting the best transformation is an optimization problem. We discuss one such transformation which can be applied generically and then present the program-specific transformations which incur lower costs (Section 5).

4.1 Setup

It is simple to adapt the standard notion of basic blocks to our notion of execution blocks. In our example code snippet in Figure 4 (a), we have 6 such execution blocks BB1 to BB6. In case of BB1, the code page $\mathbb{C}$ will comprise of virtual page address of the statement z = 2 * y, and data pages $\mathbb{D}$ will have virtual page address of variables z and y.

Note that the execution tree in Figure 4 (b) is unbalanced, i.e., the depth of the tree is not constant for all possible paths in the program. This imbalance in itself leaks information about the input to an adversary even without pigeonhole attacks simply by observing the function start-to-end time. For example, the first path (path_a) in Figure 4 (b) is of depth 2 and is only taken when value of z equals value of x. If the adversary can try all possible values of secret, then the tree depth becomes an oracle to check if the guess is correct. To capture the information leaked strictly via the page fault channel, we limit our scope to balanced execution tree. If the tree is unbalanced, then the input space is partitioned into sets which are distinguishable in the original program in the ∞-model. Since we limit our scope to achieving indistinguishability relative to ∞-model, we safely assume a balanced execution tree as shown in Figure 4 (c) [38]. Techniques such as loop unrolling, block size balancing with memory access and NOP padding can be used to balance the tree depth and block sizes [19]. In our experience, cryptographic routines which are hardened against timing and cache side-channels generally exhibit balanced execution trees. For the set of programs in our study, if necessary, we perform a preparation step manually to balance the execution tree explicitly.

Even after the execution tree is balanced, the pigeonholing adver-

Figure 5: Deterministic Multiplexing for data access. The multiplexer accesses the correct offset in the staging area.
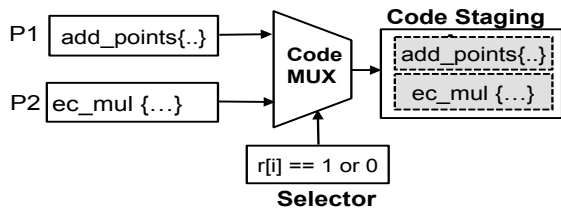
sary knows the sequence of the execution blocks that were executed for a given input only by observing page faults. For example, lets assume that the execution blocks BB5 and BB6 are in two different pages P_1 and P_2 respectively. Then the result of the branching condition $z < x + 10$ will either cause a page fault for P_1 or P_2, revealing bit of information about the sensitive input x and y. Given a balanced execution tree, we design a transformation function to make the page access profile independent of the input [35].

4.2 Deterministic Multiplexing

We now discuss a concrete design of our transformation namely deterministic multiplexing and demonstrate how it can be supported to transform legacy C / C++ applications in the current compiler infrastructure.

Basic Multiplexing. In the fetch phase, we copy the code blocks at the same level of the execution tree to a temporary page — the code staging area (SA_{code}). All data that may be used by each of these sensitive code blocks is copied to a separate temporary page — the data staging area (SA_{data}). Then in the execution phase, we use an access multiplexer which selects the correct code and data blocks and executes it (by jumping to it). At the end of the sensitive execution, the content from data staging area is then pushed back to the actual addresses. If the execution changes any data in the staging area, the new values are updated. The rest of the values are just copied back unchanged. Note that all these operations are done in a sequence in the staging area (one code page). Thus this execution is atomic — no page faults can occur between them. From an adversarial viewpoint, the execution is performed within the boundary of single code and single data page. So all that the adversary can see is the same sequence of page faults for any input. Thus our multiplexed fetch and execute mechanism ensures that the OS cannot determine which code and data block was actually used within the staging area.

Example. For our AES case, we apply deterministic multiplexing and copy the data table T_3 to staging area (See Figure 5). Each data access now incurs 2 data page copies and a code page copy followed by multiplexed accesses. Similarly for EdDSA, we can multiplex the called functions into SA_{code} (See Figure 6). This asserts that the OS cannot differentiate whether the true or the false

Figure 6: Deterministic Multiplexing for code page access. The multiplexer executes the correct function in the staging area.

branch was executed, by looking at the page access profile. Thus, in both the cases the OS can observe the fetch and execute operations only at the page granularity. It cannot determine which of the fetch or execution operations is real and which is replicated.

Compacted Multiplexing. In the multiplexing mechanism, it is important that both SA_{code} and SA_{data} must fit in a single page each to prevent information leakage. For ensuring this, we specifically pick a block size such that at any given level in the execution tree, all the blocks and the corresponding data always fit in a single page. However, there are cases where the execution tree is deep and has large number of blocks (total size of more than 4096 bytes) at a certain level. This results in a multi-page staging area. To address this, we use a compaction scheme to fit the staging area in a single page. Specifically, in the fetch phase we create a dummy (not real) block address in the staging area. The blocks which are not going to be executed are saved at this dummy location during the fetch step. Each new block from the execution tree overwrites (overlap) the same location. Only the real block (which will be executed) is copied in a non-overlapping address in the page. We term this as a *smart copy* because each copy operation writes to either dummy or real page-offset in the staging area. The adversary OS does not see the offset of the faulting address, and hence cannot distinguish a dummy vs. a real copy. Thus the staging area always fits in a single page. The semantics of the execute phase are unchanged.

4.3 Compiler-enforced Transformations

We build our design into the compiler tool chain which works on a subset of C / C++ programs. Given a program, the programmer manually annotates the source code to demarcate the secret input to the program and specifies the size of input with respect to which the transformation should guarantee PF-obliviousness. Specifically, he manually adds compiler directive `begin_pf_sensitive` and `end_pf_sensitive` to mark the start and end of sensitive code and data. For example, the developer can mark the encryption routine, decryption routine, key derivation, key, nonce, and so on as secret. Our tool comprises of analysis and transformation steps to enforce deterministic multiplexing which are discussed next.

Identifying Sensitive Code and Data. In the first step, our compiler front-end parses the source code and identifies the programmer added directives. It then performs a static analysis which transitively marks all the instructions and variables within the lexical scope of programmer-marked sensitive code as `high`. Non-sensitive instructions and variables are marked as `low`. At the end of the phase, each instruction and variable in the code has a sensitivity tag (`high` or `low`).

Determinising the Page-layout. Next, our tool performs an analysis to decide the new virtual address layout for the sensitive data and code (marked as `high`) for placing them in the staging area. The initial step is to identify the existing execution tree of the sensitive code. To achieve this, we create a super-CFG wherein each function call is substituted with the body of the function and all the bounded loops are unrolled. This creates an execution tree such that all the sensitive execution blocks are identified. We seek a

mapping $\Gamma : B \mapsto \mathcal{L}$ such that all the execution blocks at the same level in the execution tree are relocated to the same virtual page address. There are multiple possible Γ mappings which yield acceptable layouts, but our goal is to select the one where the code and data staging areas always fit in a single page. We first try to use the basic multiplexing for arranging the blocks if the total size of all the blocks at a level is less than 4096 bytes. If the size of the required staging area exceeds one page, then we resort to compacted multiplexing (See Section 4.2).

Instruction Rewriting. The last step of transformation comprises of: (a) Adding logic for multiplexing (b) Adding prologue-epilogue before and after the multiplexing to move the code / data to and from staging area. Next, we rewrite the instructions to introduce replicated accesses to data pages, and instrument each execution block with a call to the code multiplexing logic as described in Section 4.2. Finally, we add prologue and epilogue before and after each execution block at each CFG level.

Example. In case of EdDSA, we manually add compiler pragmas to mark the user key variable and the signing routine as sensitive. Our analysis phase identifies 31 functions, 701 execution blocks, 178 variables as sensitive. It also collects information about the call graph, function CFG and access type (read or write) of the variables. After the analysis, our tool calculates (a) the staging area to be created in first function `ec_mul` just before the first access to the key (b) layout of the data staging area such that all the variables fit in one page (c) the alignment of the execution block in the staging area, (d) the new addresses of the sensitive variables used in these execution block, and (e) instructions which are to be updated for accessing the staging area. Finally, we add code for preparing the staging area and instrument the code instructions to use the data staging area values.

Security Invariant. The above compiler transformation ensures that for the output program, all the execution blocks at the same level in the execution tree are mapped to same ordered list of virtual address locations. Thus for all the inputs, the program exhibits the same page access profile hence satisfying our PF-obliviousness property.
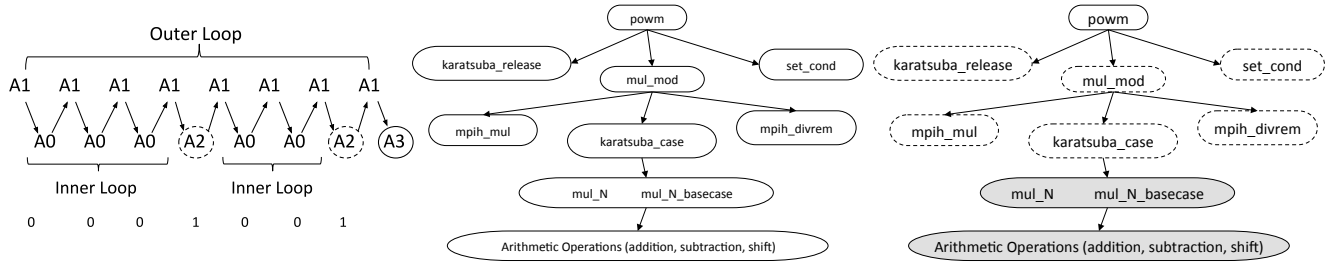
5. DEVELOPER-AIDED OPTIMIZATIONS

Apart from the compiler enforced transformation, we have manually confirmed other strategies to make programs PF-oblivious. We discuss these strategies which allow developer-aided optimizations. In the future, our compiler can be extended to search and apply these optimization strategies automatically.

5.1 Exploiting Data Locality

The main reason that input-dependent data accesses leak information in pigeonhole attacks is that the data being accessed is split across multiple pages. In all such cases, the deterministic multiplexing repetitively copies data to and fro between the staging area and the actual data locations. There are two key observations specific to these cases.

O1: Eliminating copy operations for read-only data. We observe that most of the table lookup operations are on pre-computed data and the code does not modify the table entries during the entire execution. Since these sensitive data blocks are used only in read operations, we can fetch them into SA_{data} and discard them after the code block executes. This saves a copy-back operation per code block. Moreover, if the next code block in the execution tree uses the same data blocks which already exist in SA_{data}, then we need not copy them to SA_{data}. This save all the copy operations after the data is fetched into the SA_{data} for the first time. In case of

Figure 7: (a) Simplified page access profile for `powm` (Window size = 1) where A0, A1, A2, A3 denote transitions between mul_mod(), powm(), set_cond() and karatsuba_release() respectively (b) Call graph before enforcing deterministic multiplexing. (c) Alignment after optimization (O4) where dotted and shaded functions are moved to separate code staging pages.

Algorithm 1 Libgcrypt modular exponentiation (`powm`).

INPUT: Three integers g, d and p where $d_1...d_n$ is the binary representation of d.
OUTPUT: $a \equiv g^d \ (mod \ p)$.

  procedure POWM (g, d, p) ▷ P1
     $w \leftarrow$ GET_WINDOW_SIZE$(d), g_0 \leftarrow 1, g_1 \leftarrow g, g_2 \leftarrow g^2$
     for $i \leftarrow 1$ to $2^{w-1} - 1$ **do**   ▷ Precomputation
        $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2 \ \texttt{mul\_mod} \ p$
     end for
     $a \leftarrow 1, j \leftarrow 0$
     while $d \neq 0$ **do**   ▷ Outer loop
        $j \leftarrow j+$ COUNT_LEADING_ZEROS(d)
        $d \leftarrow$ SHIFT_LEFT(d, j)
        for $i \leftarrow 1$ to $j + w$ **do**   ▷ Inner Loop
           $a \leftarrow a \cdot a \ \texttt{mul\_mod} \ p$ ▷ P2
        end for
        $t \leftarrow d_1...d_w;$
        $j \leftarrow$ COUNT_TRAILING_ZEROS(t)
        $u \leftarrow$ SHIFT_RIGHT(t, j)
        $g_u \leftarrow$ FETCH_POWER(set_cond(u))   ▷ P3
        $a \leftarrow a \cdot g_u \ \texttt{mul\_mod} \ p$ ▷ P2
        $d \leftarrow$ SHIFT_LEFT(d, w)
     end while
  end procedure

AES, we require only two operation to copy $Table_1$ from P_1 and P_2 to SA_{data}. We can apply the same strategy to $Table_3$, so that the entire execution needs only four copy operations.

O2: Page Realignment. All the data blocks which are spread across page boundaries (specifically, S-Boxes) can be grouped together and realigned at the start of the page. This ensures that the set of sensitive data pages is minimum for the entire execution. In the context of AES example, both $Table_1$ and $Table_3$ cross the page boundary and use 3 pages. They can be aligned to page boundary and fit in 2 pages. Thus for deterministic multiplexing, the patch will incur only two copy operations in total.

Note that the above strategies are safe and respect the security invariant (Section 4.3) because all the eliminations are independent of the input and thus the reduction in the copy operations affects all the inputs uniformly.

5.2 Exploiting Code Locality

In case of input-dependent control transfers, automatically determinising the control flow results in a high number of multiplexing operations. To address this short-coming we propose a set of strategies specific to the type of pigeonhole attacks, which reduces the overheads to an acceptable range. We take the example of `powm` and demonstrate our strategies.

Algorithm 1 shows the code structure and data access pattern for the `powm` example. In the Libgcrypt implementation, the actual function body (`powm`), the multiplication function (`mul_mod`) and the table lookup function (`set_cond`) are located in three separate pages say P_1, P_2, P_3 respectively. Hence, the leakage from `powm` is due to the different fault patterns generated from calls to `mul_mod` and `set_cond` functions. Figure 7 (a) shows the page fault pattern for `powm` with respect to these functions and Figure 7

(b) shows the function arrangement for `powm`. Let us consider the implementations of deterministic multiplexing in Section 4.3 that make calls to both these functions indistinguishable. For this, we generate the call graphs of both functions which identifies the set of sensitive functions are to be masked. For each call to any of these sensitive function, we perform a multiplexing operation. It iterates over the set of these sensitive functions in a deterministic manner and copies all the blocks to SA_{code}. The multiplexer then selects the correct block and executes it. In case of `powm`, we move `powm`, `mul_mod` and `set_cond` to the staging area. This implementation of Section 4.3 incurs an overhead of $4000\times$, which is prohibitive. We discuss our strategies in the context of this example to describe the reasoning for the optimization.
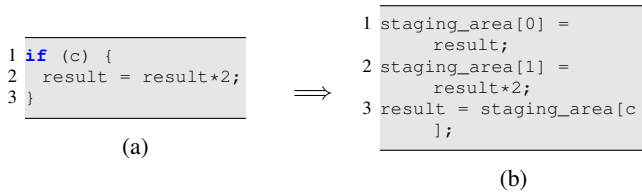
O3A: Level Merging. The dominating factor in the deterministic multiplexing is the number of copy and multiplexing operations at each level in the execution tree. We observe that by the virtue of code locality, code blocks across multiple levels can be merged together in a single level. Specifically, we place the code blocks such that the caller and callee function are contained within a page. For example, consider 3 code blocks a, b, c located in three separate pages. The call graph is such that c is called by both a and b. If total size of a, b, c put together is less than a page (4096 bytes), then we can re-arrange the code such that all three of them fit in a single page. In terms of the execution tree, it means that we fold the sub-tree to a single code block.

O3B: Level Merging via Cloning. The above strategy will not work in cases where the code blocks in a sub-tree cannot fit in a single page. To address this, we use code replication i.e., we make copies of shared code block in multiple pages. In our example, if blocks a, b, c cannot fit into a single page, we rearrange and replicate the block c in both P2 and P3. After replication, a control-flow to c from neither a nor b will incur a page fault. For `powm`, we split the `mul_mod` into 2 pages and replicate the code for `set_cond`. Thus, call to from `powm` to `set_cond` can be resolved to either of the pages. It is easy to see that since security guarantee of the compiler-transformed code holds true for the un-optimized program execution tree, it trivially holds true for the reduced trees in the above two cases because O3A-B are replicating or merging the page access uniformly for all the inputs.

O4: MUX Elimination. Our next optimization is based on the insight to eliminate the cost of the multiplexing operation itself by rearranging the code blocks. To achieve this, we place the code blocks in the virtual pages to form an execution tree such that all the transitions from one level to the other exhibit the same page fault. This eliminates the multiplexing step altogether. In the above example of blocks a, b and c, we place a and b into one page and c into another. Thus, the control-flow from both a and b to c will page fault in both the cases or none at all. We can chain successive transitions for multiple levels in the tree, such that all the blocks in

Table 1: Summary of cryptographic implementations susceptible to pigeonhole attacks. * denotes that the leakage depends on the input. $[a:b]$ denotes the split of S-Box where a and b is percentage of table content across two different pages.

Library	Algo	Secret Entity	Vulnerable Routine	Vulnerable Portion (gcc)	Vulnerable Portion (llvm)	Input Bits	Leakage (gcc)	%	Leakage (llvm)	%
Libgcrypt (v1.6.3)	AES	Symmetric key	Encryption	2 T-Boxes [11:89]	2 T-Boxes [50:50]	128, 192, 256	25	14.01	8	4.51
	CAST5		Key Generation	1 S-Box [38:62]	1 S-Box [48:52]	128	3	2.34	2	1.56
	SEED			1 SS-Box [88:12]	1 SS-Box [27:73]	128	*6	4.69	*4	3.13
	Stribog	Password used in PBKDF2	Key Derivation	4 S-Boxes [51:49]	4 S-Boxes [51:49]	512	32	6.25	32	6.25
	Tiger			2 S-Boxes [53:47]	2 S-Boxes [58:42]	512	4	0.78	4	0.78
	Whrilpool			4 S-Boxes [45:55]	4 S-Boxes [52:48]	512	32	6.25	32	6.25
	EdDSA	Session key (hence Private key)	Signing	ec_mul	ec_mul	512	512	100	512	100
	DSA	Private key	Key generation	powm	powm	256	*160	62.50	*160	62.50
	Elgamal		Modular exponentiation			400	*238	59.50	*238	59.50
	RSA	Private key mod (p-1)				2048	*1245	60.79	*1245	60.79
		Private key mod (q-1)				2048	*1247	60.89	*1247	60.89
OpenSSL (v1.0.2)	CAST5	Symmetric key	Key generation	1 S-Box [55:45]	1 S-Box [84:16]	128	2	1.56	*6	4.69
	SEED			1 SS-Box [47:53]	1 SS-Box [67:33]	128	16	12.50	*6	4.69
							Average	**28.02**		**25.64**


```
1 if (c) {
2   result = result*2;
3 }
```
(a)

```
1 staging_area[0] =
     result;
2 staging_area[1] =
     result*2;
3 result = staging_area[c
     ];
```
(b)

Figure 8: Example for O5: Control-to-Data Dependency Transformation.

next level are always placed in a different pages. Figure 7 (c) shows the arrangement of functions in the code staging area such that the functions are grouped together in the same page. We apply this to the execution sub-tree of mul_mod function in powm.

5.3 Peephole Optimizations

We apply a local peephole optimization to convert the control-dependent code to data-dependency which eliminates the need for code multiplexing.

O5: Control-to-Data Dependency Transformation. Masking data page accesses is easier and hence we can convert the input dependent code accesses to data accesses. For example, the if-condition on value of c in Figure 8 (a) can be rewritten as Figure 8 (b). Specifically, we perform an if-conversion such that the code is always executed and the condition is used to decide whether to retain the results or discard them [20]. In the case of EdDSA, we first fetch the value of res into SA_{data} (Refer to Figure 3 for code details). We execute add_points unconditionally and we use test_bit as a selector to decide if the value in SA_{data} is to be used. In the case where test_bit returns true, the actual res in SA_{data} is used in the operation and is updated, else it is discarded. The page fault pattern will be deterministic since add_points will be executed on all iterations of the loop and the operand of the function is always from SA_{data}. This optimization is applied before the compiler transformation, hence its security follows from the basic security invariant outlined in Section 4.3.

All our strategies **O1-O5** are supported by our compiler augmentation with programmer directives. Note that, our optimization strategies are sound — the compiler still asserts that the transformation preserves the PF-obliviousness of the program. We discuss the empirical effectiveness of these strategies in Section 6.4.

6. EVALUATION

Evaluation Goals. We aim to evaluate the effectiveness of our proposed solutions for following main goals:

- Does our defense apply to all of our case studies?
- What are the performance trade-offs of our defense?
- How much performance improvements do developer-assisted transformation offer?

Platform. SGX hardware is not yet fully rolled out and is not publicly available for experimentation. As a recourse, we conduct all our experiments on PODARCH [44]; a system similar to previous hypervisor solutions [18] and conceptually similar to SGX. Our machine is a Dell Latitude 6430u host, configured with Intel(R) Core(TM) i7-3687U 2.10GHz CPU, 8GB RAM. We configure PO-DARCH with one CPU, 2GB RAM and 64-bit Linux 3.2.53 Kernel on Debian Jessie for all the experiments. We use LLVM v3.4 with the default optimization flags for compiling our vanilla and patched case studies. All the results are averaged over five runs.

6.1 Case Studies

Selection Criteria. Our defense techniques can be applied to an application if it satisfies the conditions of balanced-execution tree. We checked the programs FreeType, Hunspell, and libjpeg discussed in [51] but they exhibit unbalanced execution tree. Transforming these programs to exhibit balanced execution tree causes an unacceptable performance loss, even without our defense [48]. Hence, we limit our evaluation to cryptographic implementations.

We present our results from the study of a general purpose cryptographic library Libgcrypt v1.6.3 which is used in GnuPG and a SSL implementation library OpenSSL v1.0.2 [2,3,5]. Table 1 summarizes the results of our study. Interested readers can refer to the extended version of the paper for the experimental details of each case study attack [43]. We analyzed the programs compiled with the two most-used compiler toolchains: gcc v4.8.2 and LLVM v3.4. For both the compilers, we statically compiled all our programs with the default optimization and security flags for compilation. Of the 24 routines we analyze in total from both the libraries, 10 routines are vulnerable to pigeonhole attacks on both the compilers. Since our emphasis is not on the attacks, we highlight only the important findings below.

Table 2: Performance Summary. Columns 3, 5, 12 denotes the number of page faults incurred at runtime. Columns 10 and 14 represent the total percentage overhead. $>$ symbol denotes the program did not complete within 10 hours after which we terminated it. A negative overhead means patched code executes faster than the baseline. Tc and Te denote the time spent in preparing the staging area and actual execution respectively.

Library	Cases	Vanilla		Unoptimized Deterministic Multiplexing						Optimized Deterministic Multiplexing			
		PF	T (ms)	PF	Tc (ms)	Te (ms)	T (ms)	Tc / T (%)	Ovh (%)	Opt	PF	T (ms)	Ovh (%)
Libgcrypt (v1.6.3)	AES	4 - 5	4.711	4	7.357	4.013	11.370	64.70	**141.35**	O1,O2	4	4.566	**-3.08**
	CAST5	2	3.435	2	8.050	2.578	10.629	75.74	**209.47**	O1,O2	1	3.086	**-10.15**
	EdDSA	0	10498.674	0					**>300000**	O5	0	13566.122	**29.22**
	powm	0	5318.501	0	—	>10 hrs	—		**>400000**	O3	0	399614.244	7413.66
										O4	0	5513.712	**3.67**
	SEED	2	1.377	2	4.559	1.057	5.615	81.18	**307.79**	O1, O2	1	1.311	**-4.80**
	Stribog	5	27.397	5	329.743	10.836	340.579	96.82	**1143.13**	O1, O2	4	28.563	**4.26**
	Tiger	3	2.020	3	64.482	0.546	65.029	99.16	**3119.69**	O1, O2	2	1.840	**-8.89**
	Whirlpool	5	27.052	5	141.829	10.174	151.490	93.28	**459.99**	O1, O2	4	23.744	**-12.23**
OpenSSL (v1.0.2)	CAST5	2	11.249	2	17.083	8.295	25.378	67.31	**125.60**	O1, O2	1	10.623	**-3.41**
	SEED	2	3.684	2	8.998	3.737	12.734	70.66	**245.69**	O1, O2	1	3.558	**-5.57**
Average Performance Overhead									**70575.27**				**-1.10**

- **No Leakage**. In Libgcrypt implementations of Blowfish, Camellia, DES, 3DES, IDEA, RC5, Serpent, Twofish, ECDSA, and SHA512, all the input-dependent code and data memory accesses are confined within a page for the sensitive portions. Similarly AES, Blowfish, Camellia, DES, 3DES, IDEA, RC5, Serpent, Twofish, DSA, RSA, and SHA512 in OpenSSL do not exhibit leakage via page fault side channel.
- **Leakage via input dependent code page access**. In Libgcrypt, EdDSA and `powm` exhibit input dependent code access across pages and are vulnerable to pigeonhole attacks. The `powm` function is used in ElGamal, DSA and RSA which leaks bits of information about the secret exponents.
- **Leakage via input dependent data page access**. In case of AES, CAST5, SEED, Stribog, Tiger and Whirlpool implementations in Libgcrypt, at least one of the S-Boxes crosses page boundary and leaks information about the secret inputs. Similarly, implementations of CAST5 and SEED in OpenSSL are also vulnerable.

6.2 Application to Case Studies

We transform the 8 Libgcrypt and 2 OpenSSL vulnerable implementations of our case studies in our evaluation.

Compiler Toolchain Implementation. We implement our automation tool in LLVM 3.4 and Clang 3.4 C / C++ front-end to transform C / C++ applications [1,6]. For our case studies, we log all the analysis information which is used for the layout analysis and also to facilitate our developer-assisted improvements study. Our transformation pass applies deterministic multiplexing to the programs at the LLVM IR level.

Empirical Validation. Our applications are compiled into static binaries for testing. We run these executables on PODARCH [44] which is implemented on QEMU emulator, and only supports static linking. To test our patched applications, we execute the standard regression test-suite available with the cryptographic libraries (`make check`). To empirically validate that our defenses work, we ensure that the page fault profile of patched executions under all test inputs is indistinguishable w.r.t. page access profiles. To verify the correctness, we analyze the page fault access patterns in the transformed application using a PinTool [4] that logs all instructions and memory accesses. We have analyzed the PinTools logs and report that our deterministic multiplexing produces indistinguishable page access profiles for all regression and test inputs.

6.3 Performance Evaluation

Normalized Baseline. To ensure that the choice of our evaluation platform (PODARCH) does not significantly bias the overheads, we conduct two sets of measurements. First, we run the unmodified OpenSSL and Libgcrypt implementations on PODARCH and measure the execution time. This forms the baseline for all our performance measurements. Column 3, 4 in Table 2 shows the number of page faults and the execution time for vanilla code in PODARCH. Second, to check that the overheads of our defenses are not an artifact of PODARCH, we also run our vanilla and modified binaries on native Intel CPU Intel Core i7-2600 CPU. The overheads on a native CPU are similar to that on PODARCH and deviate only within a range of 1%. This confirms that our baseline of PODARCH does not skew our experimental results significantly.

Overhead. We calculate the overhead by comparing the baseline performance of unmodified code against the execution time of the patched application functions. We use input patterns to represent the best, worst and average case executions of the application, specifically, inputs with (a) all 0s, (b) all 1s, (c) random number of 0s and 1s, and (d) all the regression tests from the built-in test-suite.

The applications patched with the deterministic multiplexing technique incurs an average overhead of $705\times$ and up to maximum overhead of $4000\times$ in case of `powm` (Column 10 in Table 2). To investigate the main sources of these overheads we measure the break-down for the fetch step and the execute step in deterministic multiplexing. We observe that the overhead is mainly dominated by the copying of data to and from the staging area in the fetch step (Column 6 and 9 in Table 2), and accounts for 76.5% out of the total overhead on average. We notice that the fetch step time is especially high for cases like Stribog and Tiger where it accounts for 96.82% and 99.16% of the overhead.

6.4 Effectiveness of Optimizations

We apply the developer-assisted strategies discussed in Section 5 to experimentally validate and demonstrate their effectiveness. They reduce the average overhead from $705\times$ to -2.7% for our 10 case studies; 29.22% in the worst case. In the case of `powm`, **O3** reduces the performance overhead from $4000\times$ to $74\times$. With **O4** we completely remove memory copying for code determinization which reduces the overhead from $74\times$ to 3.67%. We apply **O1** to the 8 cases of input dependent data page access to reduce the number of

copy operations. Further we also apply **O2** to reorder the lookup table layout, such that after the developer-assisted transformations are in place, the execution incurs lower page faults. In fact, our patched version executes faster than the baseline code (as denoted by negative overhead in Column 14 in Table 2) for 7 cases. After manual inspection, this is explained because in the patched code, the lookup tables take up less number of pages which reduces the total number of page faults incurred during the execution (Column 12 in Table 2). On the other hand, in the vanilla case, the program incurs more page faults which is a costly operation. Thus, eliminating this cost results in a negative overhead. For EdDSA, we directly apply the peephole optimization **O5** which transforms the input dependent code access to data access. This reduces the overhead from $3000\times$ to 29.22%.
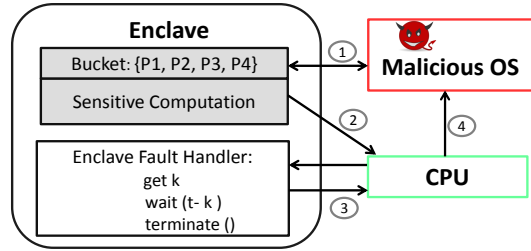
7. HARDWARE-ENABLED DEFENSES

So far we have discussed purely software solutions. Readers might wonder if pigeonhole attacks can be mitigated with hardware support. Here, we briefly discuss an alternative hardware-assisted defense which guarantees enclaved execution at an average cost of 6.77% for our benchmarks.

7.1 Our Proposal: Contractual Execution

We propose a hardware-software technique wherein the enclave is guaranteed by the hardware that certain virtual addresses will always be mapped to physical memory during its execution. The enclave application is coded optimistically assuming that the OS will always allocate specific number of physical pages to it while executing its sensitive code blocks. The enclave informs its memory requirements to the OS via a callback mechanism. These requirements act as a *contract* if the OS agrees, or else the OS can refuse to start execution of the enclave. The enclave states the set of virtual addresses explicitly to the OS before starting its sensitive computation. The CPU acts as a contract mediator and is responsible for enforcing this guarantee on the OS. We term such an execution as *contractual execution*. Note that the contract is not a hard guarantee i.e., the enclave cannot pin the pages in physical memory to launch a denial-of-service attack on the OS. In fact, the OS has the flexibility to take back pages as per its own scheduling policy. However, when the CPU observes that OS has deviated from the contract — either genuinely or by injecting random faults, it immediately reports the contract violation to the enclave. This needs two types of changes in the hardware (a) support for notifying the enclave about its own page faults and (b) guaranteeing a safe mechanism for enclave to mitigate the contract violation.

Contract Enforcement in SGX. In a traditional CPU as well as in original SGX specification [7], all page faults are reported directly to the OS without the intervention of the faulting process. Thus, the process is unaware of its own page faults. This makes it impossible for the enclave to detect pigeonhole attacks. For contractual execution, the hardware needs to report its faults to the process instead, which calls for a change in the page fault semantics. A limited amount of support is already available for this in SGX. As per the new amendments in Revision 2, SGX can now notify an enclave about its page faults by setting the `SECS.MISCSELECT.EXINFO` bit [8, 10]. When an enclave faults, the SGX hardware notifies the enclave about the fault, along with the virtual address, type of fault, the permissions of the page, register context. We can think of implementing contractual execution on SGX directly by setting the SGX configuration bit such that when there is a page fault, the enclave will be notified directly by the CPU. The benign OS is expected to respect the contract and never swap out the pages during the execution. However a malicious OS may swap out pages,

Figure 9: Contractual Execution. (1) Enclave registers a contract (2) CPU directly reports the fault to the enclave page fault handler. (3) Enclave page fault handler fakes access for time `t − k` and sends command to terminate. (4) Enclave fault handler terminates the enclave.

in which case the CPU is responsible for reporting page faults for these pages to the enclave directly.

Mitigating Contract Violation. When the CPU signals contract violation and the control returns to the enclave, it is important to terminate the program safely, without leaking any information (See Figure 9). When the enclave is notified about contract violation, it is the enclaves responsibility to decide whether to handle the fault or ignore it. One straightforward way to handle the fault is terminate the enclave, but our observation is that immediate program termination leaks information. In our solution, our goal is to hide the following facts (a) whether the enclave incurred a page fault during the execution after the contract is enforced (b) if so, at which point in the execution tree did the fault occur. To this end, in our defense we intercept the page faults from the underlying hardware and from that point of contract violation, we perform a fake execution to suppress the location at which the fault happened. This defense can only work if we can ensure that the enclave page fault handler is necessarily invoked. In the present SGX design it is unclear if the hardware can guarantee the invocation of the page fault handler. So we propose that SGX can adopt this solution in the future. The details of this mechanism are a bit involved and for brevity we discuss it in the extended version for interested readers [43]. We have implemented this defense in PODARCH and our evaluation on Libgcrypt [8] shows that such an approach incurs an overhead of 6.77% which is much lower as compared to the purely software based solutions (Table 3). We elide the details here due to space limits. Please refer to [43] for details.

7.2 Discussion: Other Alternative Approaches

Randomization of Page Access. Oblivious RAM (ORAM) is a generic defense that randomizes the data access patterns [25, 46]. Intuition suggests that the enclave can use ORAM techniques to conceal its memory access pattern. In this case, when an adversary observes the physical storage locations accessed, the ORAM algorithm will ensure that the adversary has negligible probability of learning anything about the true (logical) access pattern. For our AES example, we can place the tables in an ORAM to randomize their ordering, such that the adversary cannot distinguish which offsets in the tables are accessed. However, ORAM involves continuous shuffling and re-encryption of the data after every access. In our case studies, the lookup operations dominate the computation in cryptographic implementations. For millions of accesses, the cost incurred for the shuffling is significant poly log (say over $1000\times$) and slows down the applications, which is not desirable [42]. Fur-

---

[8]We did not implement contractual execution for OpenSSL because it requires dynamic loading which is not supported in PODARCH.

Table 3: Evaluation. Column 2 denotes the bucket size (Code + Data). Columns 5 and 7 denote average execution time and deviation in benign OS. Columns 8-10 denote total time spent for 3 test-case scenarios that stress the corner cases in Libgcrypt. Both the executions exhibit no statistically significant differences.

Cases	Bucket Size	PF Handler (Bytes)	Benign OS				Malicious OS		
			Vanilla Time (ms)	Contractual Time (ms)	Ovh (%)	Dev (%)	T1 (ms)	T2 (ms)	T3 (ms)
AES	3 + 3	274	4.157	4.161	**0.107**	4.689	4.287	4.179	4.059
CAST5	1 + 2	231	2.901	2.969	**2.34**	9.938	3.054	3.003	2.845
EdDSA	19 + 1	204	9729.526	9754.806	**0.260**	35.952	9960.311	9815.837	10146.534
powm	21 + 1	256	4783.997	4813.028	**0.607**	12.225	5155.958	5103.789	5224.345
SEED	2 + 2	261	1.269	1.381	**8.917**	4.821	1.337	1.392	1.333
Stribog	1 + 5	253	0.803	0.874	**8.957**	1.940	0.863	0.879	0.887
Tiger	1 + 3	244	0.506	0.644	**27.255**	4.876	0.667	0.659	0.675
Whirlpool	1 + 5	245	12.680	13.409	**5.746**	1.338	13.559	13.451	13.308
				Average	**6.77**				

ther, the best known ORAM technique requires a constant private storage for shuffling the data blocks [34]. In case of pigeonhole attack in SGX, the private storage is not permanently available to the enclave and the OS can probe operations on private memory via page faults. Thus, additional hardware support is necessary for ORAM based randomization to justify the assumption of a secure constant private storage.

Self-Paging. Instead of relying on the OS for page management, the enclaved execution can take the responsibility of managing its memory. Applications can implement self-paging to deal with their own memory faults using their own physical memory to store page tables [26]. In self-paging CPU design, all the paging operations are removed from the kernel; instead the kernel is simply responsible for dispatching fault notifications. Given a fixed amount of physical memory, the enclave can decide which virtual addresses are mapped to this memory, and which are swapped out. The problem with self-paging is — how can the enclave ensure that the OS has allocated physical pages to it? To guarantee this, the enclave should be able to pin certain physical memory pages, such that the OS cannot swap them out. This directly opens the possibility for a denial-of-service attack from the enclave, because it can refuse to give up the pinned pages. A hardware reset would be the only alternative to reclaim all the enclave pages, which is an undesirable consequence for the OS. Another possibility is that the enclave performs self-paging without assuming fixed private physical memory. But this is unsafe, since the OS still controls how much memory to allocate to the enclave, retaining the ability to pigeonhole the memory pages. In both the above alternatives, there is a dilemma — should the enclave trust the OS and likewise. Hence, it is unclear how self-paging, with or without fixed physical memory, can defend against pigeonhole attacks.

8. RELATED WORK

Attacks on Enclaved Execution. Xu et al. have recently shown that the OS can use the page fault channel on applications running on SGX based systems to extract sensitive information [51]. The attacks are limited to general user programs such as image and text processing. On the contrary we study a cryptographic implementations which is specific class of applications more relevant in the context of enclaves. More importantly, we show that the purported techniques discussed are not effective against pigeonhole attacks. As as a new contribution, we propose and measure the effectiveness of concrete solutions to prevent against such attacks on cryptographic implementations.

Side-channel Attacks. Yarom et al. study cache channel attacks wherein the adversary has the power to flush and reload the cache,

which can be used to attacks elliptic curve cryptographic routines such as ECDSA [52]. Timing and cache attacks have been used to by-pass kernel space ASLR [29], VMs [30], android applications [31], cloud servers [55] and users [40] both locally and remotely [14]. Even web browsers can be exploited remotely via cache attacks on JavaScript [39].

Side-channel Detection & Defenses. Various detection mechanisms have been explored for side channels ranging from instruction level analysis to compiler techniques [16, 23, 53]. Tools such as CacheQuant can automatically quantify the bits of information leaked via cache side-channels [33]. Techniques such as input blinding, time bucketing are also available but are limited to specific algorithms [32]. Side channel attacks in hypervisors, cloud VMs, kernel are mitigated using determinising strategies, control-flow independence and safe scheduling [9, 38, 50, 56]. Our deterministic multiplexing defense is similar to memory-trace obliviousness techniques proposed for secure computation [35].

Randomization & Self-paging Defenses. ORAM techniques are widely used in secure computation and multi-party computations. Recent work demonstrate safe language, compiler techniques, and hypervisor based approaches which use ORAM. As discussed in Section 7.2, ORAM techniques may be insufficient without extra hardware support. On the other hand, self-paging assumes that the enclave will always have control over a fixed size [26]. In case that either party breaks this assumption, it opens a potential for DOS from enclave and pigeonholing from the OS.

9. CONCLUSION

We systematically study pigeonhole attack, a new threat prevalent in secure execution platforms including Intel SGX, InkTag, OverShadow and PodArch. By analyzing cryptographic implementation libraries, we demonstrate the severity of pigeonhole attacks. We propose a purely software defense called deterministic multiplexing and build a compiler to make all our case studies safe against pigeonhole attacks. It is practically deployable with modest overhead. Finally, we present an alternative hardware-based solution which incurs an average overhead of 6.77%.

10. ACKNOWLEDGMENTS

11. REFERENCES

[1] Clang: A C language family frontend for LLVM. http://clang.llvm.org/.

[2] Libgcrypt - GNU Project - Free Software Foundation (FSF). https://www.gnu.org/software/libgcrypt/.

[3] OpenSSL: The Open Source toolkit for SSL/TLS. https://www.openssl.org/.

[4] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[5] The GNU Privacy Guard. https://www.gnupg.org/.

[6] The LLVM Compiler Infrastructure. http://llvm.org/.

[7] Software Guard Extensions Programming Reference. software.intel.com/sites/default/files/329298-001.pdf, Sept 2013.

[8] Software Guard Extensions Programming Reference Rev. 2. software.intel.com/sites/default/files/329298-002.pdf, Oct 2014.

[9] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating Timing Channels in Compute Clouds. In *CCSW*, 2010.

[10] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, 2014.

[11] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[12] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique Cryptanalysis of the Full AES. ASIACRYPT, 2011.

[13] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[14] D. Brumley and D. Boneh. Remote Timing Attacks are Practical. In *USENIX Security*, 2003.

[15] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang. *You Can't Be Me: Enabling Trusted Paths and User Sub-origins in Web Browsers*. RAID '14.

[16] R. Callan, A. Zajic, and M. Prvulovic. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In *MICRO*, 2014.

[17] S. Checkoway and H. Shacham. Iago attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ASPLOS*, 2013.

[18] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ASPLOS*, 2008.

[19] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler Mitigations for Time Attacks on Modern x86 Processors. *ACM Trans. Archit. Code Optim.*, 2012.

[20] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE S&P*, 2009.

[21] A. Dinh, P. Saxena, E. chien Chang, C. Zhang, and B. C. Ooi. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*, 2015.

[22] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting Sensitive Web Content from Client-side Vulnerabilities with CRYPTONS. CCS '13.

[23] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security*, 2013.

[24] J. Geffner. VENOM Vulnerability, May 2015.

[25] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[26] S. M. Hand. Self-paging in the Nemesis Operating System. In *OSDI*, pages 73–86, 1999.

[27] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *HASP*, 2013.

[28] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. ASPLOS, 2013.

[29] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE S&P*, 2013.

[30] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses*, LNCS, Springer. 2014.

[31] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE S&P*, May 2012.

[32] B. Köpf and M. Dürmuth. A Provably Secure and Efficient Countermeasure Against Timing Attacks. In *CSF*, 2009.

[33] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-channels. In *CAV*, 2012.

[34] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)Security of Hash-based Oblivious RAM and a New Balancing Scheme. In *SODA*, 2012.

[35] C. Liu, M. Hicks, and E. Shi. Memory Trace Oblivious Program Execution. In *CSF*, 2013.

[36] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. *SIGOPS Oper. Syst. Rev.*, 2008.

[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[38] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks. In *ICISC*, 2006.

[39] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The Spy in the Sandbox - Practical Cache Attacks in Javascript. *CoRR*, 2015.

[40] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *CCS*, 2009.

[41] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud. In *IEEE S&P*, 2015.

[42] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O ((logN) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214. Springer, 2011.

[43] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Your Faults from Telling Your Secrets: Defenses against Pigeonhole Attacks. *CoRR*, abs/1506.04832, 2015.

[44] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. PodArch: Protecting Legacy Applications with a Purely Hardware TCB. Technical report.

[45] G. Smith. On the Foundations of Quantitative Information Flow. In *FOSSACS*, 2009.

[46] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*, 2013.

[47] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *ASPLOS*, 2000.

[48] S. Tople and P. Saxena. On the Trade-Offs in Oblivious Execution Techniques. Technical report.

[49] S. Tople, S. Shinde, Z. Chen, and P. Saxena. AUTOCRYPT: Enabling Homomorphic Computation on Servers to Protect Sensitive Web Content. CCS '13.

[50] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based Defenses Against cross-VM Side-channels. In *USENIX Security*, 2014.

[51] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P*, 2015.

[52] Y. Yarom and N. Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive*, 2014.

[53] D. Zhang, A. Askarov, and A. C. Myers. Language-based Control and Mitigation of Timing Channels. In *PLDI*, 2012.

[54] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*, 2012.

[55] Y. Zhang, A. Juels, M. K. R. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*, 2014.

[56] Y. Zhang and M. K. Reiter. Duppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *CCS*, 2013.