A Compiler and Verifier for Page Access Oblivious Computation

Rohit Sinha
University of California, Berkeley
rsinha@berkeley.edu

Sriram Rajamani
Microsoft Research, India
sriram@microsoft.com

Sanjit A. Seshia
University of California, Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

Trusted hardware primitives such as Intel's SGX instructions provide applications with a protected address space, called an *enclave*, for trusted code and data. However, building enclaves that preserve confidentiality of sensitive data continues to be a challenge. The developer must not only avoid leaking secrets via the enclave's outputs but also prevent leaks via side channels induced by interactions with the untrusted platform. Recent attacks have demonstrated that simply observing the page faults incurred during an enclave's execution can reveal its secrets if the enclave makes data accesses or control flow decisions based on secret values. To address this problem, a developer needs compilers to automatically produce confidential programs, and verification tools to certify the absence of secret-dependent page access patterns (a property that we formalize as page-access obliviousness). To that end, we implement an efficient compiler for a type and memory-safe language, a compiler pass that enforces page-access obliviousness with low runtime overheads, and an automatic, modular verifier that certifies page-access obliviousness at the machine-code level, thus removing the compiler from our trusted computing base. We evaluate this toolchain on several machine learning algorithms and image processing routines that we run within SGX enclaves.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Logic and verification*; • **Software and its engineering** → Compilers;

KEYWORDS

Enclave Programs; Secure Systems; Confidentiality; Side Channels

ACM Reference format:
Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. 2017. A Compiler and Verifier for Page Access Oblivious Computation. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106248

1 INTRODUCTION

A typical computing platform contains large software layers (e.g. OS, hypervisor, firmware) in its trusted computing base (TCB), where numerous exploits have allowed privileged malware to execute [8, 9]. Recognizing this problem, processor vendors are now shipping CPUs with hardware primitives, such as Intel SGX *enclaves* [6], for isolating sensitive code and data within protected memory regions which are inaccessible to all other software running on the machine. Amongst recent applications of SGX, VC3 [16] runs Map-Reduce analytics on an untrusted cloud by computing map and reduce functions on sensitive data within enclaves, while the rest of the Hadoop stack (comprising over million lines of code) is untrusted, sees only encrypted data, and is developed using legacy software toolchains. In this new paradigm, enclaves are the only trusted (hopefully tiny) components of an application, providing us the luxury of programming them with greater rigor and stronger defenses.

Nevertheless, developing enclaves that guarantee confidentiality of sensitive data is non-trivial. Enclaves rely on the compromised host OS for I/O interactions with remote parties, scheduling, and resource management, and such interactions can reveal secrets, either directly or via side channels. In addition to protecting the enclave's outputs (e.g. by encrypting them before writing to unprotected memory), the developer faces the burden of programming the enclaves correctly against privileged attacks. Recently, Xu et al. [21] demonstrated a side-channel exploit that extracts secrets from an enclave by observing its access pattern to code and data pages — to observe page accesses, the attacker controls the valid bit in page table entries to the effect of inducing a page fault on each memory access. Practical defenses against a privileged adversary is an open research problem.

In this paper, we consider the problem of defending against an adversary that can observe page access patterns of an enclave program. We formalize a confidentiality property, termed *page access obliviousness* (PAO), that asserts that the adversary's observation of page accesses must be independent of the enclave's secrets. Our key contribution is a method for compiling high-level source programs to machine code (containing x86-64 and SGX instructions) such that the machine code satisfies PAO, and a method for efficiently verifying that the enclave binary provably satisfies PAO. This is a first step towards making the developer blissfully unaware of the sophisticated attacks that can be mounted by a privileged adversary, allowing them to focus entirely on application logic.

To that end, we implement PAO enforcement within a compiler for EncLang, a general-purpose language for programming enclaves. The rationale for developing EncLang is two-fold: 1) memory accesses in mainstream languages (such as C, Rust) are determined by the compiler implementation, which offers us no control over the placement of objects and code in memory, thus hindering a static scheme for enforcing PAO — a dynamic scheme for LLVM bytecode [17] has been proposed, but with prohibitively high performance overheads; 2) because enclaves do not trust non-enclave software, they cannot use legacy software toolchains and thus present a rare opportunity for clean slate programming and verification. Our compiler accepts arbitrary programs in EncLang and *obliviates* the page accesses in the compiled x86-64 program i.e. satisfies PAO by making page access pattern independent of

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106248

secrets. For instance, the compiler ensures that secret-dependent branches fetch instructions from the same (sequence of) pages in both branches. The compiler also performs stack allocation and lays out data structures in the enclave's heap so that a memory access via a secret address (e.g. array access with a secret index) generates a deterministic page access sequence in all executions. During this process, the compiler instruments dummy memory accesses or inserts padding space between objects in memory to *obliviate* the page accesses. For performance, the compiler performs stochastic optimization (based on Markov Chain Monte Carlo sampling) to reduce the increase in code and data size of compiled programs. Although we implement these techniques in a compiler for EncLang, the ideas are generally applicable to mainstream languages. Moreover, we use EncLang to only program the enclave components of an application, which we empirically observe to be a small fragment of the entire application (e.g. map and reduce functions in VC3).

Next, we develop a separate verifier that proves that the output machine code satisfies PAO. The gains are two-fold: 1) the compiler is no longer trusted, thereby freeing us to implement aggressive optimizations without inadvertently sacrificing PAO, and 2) the verifier is significantly simpler, and shrinks the size of our TCB, which now includes only the CPU hardware and our verifier. Furthermore, to alleviate the complexity of verifying arbitrary machine code, we engineer the compiler to supply (untrusted, but useful) hints to the verifier — we follow the typed assembly language paradigm of having the compiler output typing annotations along with the machine code, and create a set of simple typing judgments for efficient verification.

This paper makes the following novel contributions:

(1) We formalize PAO for enclave programs which execute in the presence of a privileged adversary.
(2) We present a toolchain, consisting of a type system and compiler, that automatically enforces PAO while compiling to machine code, and implements a stochastic optimization to reduce runtime and memory overheads.
(3) To minimize our TCB, we develop a verifier that analyzes the compiled machine code to prove PAO. To further simplify the verifier, we design a typed assembly language, where the typing annotations accompany the machine instructions and reduce the verification task to efficient type checking.
(4) We evaluate the toolchain on several machine learning algorithms and image processing routines, for which attacks have been demonstrated in [21].

2 OVERVIEW

Enclaves. An enclave is a protected region in memory (containing code and data) that can only be accessed by the code running within the enclave. The trusted CPU implements instructions for the untrusted software to launch enclaves; after launch, any non-enclave software is prohibited from accessing the enclave's memory — the CPU implements measurement and remote attestation primitives to ensure launch-time integrity [6]. An enclave (e.g. Hadoop reducer in VC3 [16]) occupies a virtual address space contained within the hosting application's (e.g. Hadoop process) address space; the enclave's code runs at the lowest (ring 3) privilege level. An enclave is not allowed to make system calls since the OS cannot be trusted to modify the enclave's memory safely. However, the enclave can access the hosting application's memory (but not the other way around), which allows for efficient I/O between the enclave and the external world — all system calls are proxied by the hosting application. The host application can invoke code inside the enclave via a statically defined entry-point, and the enclave code can transfer control back via an exit instruction. Control may also transfer out of the enclave asynchronously due to interrupts, faults, and exceptions, in which case the CPU protects sensitive state e.g. by saving context in the enclave's memory and zeroing out the registers.

A typical enclave follows a stylized idiom where it copies encrypted inputs from the host application's memory, decrypts the input using a key known only to the enclave, computes and encrypts the output, and copies the encrypted output to the host application's memory. To support this programming idiom, we emulate the design of [18] and supply a runtime library with send and recv APIs (whose specification is formalized in [18]), which provide the only means for communicating with the external world; send encrypts the message and writes out the ciphertext, whereas recv decrypts the incoming ciphertext, and provides the plaintext message to the caller.

Threat Model. We assume a software adversary that has full control of all system software: OS, hypervisor, system management mode firmware, and BIOS. As a result of these privileges, the adversary has full control over non-enclave memory, I/O peripherals, disks, and network; it may record, replay, or modify network messages and disk contents. The adversary may force the CPU to transfer control from the enclave to the untrusted OS at any time during execution (by generating an interrupt, for example). Once the CPU transfers control to the adversary, the adversary may execute an arbitrary adversarial operations before transferring control back to the enclave.

To allow an OS autonomy over memory paging decisions, Intel SGX places the page tables under the OS control. For security, the CPU implements an inverse page table mapping to ensure that the OS cannot change the physical mapping for any address in enclave's region. However, at any point, the attacker may modify the page table entries to the effect of inducing a page fault on each enclave memory access (e.g. by clearing the valid bit). That being said, the OS' page fault handler only needs to know the accessed page (and not all bits of the address), hence the CPU clears the page offset bits from the faulting address (12 least significant bits for 4KB-sized pages) prior to delivering the page fault exception. This reveals the enclave's memory access patterns only at the page-level granularity. Recent attacks [21] on enclaves have extracted secrets via this channel, and preventing such leaks is a key contribution we make.

Defenses against hardware attacks are out of scope for this paper. For instance, we assume that the adversary cannot physically attack the CPU package to extract secrets nor snoop on the hardware bus connecting the CPU and DRAM. The latter assumption prevents the adversary from learning access patterns at the byte-level granularity, which would necessitate a more sophisticated defense. We also don't defend against timing leaks, which may result from 1) timing of page accesses, 2) cache timing attacks which the attacker uses to infer access patterns at a cache-line granularity.

Challenges in Guaranteeing Page Access Obliviousness. We choose a simple notion of confidentiality: the adversary's observations of page accesses during enclave execution must be independent of the enclave's secrets.

Consider a sample enclave in Figure 1, which evaluates a decision tree to classify an input instance. The decision tree and the evaluate algorithm are known to the adversary, whereas the input instance and output decision must be kept confidential — to that end, we invoke the runtime's send API to encrypt the output before writing to non-enclave memory. To classify the instance, the procedure traverses the tree (stored as a flattened array) starting from the root node, until it reaches a leaf node (which is any node with a non-zero value in the decision field); the evaluation uses an index variable to record the current node in the traversal. At each interior node, the procedure compares the value of a decision variable with a threshold value, and recurses on either the left or right subtree based on the outcome. The path taken through the tree reveals predicates that hold on the secret instance, which the attacker infers by monitoring the enclave's accesses to code and data pages. Therefore, amongst other measures, the enclave developer must ensure that the page accesses are independent of the path, a property entailed by PAO. Guaranteeing PAO for the evaluate procedure has the following challenges:

- In the case of an unbalanced decision tree, evaluate terminates after varying number of iterations (based on secret), and the attacker may infer the path length by counting the number of page accesses. In other words, each invocation of evaluate leaks at most $\log_2 k$ bits of secret, where k is the height of our decision tree.
- We have a secret-dependent conditional statement (line 23). Monitoring the enclave's accesses to the code pages allows the attacker to infer which branch is taken, if any of the instructions implementing the if branch (line 25) is placed in a different page than the instructions implementing the else branch (line 27). Mainstream compilers often optimize for code size and performance, but make no effort to control the layout of instructions.
- We find several data accesses where the address depends on a secret value. For instance, in lines 22 - 27, the array access tree[index] computes a reference to a node within the tree, where index is a secret; this makes the address evaluate to a secret value. In the case that tree is stored across multiple pages — because the tree size is larger than a single page, or due to layout decisions made by the compiler — the attacker infers some bits of the secret index by monitoring the enclave's accesses to data pages.

Compilation for Page Access Obliviousness. We develop a compiler for producing PAO-satisfying x86-64 code from arbitrary EncLang programs. First, a type system (described in § 4.5) flags violations where the enclave leaks secrets in ways that an automatic compiler cannot fix (without developer intervention): loops with secret-dependent condition i.e. secret number of iterations, explicit leaks via assignment of secret values to public state, and implicit leaks via assignment to public state within a secret conditional branch — these typing restrictions are common for type systems for non-interference and side-channel mitigations. In the evaluate procedure in Figure 1, the developer replaces the secret-dependent loop condition (line 19) with a loop that executes for fixed number of iterations (line 20), thus trivially satisfying the typing rule that

```
1  global tree:
2    array[2^k-1]
3      struct {
4        left : idx<2^k><public>, /* index of left subtree*/
5        right : idx<2^k><public>, /* index of right subtree*/
6        decision : uint64<public>, /* != 0 for leaf node */
7        dvar : idx<d><public>,   /* decision variable */
8        threshold : uint64<public> /* threshold value */
9      };
10
11 void evaluate(instance : ref array[d]<secret> uint64 )
12 {
13   local decision: uint64<secret>; /* evaluation result */
14   local index: idx<2^k><secret>; /* values 0 to 2^k-1 */
15
16   index := 0;            /* start traversal at root */
17   decision := 0;         /* terminates when non 0 */
18
19   while (decision = 0) {
20   /* for (0..k) { */
21     if (decision = 0) {
22       decision := tree[index]->decision;
23       if (instance[tree[index]->dvar] <=
24           tree[index]->threshold) {
25         index := tree[index]->left; /* recurse left */
26       } else {
27         index := tree[index]->right; /* recurse right */
28       }
29     }
30   }
31   send(decision);
32 }
```

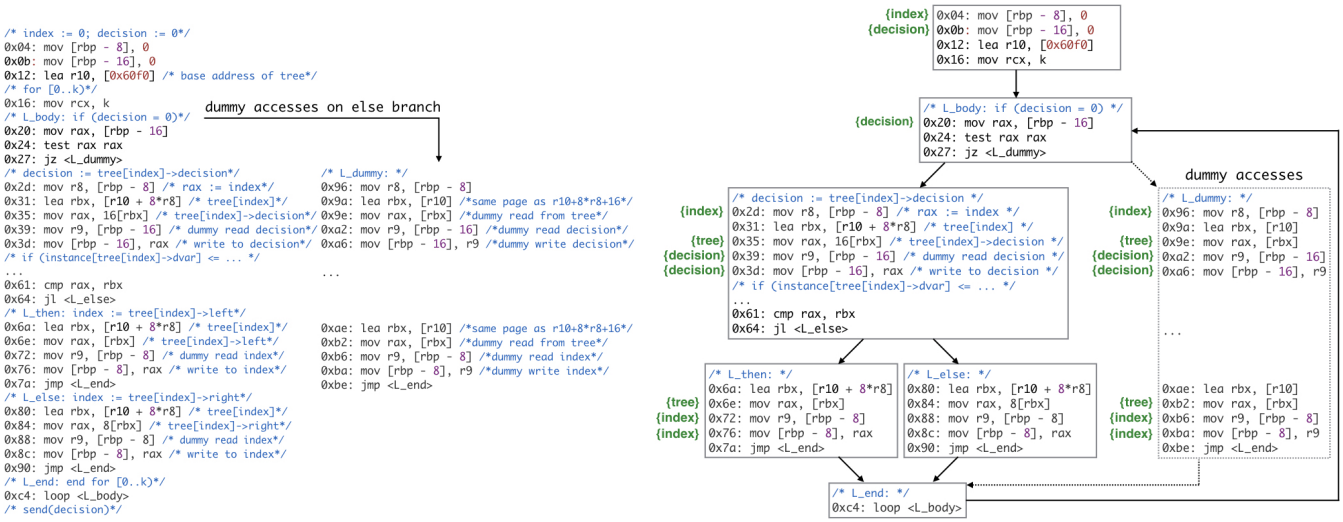
Figure 1: Decision tree evaluation

loop exit conditions must only depend on public values. The type checker finds no other violations.

The compiler (described in § 4) then compiles to x86-64 code, listed in Figure 2(a), while also enforcing PAO by controlling the layout of data structures and instructions in memory — where necessary, it generates dummy page accesses to *obliviate* the access patterns, as discussed below. It takes the following necessary measures for our sample enclave.

Even with the fix on line 20, the enclave remains vulnerable — the program effectively stops computing and does not perform data accesses once the traversal reaches a leaf node (i.e. condition on line 21 evaluates to `false`), thus allowing the adversary to infer the path length by counting the accesses to the data pages. To correctly conceal this leakage, the compiler places dummy accesses in the `else` branch (corresponding to the `if` on line 21) to account for the imbalance in the tree. As seen in Figure 2(a), the dummy accesses are performed using instructions within the address range 0x96 to 0xbe, and they target the same sequence of pages as the two program paths in the input program — this is achieved via dummy reads from the same object, and by controlling the placement of objects.

Second, to prevent secrets from leaking via data accesses (e.g. tree[index], where the address depends on a secret), the compiler must either 1) layout the tree to fit entirely within a page (if possible), causing all accesses to the tree to target the same page, or 2) allow the tree to span multiple pages, and introduce dummy accesses to all pages except the page containing tree[index]. For a simpler presentation, the compilation in Figure 2 assumes that the tree object fits within a single page — § 4.1 presents a general scheme. Despite this simplifying assumption, we must generate dummy accesses to tree in the `else` branch (corresponding to the



(a) Compiler safely lays out code and data pages, and instruments dummy accesses

(b) Verifier constructs CFG and computes aliases for all memory accesses

Figure 2: Compiling and verifying evaluate **to guarantee page access obliviousness**

`if` on line 21). A dummy read (e.g. instruction 0x9e, which mimics 0x35) is performed by fabricating an address within the tree. A dummy write (e.g. instruction at 0xa6, which mimics 0x3d) is performed by first issuing a dummy read (instruction 0xa2, which we compensate by adding instruction 0x39 in the original path), and then writing the read value back at the same address, thus preventing dummy accesses from modifying state.

Finally, To hide control decisions based on secret input (line 23), the compiler lays out the instructions from both branches onto the same page, when possible. Inevitably, to handle cases where the cumulative code within the branches of a secret conditional cannot be fit onto a single page, the compiler partitions code across pages such that the sequence of code page accesses is equivalent in the two branches — the compiler splits each branch into snippets, and maps snippets to pages such that the *n*th chunk of both branches have equal number of instructions and occupy the same page. Finally, the compiler instruments nop instructions to equalize the number of instructions (code accesses) in the two branches. Further details are presented in § 4.3. For simplicity, we elide the nop instructions in Figure 2, and also manage to place all of the compiled machine code for evaluate within one page.

Verifying Page Access Obliviousness. An enclave violates PAO if executions with different secret values produce different sequences of page accesses. We verify that the output machine code satisfies PAO, thus removing the compiler's implementation from the trusted computing base.

In this paper, we show that verifying PAO requires sound (but necessarily incomplete) algorithms for alias analysis and control flow analysis — in practice, we are able to implement a simple, yet precise algorithm for these analyses because our compiler produces idiomatic code and supplies hints to the verifier, following the paradigm of typed assembly language. First, our verifier takes the enclave program as input and computes its control flow graph (CFG), as shown in Figure 2(b). Next, our verifier performs an alias analysis, annotating each memory access with a set of objects that the access may target. We show the aliases within curly braces in Figure 2(b); the verifier also annotates the aliases for the dummy accesses along the else branch (shown within the dotted box in the CFG), corresponding to the if in line 21. The verifier uses these analyses to prove PAO: for any pair of executions of the enclave binary (that only differ in secret values), the sequence of page accesses must be equivalent, where two accesses are equivalent if they target the same page and have same type (read / write / execute). The machine code in Figure 2 satisfies PAO trivially because the alias analysis computes only one object for each memory access, and all paths have equivalent sequence of accesses to code and data pages.

3 PAGE ACCESS OBLIVIOUSNESS

3.1 Formal Modeling of Enclave Code

Instructions. An enclave program is a partial map from 64-bit addresses (in enclave memory) to user-mode instructions, with a unique entrypoint. In addition to standard x86-64 instructions, the CPU enables SGX instructions in enclave mode to perform cryptography (e.g. ereport, egetkey), and exit to non-enclave code (eexit). On the other hand, certain instructions such as system calls are disabled because the enclave cannot trust the OS to update enclave's memory. For simplicity (of our implementation rather than the methodology), we assume that the compiler produces enclave programs that only contain a subset of x86 instructions — it includes mov for loads and stores, conditional jumps such as jl, call and ret for procedure calls, and several arithmetic and relational operators. Our methodology assumes single-threaded enclaves.

State. The enclave program's state consists of variables (denoted by *Vars*): `regs`, `flags`, and `mem`. `regs` are CPU registers (e.g., rax, r8, rsp, etc.), each being 64 bits wide. CPU flags (e.g., CF, ZF, etc.) are 1-bit values. The instruction pointer (rip $\in$ `regs`) stores the address of the next instruction to be executed and is incremented automatically after every instruction, except in those that change the control flow: jumps, call, and ret. Memory (mem) is modeled as

a map from 64-bit addresses to 8-bit data. Although the machine's state may consist of other elements (such as control register CR4, etc.), we omit them from the enclave's state even though they impact its execution. There are two main reasons for this modeling choice: 1) we assume the ISA-defined operational semantics of each instruction, which the CPU must fulfill in enclave mode, regardless of how the adversary modifies the unprotected state elements, and 2) we would like to abstract away from the specifics of a particular ISA implementation, and we find that our abstraction of machine's state (i.e. `regs`, `flags`, and `mem`) applies to other trusted hardware platforms.

Semantics. We define a concrete state σ to be a valuation of all variables in *Vars*. Let $\sigma(\mathsf{v})$ be the value of a variable $\mathsf{v} \in \mathit{Vars}$ in state σ. Let $\mathtt{instr}(\sigma)$ be the instruction executed in state σ (computed from the instruction pointer rip and the contents of `mem`). The semantics of an instruction $i \in \mathit{Instr}$ is given by the relation $\Downarrow$, where $\langle i, \sigma\rangle \Downarrow \sigma'$ if and only if $i = \mathtt{instr}(\sigma)$ and there is an execution of i starting at σ and ending in σ' (as per the operational semantics). Due to space constraints, we define the operational semantics for a subset of *Instr* in the technical report [20]. A sequence $\pi = [\sigma_0, \ldots, \sigma_n]$ is called an *execution trace* if $\langle \mathtt{instr}(\sigma_k), \sigma_k\rangle \Downarrow \sigma_{k+1}$ for each $k \in \{0, \ldots, n-1\}$.

3.2 Modeling Adversary's Effect on Enclave Execution

Our formal model of an active adversary's operations is similar to Moat [19] — we only extend Moat's adversary with observations of page-level accesses. The adversary may force the CPU to transfer control from the enclave to the untrusted OS at any time during execution (by generating an interrupt, for example). From then on, the adversary executes an arbitrary sequence of instructions before transferring control back to the enclave. The adversarial operations include modifications to non-enclave memory, privileged state accessible to the OS and hypervisor layers (e.g. page tables), and devices. Moat proves a theorem that an unbounded sequence of these privileged operations can be simulated by an adversary that is only allowed to modify non-enclave memory — in other words, the CPU ensures that the adversary can only impact the enclave's execution when the enclave loads inputs from non-enclave memory. For sound analysis, our model of the enclave program havocs the output of recv, which is the only mechanism for fetching inputs from non-enclave memory.

3.3 Page Access Obliviousness

We first define confidentiality for enclave programs, and then instantiate this definition to attain PAO. An execution trace π starts in the initial state of the machine following a power cycle; at some point in the trace, the adversary launches the enclave program. From then on, π consists of alternating sequences of adversarial and enclave instructions. We use $seq_A(\pi, i)$ and $seq_E(\pi, i)$ to denote the i-th subsequence of adversarial and enclave instructions, respectively; figure 3 illustrates these functions. Let the projection function A denote the component of enclave-observable machine state that the adversary is allowed to control; we define $A(\sigma) = \sigma(\mathtt{mem}_{\neg\mathsf{ENC}})$, where $\mathtt{mem}_{\neg\mathsf{ENC}}$ denotes non-enclave memory. Note that the adversary may invoke privileged instructions that modify state beyond $\mathtt{mem}_{\neg\mathsf{ENC}}$ (e.g. control register CR4). However, we omit these state variables from A because they are not included in the enclave's state or the operational semantics, and the trusted CPU must guarantee the operational semantics during enclave execution, regardless of how the adversary manipulates this additional machine state.

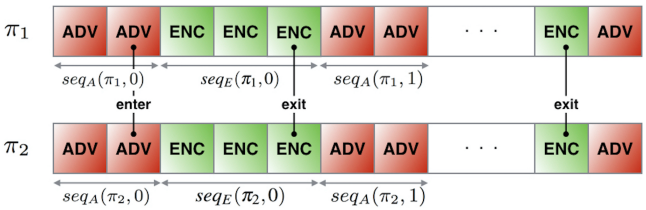


Figure 3: Illustration of confidentiality definition.

Definition 3.1. **Confidentiality** For any pair of execution traces of the machine, if the adversary's operations along the two traces are equivalent, then the adversary's observations along the two traces must also be equivalent.

$$\forall \pi_1, \pi_2 \in \Sigma^*. \pi_1 \equiv_A \pi_2 \Rightarrow \pi_1 \equiv_O \pi_2$$

where

$$\begin{aligned}&\pi_1 \equiv_A \pi_2 \Leftrightarrow\\ &\forall i.\ \mathtt{instr}(seq_A(\pi_1, i)) \equiv \mathtt{instr}(seq_A(\pi_2, i)) \wedge\\ &A(seq_E(\pi_1, i)[0]) \equiv A(seq_E(\pi_2, i)[0])\end{aligned}$$

Confidentiality, a hyper-property defined over pairs of executions, is violated when the enclave produces observationally different traces for equivalent adversarial operations. Equivalence is defined using relations $\equiv_A$ and $\equiv_O$. The equivalence relation $\equiv_A$ over pairs of adversarial subsequences ($seq_A(\pi_1, i)$ and $seq_A(\pi_2, i)$) only includes traces that 1) have equal lengths, 2) have the same instructions, and 3) produce the same sequence of states (where equivalence is defined modulo the projection function A). Equality of states modulo A is naturally defined to be bitwise equivalence for all locations in $\mathtt{mem}_{\neg\mathsf{ENC}}$, with the caveat that encrypted values can differ. Specifically, using the approach of cryptographically-masked flows [2], we treat all valid ciphertexts to be equivalent — this ensures that calling recv will succeed and produce different secret inputs in both π_1 and π_2. Without this restriction, definition 3.1 would force all encrypted inputs to have equivalent bitwise values, thereby forcing secrets to have the same values in π_1 and π_2, which may hide information leaks. For checking that the enclave's observed behaviors are equivalent, we use an equivalence relation $\equiv_O$, defined below.

Definition 3.2. **Page Access Obliviousness** An enclave is page access oblivious if it satisfies confidentiality, with observation function O set to the following PA function.

$$\begin{aligned}&\pi_1 \equiv_O \pi_2 \Leftrightarrow\\ &PA(seq_E(\pi_1, i)) \equiv PA(seq_E(\pi_2, i))\end{aligned}$$

$$PA(\sigma_0, \ldots, \sigma_n) \doteq [PA_c(\sigma_0) \cdot PA_d(\sigma_0) \cdot \ \ldots\ \cdot PA_c(\sigma_n) \cdot PA_d(\sigma_n)]$$

$$PA_c(\sigma) \doteq \langle \text{execute}, \sigma(\text{rip}/2^{\text{p}})\rangle$$

$$PA_d(\sigma) \doteq \begin{cases} \langle \text{read}, \sigma(\text{reg}_{\text{a}}/2^{\text{p}})\rangle & \mathtt{instr}(\sigma) = \text{mov reg}_{\text{d}}\ [\text{reg}_{\text{a}}] \\ \langle \text{write}, \sigma(\text{reg}_{\text{a}}/2^{\text{p}})\rangle & \mathtt{instr}(\sigma) = \text{mov}\ [\text{reg}_{\text{a}}]\ \text{reg}_{\text{d}} \\ \langle \text{read}, \sigma(\text{rsp}/2^{\text{p}})\rangle & \mathtt{instr}(\sigma) \in \{\text{pop reg, ret}\} \\ \langle \text{write}, \sigma(\text{rsp}/2^{\text{p}})\rangle & \mathtt{instr}(\sigma) \in \{\text{push reg, call}\} \\ \epsilon & \text{otherwise} \end{cases}$$

The *PA* function permits the adversary to observe all memory accesses at the page-level granularity (enforced by the division by page size 2^p). The value of p is architecture-specific; a page has size 4096 bytes in Intel SGX CPUs, which makes p = 12. As dictated by *PA*, for each instruction executed in enclave-mode, the attacker records 1) access to a code page at address rip to fetch the instruction, and 2) access to a data page, if the instruction triggers a data access (e.g. push performs a write access to a data page at address rsp). This definition of *PA* represents the x86-64 ISA semantics, even in the presence of optimizations such as prefetching and caching, because the CPU evaluates the page permission check for each memory access. The adversary also observes the type of memory access: read, write, or execute.

4 PAO-ENFORCING COMPILATION

This section presents an algorithm for obliviating data accesses at the machine code level (§ 4.1), a stochastic optimization step for lowering runtime overheads of this defense (§ 4.2), and an algorithm for obliviating code accesses (§ 4.3). We also present EncLang (§ 4.4) and its compiler (§ 4.5, § 4.6), which implements these algorithms to produce PAO-satisfying machine code.

4.1 Obliviating Data Accesses

Consider the following secret-dependent conditional branch:

```
if (s) {          /* s: bool<secret> */
  b[i] := a[k];   /* a: array[5000] uint8<public> */
} else {          /* b: array[10] uint8<secret> */
  c := 0;         /* c: uint64<secret> */
}
```

With different values of the secret s in executions π_1 and π_2 (from definition 3.1), the attacker observes different data accesses in π_1 and π_2:

- π_1 (if branch): $R[\![\text{s}]\!]$, $R[\![\text{a[k]}]\!]$, $W[\![\text{b[i]}]\!]$
- π_2 (else branch): $R[\![\text{s}]\!]$, $W[\![\text{c}]\!]$

Here, $R[\![]\!]$ and $W[\![]\!]$ indicate read and write operation, respectively — we defer the treatment of code accesses (for fetching instructions) to section 4.3. A strawman PAO-enforcement scheme *obliviates* page accesses in the two branches by introducing dummy read ($\hat{R}[\![e]\!]$) and dummy write ($\hat{W}[\![e]\!]$) operations, which are guaranteed to 1) target the same page as the real read ($R[\![e]\!]$) and write ($W[\![e]\!]$) operations that they mimic, and 2) not cause any side-effect on the program's state. This approach results in the following data accesses along the two branches of the above program:

- π_1 (if branch): $R[\![\text{s}]\!]$, $R[\![\text{a[k]}]\!]$, $W[\![\text{b[i]}]\!]$, $\hat{W}[\![c]\!]$
- π_2 (else branch): $R[\![\text{s}]\!]$, $\hat{R}[\![\text{a[k]}]\!]$, $\hat{W}[\![\text{b[i]}]\!]$, $W[\![\text{c}]\!]$

The implementation of dummy operations $\hat{R}[\![]\!]$ and $\hat{W}[\![]\!]$, which execute in a different code path than the real operations they mimic, must compute the same address or at least an address to the same page. This is trivial for scalar objects (e.g. variable c, which has a fixed location on the program stack). In the case where the object spans multiple pages (e.g. larger-than-page array a in $R[\![\text{a[k]}]\!]$), we cannot statically identify a unique page targeted by a read / write operation — in general, the address may be computed using a secret (e.g. secret k in a[k]), and secrets may evolve to different values within the two branches of a secret-dependent conditional. Therefore, the dummy operation $\hat{R}[\![\text{a[k]}]\!]$ must access each page that contains some part of the object, which also forces us to access the same pages to implement the real $R[\![\text{a[k]}]\!]$ lest we violate PAO. In other words, the $R[\![]\!]$, $\hat{R}[\![]\!]$, $W[\![]\!]$, and $\hat{W}[\![]\!]$ operators — defined in Figure 4 and described below — may perform multiple memory accesses for each read / write operation.

First, given a $R[\![e]\!]$ or $W[\![e]\!]$ operation to mimic, we perform a best-effort analysis to identify the object being targeted by the reference *e*, hereby called the *statically-identifiable* object. The statically-identifiable object, or $si(e)$, denotes a (contiguous) region of memory that is guaranteed to contain the address *e*, in all executions of the program. EncLang provides two constructs for computing references: the $\rightsquigarrow$ operator converts a reference to a struct into a reference to the named field, and the [.] operator returns a reference to the indexed array element — both operators have standard C-like semantics. We compute $si(e)$ syntactically by traversing struct accesses in *e* until we arrive at a scalar or an array access — we stop the recursion at an array access to avoid performing range analysis, thereby allowing the computation of si to be syntactic. For instance, $si(\text{a[k]})$ is a; $si(\text{w} \rightsquigarrow \text{x[y} \rightsquigarrow \text{z]})$ is $\text{w} \rightsquigarrow \text{x}$; $si(\text{w} \rightsquigarrow \text{x})$ is $\text{w} \rightsquigarrow \text{x}$; $si(\text{s})$ is s. In general,

$$si(e) \doteq \begin{cases} id & \text{where } e \leftarrow \text{id} \\ si(e_a) & \text{where } e \leftarrow e_a[e_i] \\ si(e_s) & \text{where } e \leftarrow e_s \rightsquigarrow \text{id and } e_s \text{ contains array access} \\ e & \text{where } e \leftarrow e_s \rightsquigarrow \text{id and } e_s \text{ has no array access} \end{cases}$$

Figure 4 defines the machine code implementation of the $R[\![e]\!]$, $W[\![e]\!]$, $\hat{R}[\![e]\!]$, and $\hat{W}[\![e]\!]$ operators. If $si(e)$ resolves to a scalar value (i.e. ref cell in EncLang), then both $R[\![e]\!]$ and $\hat{R}[\![e]\!]$ execute a single mov instruction to read the value. Else, we perform a linear scan over all pages that contain the array referenced by $si(e)$ (as illustrated in Figure 4). To do so, it suffices to know the size of the array referenced by $si(e)$, which we compute using the type inference in EncLang (§ 4.5) — the base address can be evaluated at runtime. Assuming the worst case layout of $si(e)$, where the object may span upto $\lceil \text{size(si(e))} / 2^p \rceil + 1$ pages, we perform the linear scan in $\hat{R}[\![e]\!]$ by issuing dummy loads from all these pages, and perform the linear scan in $R[\![e]\!]$ by issuing a real load to the page containing address *e* and dummy loads to the remaining $\lceil \text{size(si(e))} / 2^p \rceil$ pages. To an attacker, a real load is indistinguishable from a dummy load — PAO is preserved by having equivalent number and type of page accesses in $R[\![e]\!]$ and $\hat{R}[\![e]\!]$. Observe that a dummy load targets the lowest address of a page (i.e. bottom p bits are 0), and discards the result of the load, whereas a real load targets the intended address *e* and saves the result in rax. The implementation is similar for $W[\![e]\!]$ and $\hat{W}[\![e]\!]$. The linear scan uses dummy stores that target the lowest address of a page, and first loads a value from that address, and then stores that same value back. Meanwhile, a real store first loads a value from the target address into a dead register, and then stores the new value — to an attacker, a real store is indistinguishable from a dummy store. The implementation makes use of an oblivious move primitive called omove, which we borrow from [14]. Effectively, dst := omove(c, x, y) performs a conditional move without introducing a conditional branch — it moves both x and y into temporary registers, evaluates c, and uses the cmovz instruction to move either x (if c is true) or y (if c is false) into dst.

Note that the compiler only introduces dummy operations $\hat{R}[\![e]\!]$ and $\hat{W}[\![e]\!]$ within secret-dependent conditional branches. Code within public conditionals do not require any PAO-related defenses, as the program executes the same branch in both π_1 and π_2, and hence the attacker observes equivalent page accesses. Although, for simplicity, our example only contains one conditional statement,

Reference Type	$R[\![e]\!]$: expects input e in rax, and produces output in rax	$\hat{R}[\![e]\!]$: gets input e in rax
$\Gamma \vdash_e si(e)$: ref cell ...	mov rax [rax]	mov rax [rax]

$\Gamma \vdash_e si(e)$: ref array ...
 	
let s = ⌈ size(si(e)) / 2^p ⌉ + 1
denote the maximum number
of pages that si(e) can occupy
	for k in [0..s − 1] {
	r11 := pg(si(e)) + 2^p * k ≥ rax < pg(si(e)) + 2^p * (k + 1)
	rcx := omove(r11, rax, pg(si(e) + 2^p * k))
	mov rcx [rcx]
	rdx := omove(r11, rcx, rdx)
	}
	rax := rdx
		for k in [0..s − 1] {
		rdx := pg(si(e)) + 2^p * k
		mov rdx [rdx]
		}

Reference Type	$W[\![e]\!]$: expects input e in rax, and value in rbx	$\hat{W}[\![e]\!]$: gets input e in rax
$\Gamma \vdash_e si(e)$: ref cell ...	mov rdx [rax] mov [rax] rbx	mov rdx [rax] mov [rax] rdx

$\Gamma \vdash_e si(e)$: ref array ...
 	
let s = ⌈ size(si(e)) / 2^p ⌉ + 1
denote the maximum number
of pages that si(e) can occupy
	for k in [0..s − 1] {
	r11 := pg(si(e) + 2^p * k) ≥ rax < pg(si(e)) + 2^p * (k + 1)
	rcx := omove(r11, rax, pg(si(e)) + 2^p * k)
	mov r12 [rcx]
	rdx := omove(r11, rbx, r12)
	mov [rcx] rdx
	}
		for k in [0..s − 1] {
		rcx := pg(si(e)) + 2^p * k
		mov rdx [rcx]
		mov [rcx] rdx
		}

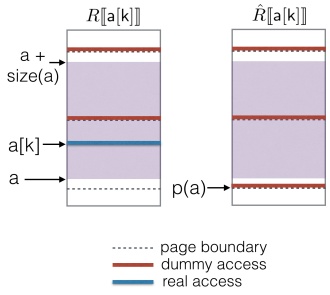


Figure 4: Implementation of oblivious read / write primitives. The omove **primitive performs a conditional move using the** cmovz **instruction.** pg(x) **denotes the starting address of the page containing address** x **and is defined to be** x & !(2^p − 1)**.**

the algorithm handles arbitrary nesting of conditional statements. This section proposed a naive, but sound defense for enforcing PAO; the following section optimizes the program to remove unnecessary dummy accesses, without compromising soundness.

4.2 Stochastic Optimization of Dummy Accesses

So far, our strategy for obliviating data accesses assumed the worst-case layout of objects in memory. First, the $R[\![]\!]$ and $W[\![]\!]$ operators produce ⌈ size(si(e)) / 2^p ⌉ + 1 page accesses, accounting for the pathological case where even a two-byte object can span two pages — we observe this phenomenon in $W[\![\text{b[i]}]\!]$, which needs only one page access if the 10-byte array b can be placed entirely within a page. Second, the strategy of introducing one dummy operator for each real operator in the other path assumes the worst case setting where all objects are placed in unique pages, which is rarely the case — we observe this phenomenon in the toy example from section 4.1, where both $\hat{W}[\![\text{b[i]}]\!]$ and $\hat{W}[\![\text{c}]\!]$ can be elided objects if b and c are placed in the same page. With this insight, we have an optimized sequence of data accesses:

- π_1 (if branch): $R[\![\text{s}]\!]$, $R[\![\text{a[k]}]\!]$, $W[\![\text{b[i]}]\!]$
- π_2 (else branch): $R[\![\text{s}]\!]$, $\hat{R}[\![\text{a[k]}]\!]$, $W[\![\text{c}]\!]$

We formulate this program transformation as a stochastic optimization problem, and solve it using Markov Chain Monte Carlo (MCMC) sampling, which quickly explores a large search space of program rewrites. The cost function to optimize is the number of dummy page accesses. The MCMC sampling chooses amongst two candidate moves:

(1) *Map objects to pages*: Randomly select a stack-allocated object and place it onto a random page (on the current stack frame) that has enough contiguous, free memory.
(2) *Dummy accesses*: Randomly select a pair of $\hat{R}[\![]\!]$ (or $\hat{W}[\![]\!]$) operations and remove it from the program. Or, for ergodicity [15], create a pair of $\hat{R}[\![]\!]$ (or $\hat{W}[\![]\!]$) operations, each with a randomly chosen object as the argument.

These optimizations require the compiler to control the placement of local, stack-allocated objects. We assume no control over heap layout as it is performed by malloc, and the layout of globals is performed independently of all procedures — in principal, these can be optimized as well, but that would add unmanageable complexity to the compiler.

4.3 Obliviating Code Accesses

The branches of a secret-dependent conditional may contain unequal number of instructions, and hence produce unequal number of code page accesses in π_1 and π_2 (from definition 3.1) — in fact, the reader may notice in Figure 4 that although our implementation of $R[\![e]\!]$ produces equal number of data accesses as $\hat{R}[\![e]\!]$, $R[\![e]\!]$ uses far more instructions. To obliviate the code accesses, the compiler instruments nop instructions on the same page as the instructions being mimicked — we apply this defense after the transformations in sections 4.1 and 4.2, ensuring that enough code accesses are interleaved with the data accesses to satisfy PAO. Note that this instrumentation is only applied to secret-dependent conditionals.

The nop instrumentation ensures that all paths within a secret-dependent conditional have equal number of x86-64 instructions. However, this is not enough for PAO. Without consideration to code layout, instructions in different branches may get placed onto separate pages. This may happen if there is not enough space left on the current code page, or if the cumulative size of the branches is larger than a page. In such cases, we split each branch into a sequence of snippets, map snippets onto pages such that the *n*th snippet of both branches has equal number of instructions and occupies the same page, and stitch the snippets together using unconditional jumps.

4.4 The EncLang Language

Our goal is to empower the developer with a PAO-enforcing compiler and verifier toolchain, which implements the defenses in sections 4.1-4.3. Modifying off-the-shelf compilers for mainstream languages would require significant overhaul. We must at the very least: (1) add security types to identify secret branch conditions — in the case of C/C++, the type system is also unable to strictly enforce type and memory safety, making it exceedingly difficult to statically reason about aliasing, control flow, and object sizes

(e.g. evaluating *si* in 4.1); (2) extend the operational semantics to define memory accesses for each expression and statement in the language and enforce these semantics in all phases of the compilation — mainstream languages (e.g. C, Rust) leave these details to a compiler; (3) extend the compiler implementation with techniques proposed in this paper: issue dummy accesses, place nop instructions, arrange objects in data pages, arrange instructions in code pages, etc. — mainstream compilers don't offer this level of control over the code generation. Therefore, we implement our PAO enforcement within a compiler for a new language, called EncLang, that is mostly inspired from Ivory [4], Cyclone [7] and the work by Jones [3]. Note that EncLang is used to program the (security-critical, and hopefully small) enclave, whereas the rest of the application is developed using commodity toolchains. Figure 5 presents the syntax, and the rest of this section discusses the key features of EncLang.

Establishing memory safety is paramount for enforcing PAO — a memory safety violation, such as a control flow attack, may perform arbitrary, malicious computation within enclaves, making it infeasible to give any meaningful security guarantee. To that end, the EncLang restricts how references to objects are computed and stored. The language forbids out-of-bounds access and nullable pointers. The developer may allocate memory for either primitive types (cell β, where β is a machine word type) or aggregate structures of type struct and array. The array size must be declared statically, and the array is indexed using a special integer type idx⟨k⟩, which represents values from 0 to k − 1. All allocations are either local (in the current procedure's stack frame), global, or on the heap (managed with malloc and free). Though we intend to relax this restriction in future work, we force the programmer to specify the type of the object (which also specifies its size) as an argument to malloc, in lieu of introducing runtime bounds checks. Allocations (via local, global, and malloc) bind the object's name to a reference that points to the base of the object. The $\rightsquigarrow$ operator converts a reference to a struct into a reference to the named field, and the [.] operator returns a reference to the indexed array element (if the type checker can prove that the index's type cannot allow an out-of-bounds access). These are the only two ways of computing references to objects in memory (pointer arithmetic is disallowed), and we also prevent references from being stored in memory to keep alias analysis simple yet precise (though we intend to relax this in future). Once a reference to a cell is obtained, deref and store is used to read and write a machine word.

A program may save intermediate results on the local stack using let statements and local declarations. A procedure (including malloc, free) is invoked using a call statement, and execution terminates within the procedure at the ret statement. Our type checker guarantees that references to stack allocated objects are not returned from a procedure, thus eliminating the possibility of a dangling pointer.

A standard feature of information flow type systems is a secrecy type, which has two values: public ($\bot$) or secret ($\top$). EncLang allows both values (i.e. cell) and references to have type $\bot$ or $\top$ — a secret reference indicates that the pointer is computed using secrets e.g. a[i], where i is secret. The type checker uses these types to identify illegal information flows (e.g. storing a secret value in a public state variable), and the PAO enforcement algorithm (§ 4.1-§ 4.3) uses these types to identify the secret-dependent branching conditions.

Advantages of using EncLang *for PAO.* The compiler can perform simple, yet precise, alias analysis to determine the set of objects that are potentially targeted by a store or deref.This is because the language provides limited constructs ($\rightsquigarrow$ and [.]) to compute pointers, and such pointers are always computed in a small number of registers and never saved in memory. In principle, we can relax these language restrictions, and develop sophisticated alias analysis, value analysis, etc. as part of the PAO-enforcing compilation. However, we find these challenges to be orthogonal to the problems addressed in this work.

Const

	n	::=	$\mathbb{N}$
	c	::=	true | false | 0bv8 | . . . | 18446744073709551615bv64

Vars

	v	::=	id

Ops

	$\otimes$	::=	$+ \mid \gg \mid \ll \mid * \mid = \mid < \mid \wedge \mid \vee \mid \neg \mid \ldots$

Region

	r	::=	local | global | heap

Types

	l	::=	public($\bot$) | secret($\top$)
	τ	::=	ref$\langle l \rangle$ r α | β
	β	::=	bool$\langle l \rangle$ | sint$\langle n, l \rangle$ | uint$\langle n, l \rangle$ | idx$\langle n, l \rangle$
	α	::=	array n α | struct{id : α, . . . , id : α} | cell $\beta\langle l \rangle$

Expr

	e	::=	$v \mid c \mid e \otimes e \mid e \rightsquigarrow$ id $\mid e[e] \mid$ deref e

Stmt

	s	::=	s ; s | local $v : \alpha$ in s | let $v = e$ in s | store e e | call v = id(e, . . .) in s | ret e | if (e) {s} else {s} | while(e) {s}

Prog

	p	::=	proc id($v : \tau$, . . . , $v : \tau \rightarrow \tau$){$s$} | var $v : \alpha$ | p; . . . ; p

Figure 5: EncLang **syntax**

4.5 Type Checking of EncLang programs

Figure 6 presents a type system for EncLang. The type checker flags information flow violations that a compiler cannot automatically repair e.g. assigning secrets to public state variables. A typing judgment for a well-typed statement s has the form $\Gamma,\ l_c \vdash_s s$, where Γ is the typing environment and l_c is the secrecy context: true if the statement occurs within a secret-based conditional. A well-typed expression e has a typing judgment $\Gamma \vdash_e e : \tau$, where τ is the inferred type. To check for valid information flows, we define a subtyping relation $\sqsubseteq$: $\bot$ is a subtype of $\top$, which creates a lattice with the join operator $\sqcup$: $l_1 \sqcup l_2$ is equal to $\bot$ if $l_1 = \bot \wedge l_2 = \bot$, and $\top$ otherwise.

The typing rules for LOAD and STORE mandate that deref and store recieve a reference to a primitive (cell) type, which is computed using a combination of array indexing and struct field accesses. In the case of store statements, the type checker checks that the secrecy type of reference (l_r) and the secrecy type of the written value (l_d) are subtypes of the referenced cell's secrecy type (l_d) — a secret reference can be thought of as a secret-based choice over a set of cells, and therefore must not be used to modify a public cell. The type system also checks that no store is made to a public cell in a secret context, which is a standard feature of type systems for

$$\frac{\upsilon \mapsto \tau \in \Gamma}{\Gamma \vdash_e \upsilon : \tau} \qquad \frac{\mathsf{id} \mapsto \tau_1, \ldots, \tau_n \to \tau \in \Gamma}{\Gamma \vdash_e \mathsf{id} : \tau_1, \ldots, \tau_n \to \tau} \qquad \frac{\exists r.\ \Gamma \vdash_e e : \mathsf{ref}\langle l_r \rangle\ r\ \mathsf{cell}\ \beta\langle l_\upsilon \rangle}{\Gamma \vdash_e \mathsf{deref}\ e : \beta\langle l_r \sqcup l_\upsilon \rangle}\ \text{LOAD}$$

$$\frac{\Gamma \vdash_e e : \mathsf{ref}\langle l \rangle\ r\ \mathsf{struct}\ \{\ldots, \mathsf{id} : \alpha, \ldots\}}{\Gamma \vdash_e e \rightsquigarrow \mathsf{id} : \mathsf{ref}\langle l \rangle\ r\ \alpha}\ \text{STRUCT} \qquad \frac{\Gamma \vdash_e e_a : \mathsf{ref}\langle l \rangle\ r\ \mathsf{array}\ n\ \alpha \qquad \Gamma \vdash_e e_i : \mathsf{idx}\langle n', l' \rangle \qquad n' \leq n}{\Gamma \vdash_e e_a[e_i] : \mathsf{ref}\langle l \sqcup l' \rangle\ r\ \alpha}\ \text{ARRAY}$$

$$\frac{\exists r.\ \Gamma \vdash_e e_a : \mathsf{ref}\ r\langle l_r \rangle\ \mathsf{cell}\ \beta\langle l_\upsilon \rangle \qquad \Gamma \vdash_e e_d : \beta\langle l_d \rangle \qquad l_d \sqcup l_r \sqcup l_c \sqsubseteq l_\upsilon}{\Gamma,\ l_c \vdash_s \mathsf{store}\ e_a\ e_d}\ \text{STORE} \qquad \frac{\Gamma \vdash_e e : \tau \qquad \neg\exists \alpha, l.\ \tau \neq \mathsf{ref}\langle l \rangle\ \mathsf{local}\ \alpha}{\Gamma,\ \bot \vdash_s \mathsf{ret}\ e}\ \text{RET}$$

$$\frac{\Gamma[\upsilon \mapsto \mathsf{ref}\langle \bot \rangle\ \mathsf{local}\ \alpha],\ \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathsf{local}\ \upsilon : \alpha\ \mathsf{in}\ s}\ \text{LOCAL-BIND} \qquad \frac{\Gamma \vdash_e e : \tau \qquad \Gamma[\upsilon \mapsto \tau],\ l_c \vdash_s s}{\Gamma,\ l_c \vdash_s \mathsf{let}\ \upsilon = e\ \mathsf{in}\ s}\ \text{LET-BIND}$$

$$\frac{\Gamma \vdash_e e : \mathsf{bool}\langle l \rangle \qquad \Gamma,\ l_c \sqcup l \vdash_s s_1 \qquad \Gamma,\ l_c \sqcup l \vdash_s s_2}{\Gamma,\ l_c \vdash_s \mathsf{if}\ (e)\ \{s_1\}\ \mathsf{else}\ \{s_2\}}\ \text{COND} \qquad \frac{\Gamma \vdash_e e : \mathsf{bool}\langle \bot \rangle \qquad \Gamma, \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathsf{while}(e) s}\ \text{WHILE}$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \ldots \Gamma \vdash_e e_n : \tau_n \qquad \Gamma \vdash_e \mathsf{id} : \tau_1, \ldots, \tau_n \to \tau \qquad \Gamma[\upsilon \mapsto \tau],\ \bot \vdash_s s}{\Gamma,\ \bot \vdash_s \mathtt{call}\ \upsilon = \mathsf{id}(e_1, \ldots, e_1)\ \mathsf{in}\ s}\ \text{CALL-BIND} \qquad \frac{\Gamma,\ l_c \vdash_s s_1 \qquad \Gamma,\ l_c \vdash_s s_2}{\Gamma,\ l_c \vdash_s s_1; s_2}\ \text{SEQ}$$

$$\frac{\forall\ \mathsf{var}\ \upsilon : \alpha \in p.\ \upsilon \mapsto \mathsf{ref\ global}\ \alpha \in \Gamma \qquad \Gamma[\upsilon_1 \mapsto \tau_1, \ldots, \upsilon_n \mapsto \tau_n],\ \bot \vdash_s s \qquad \text{each path in id ends in ret } e : \tau}{\Gamma \vdash_p \mathsf{proc\ id}(\upsilon_1 : \tau_1, \ldots, \upsilon_n : \tau_n \to \tau)\{s\}}\ \text{PROC}$$

$$\frac{\forall\ \mathsf{proc\ id}(\upsilon_1 : \tau_1, \ldots, \upsilon_n : \tau_n \to \tau)\{s\} \in p.\ \Gamma \vdash_p \mathsf{proc\ id}(\upsilon_1 : \tau_1, \ldots, \upsilon_n : \tau_n \to \tau)\{s\})}{\vdash p}\ \text{PROGRAM}$$

Figure 6: Typing rules for EncLang. Typing environment $\Gamma ::= \emptyset \mid \upsilon \mapsto \tau,\ \Gamma$

non-interference. A load (deref) produces a secret value if it uses either a secret reference or a reference to a secret-valued cell. Both CALL-BIND and RET rules enforce that call and ret statements are performed in a public context — forcing calls (including malloc and free) to occur in a public context allows the PAO enforcement to be modular. The COND rule checks that both branches are well-typed in the security context determined by the branch condition. The LOOP rule forbids loops with a secret branching condition — secret (number of) loop iterations causes the (number of) page accesses to depend on a secret, and therefore hinders PAO enforcement. The ARRAY rule checks that the index expression of type idx⟨k⟩ does not cause out-of-bounds access on an array of size n by requiring $k \leq n$.

4.6 Compiling EncLang to Typed Assembly Language

To produce PAO-satisfying code, the compiler must control the placement of stack-allocated objects and instructions, using the algorithms from § 4.1-§ 4.3. However, these algorithms operate at the level of read / write operations and machine instructions, which is produced after compilation. This circular dependency is broken by compiling the EncLang program in phases: 1) produce machine code with placeholders for the location of stack-allocated objects, 2) obliviate data accesses and optimize using MCMC sampling, which computes the location for stack-allocated objects, 3) obliviate code accesses, which computes the location for each instruction in the compiled program, and 4) assign placeholders from phase 1 using the locations computed in phase 2. The (nearly) entire implementation of the compiler's phase 1 is formalized in Figure 7; phases 2 and 3 are described in 4.1 and 4.3.

During phase 1, the compiler maintains a context $\phi = (\phi_L, \phi_G, \phi_B, \phi_\delta)$, which is read and modified during compilation to x86-64, as seen in Figure 7. Computed in phase 2, ϕ_L maps stack-allocated objects to locations in the current stack frame, specified as an offset relative to the frame pointer rbp — this is left as a placeholder in phase 1. ϕ_B tracks the location of bindings (produced by call and let statements), and is modified each time the compilation encounters such statement. ϕ_G maps globally-scoped objects to fixed, statically-computed addresses in the enclave address space.

The compiler is modular: each procedure p in the enclave is compiled independently by invoking $P_\phi[\![\mathrm{p}]\!]$. The procedure first pushes the callee-preserved registers on the stack, generates a code block called prologue (which we describe later), and invokes compilation on the procedure's body. A statement s is compiled using $S_\phi[\![s]\!]$, which recursively compiles the constituent statements (using $S[\![]\!]$) and expressions (using $E[\![]\!]$), while making use of context ϕ. For any expression e, $E_\phi[\![e]\!]$ stores the evaluation result in rax and is allowed to use rdx in any way. This process also produces $R[\![e_a]\!]$ and $W[\![e_a]\!]$ primitives, which is used in phase 2 to obliviate data accesses.

Recall that phase 2 produces ϕ_L, a mapping from stack-allocated object to its location on the stack frame (potentially many pages), by using stochastic optimization to assign multiple objects onto a page. However, the compiler cannot compute the base address of the local stack frame (as the procedure can be called via an arbitrary call chain), and this hinders our ability to implement ϕ_L. For this reason, the compiled program maintains two stacks: 1) a *bindings stack* used for storing bound variables (produced by a call or let) and intermediate results while evaluating expressions, and 2) a *locals stack* used for storing stack-allocated objects, which we will align at the page-boundary. The bindings stack bears resemblance to the stack used in a stack-based procedural language, and is accessed using the frame pointer rbp and stack pointer rsp. On the other hand, the locals stack is accessed relative to the register rsi, which we modify in the prologue to be the next page boundary, and it remains constant throughout the procedure — there is a caveat that if the procedure's stack space requirement can be met within the current page (e.g. it uses small objects), then we do not page-align

$S_\phi[\![\text{local } v : \alpha \text{ in } s]\!]$	=	$S_\phi[\![s]\!]$	
$S_\phi[\![\text{let } v = e \text{ in } s]\!]$ $\phi' = (\phi_L, \phi_G,$ $\quad \phi_B[v := \phi_\delta + 8],$ $\quad \phi_\delta + 8)$	=	$E_\phi[\![e]\!]$ mov rbx ϕ'_B[v] mov [rbx] rax $S_{\phi'}[\![s]\!]$	
$S_\phi[\![\text{store } e_a\ e_d]\!]$	=	$E_\phi[\![e_d]\!]$ mov rbx rax $E_\phi[\![e_a]\!]$ $W[\![e_a]\!]$	$\{\text{rbx} \mapsto \Gamma(e_d)\}$
$S_\phi[\![\text{call } v = \text{p}(e_1, \ldots, e_n) \text{ in } s]\!]$ $\phi' = (\phi_L, \phi_G, \phi_B[v := \phi_\delta + 8]$ $\quad, \phi_\delta + 8)$	=	$E_\phi[\![e_1]\!]$ push rax … $E_\phi[\![e_n]\!]$ push rax call p push rax $S_{\phi'}[\![s]\!]$	
$S_\phi[\![\text{ret } e]\!]$	=	$E_\phi[\![e]\!]$ mov rsp rbp pop rbp pop rdi pop rsi	
$S_\phi[\![\text{if } (e)\ \{s_1\} \text{ else } \{s_2\}]\!]$	=	$E_\phi[\![e]\!]$ test rax rax jz l_{else} l_{then}: $S_\phi[\![s_1]\!]$ jmp l_{end} l_{else}: $S_\phi[\![s_2]\!]$ l_{end}:	$\{\text{zf} \mapsto \Gamma(e = 0)\}$

$E_\phi[\![v]\!]$ for $v \in$ locals	=	mov rax ϕ_L[v]	$\{\text{rax} \mapsto \Gamma(v)\}$
$E_\phi[\![v]\!]$ for $v \in$ globals	=	mov rax ϕ_G[v]	$\{\text{rax} \mapsto \Gamma(v)\}$
$E_\phi[\![v]\!]$ for $v \in$ bindings	=	mov rax ϕ_B[v] mov rax [rax]	$\{\text{rax} \mapsto \Gamma(v)\}$
$E_\phi[\![\text{deref } e]\!]$	=	$E_\phi[\![e]\!]$ $R[\![e]\!]$	$\{\text{rax} \mapsto \Gamma(e)\}$ $\{\text{rax} \mapsto \Gamma(\text{deref } e)\}$
$E_\phi[\![e_1 \otimes e_2]\!]$	=	$E_\phi[\![e_2]\!]$ push rax $E_\phi[\![e_1]\!]$ pop rdx $\otimes$ rax rdx	 $\{\text{rdx} \mapsto \Gamma(e_2)\}$ $\{\text{rax} \mapsto \Gamma(e_1 \otimes e_2)\}$
$E_\phi[\![e \rightsquigarrow \text{id}]\!]$	=	$E_\phi[\![e]\!]$ add rax off(Γ(e), id)	 $\{\text{rax} \mapsto \Gamma(e \rightsquigarrow \text{id})\}$
$E_\phi[\![e_a[e_i]]\!]$	=	$E_\phi[\![e_i]\!]$ imul rax sz(Γ(e_a)) push rax $E_\phi[\![e_a]\!]$ pop rdx add rax rdx	 $\{\text{rax} \mapsto \text{idx}\langle \text{sz}(\Gamma(e_a)) * n \rangle\}$ $\{\text{rdx} \mapsto \text{idx}\langle \text{sz}(\Gamma(e_a)) * n \rangle\}$ $\{\text{rax} \mapsto \Gamma(e_a[e_i])\}$
$P_\phi[\![\text{proc p}(v_1, \ldots, v_n)\{s\}]\!]$ $\phi' = (\phi_L, \phi_G,$ $\quad \phi_B[v_1 := \phi_\delta - 8, \ldots$ $\quad\quad v_n := \phi_\delta - 8 * n],$ $\quad 0)$	=	push rsi push rdi push rbp mov rbp rsp ⟨prologue⟩ $S_{\phi'}[\![s]\!]$	

Figure 7: Compilation of EncLang to Typed Assembly Language. For any typing derivation $\Gamma \vdash_e e : \tau$, we use $\Gamma(e)$ to denote τ.

rsi. This invariant on rsi enables the compiler to statically layout the objects to comply with the ϕ_L mapping and satisfy PAO.

Section 5 describes an independent verifier which certifies the compiled machine code. We follow the typed assembly language paradigm [13] of: 1) a strongly-typed source language, EncLang, 2) a type-preserving compiler, and 3) a strongly-typed assembly language (TAL from here on). As seen in the right-most column of Figure 7, each update to a register accompanies an annotation assigning its new type, derived using the type inference rules [20]. The type annotations on registers, which are used for computing both references and values, simplifies information flow tracking and alias analysis of the otherwise unstructured, untyped machine code.

4.7 Supporting Heap Allocation and Procedure Calls

Since malloc enjoys complete control over the placement of objects within the heap, we obliviate heap accesses by performing $\hat{R}[\![.]\!]$ and $\hat{W}[\![.]\!]$ in the other branches of a secret conditional, using the technique described in section 4.1. Procedure arguments are treated similarly, the compiler has no knowledge of their location. The $\hat{R}[\![.]\!]$ and $\hat{W}[\![.]\!]$ requires the size of the object to be statically known. Therefore, our malloc and free routines require the type of the requested object.

5 VERIFYING PAO

The verifier proves that for any pair of executions of the enclave binary (that only differ in secret values), the sequence of page accesses must be equivalent, where two accesses are equivalent if they target the same page and have same type (r/w/x). The verifier independently analyzes each procedure because the compiler is modular and does not optimize globally. For each procedure, the verifier must:

1. Enumerate secret-dependent paths. Enumerate all paths in the procedure such that each constituent path represents a unique evaluation of a secret-dependent conditional — number of paths is worst case exponential in the number of secret-dependent conditionals within the procedure. The compiler assists this step by providing secrecy types of CPU flags at all conditional jumps in the TAL program (see Figure 7). Furthermore, the lack of indirect jumps in the binary enables simple, yet precise control flow analysis.

2. Identify sequence of memory accesses. : The verifier computes the sequence of memory accesses that the CPU performs on each path. This step is purely syntactic, and is implemented verbatim as the definition of PAO in § 3.3.

3. Identify target page(s) for each access. : The verifier computes the exact code page for each instruction, which is given verbatim in the enclave executable. Identifying the target pages for data accesses ordinarily necessitates an alias analysis which identifies the object(s) targeted by a reference, and recovering the mapping from objects to pages. We circumvent these analyses with a key insight: for any reference e, the verifier only needs to determine the statically-identifiable object $si(e)$ (defined in section 4.1), for which the typing annotations are sufficient. In other words, the verifier can establish the equivalence of any two page accesses, which use references e_1 and e_2, knowing only $si(e_1)$ and $si(e_2)$. The reasoning is a combination of 1) our primitives $R[\![e]\!]$, etc. use only

the knowledge of $si(e)$ to generate the memory accesses, 2) for any reference e, the semantics of EncLang ensures that $si(e)$ evaluates to the same object in all paths, and 3) the verifier is able to infer when different local objects ($si(e_1) \neq si(e_2)$) are mapped to the same page by leveraging the fact that all local objects are addressed relative to a fixed frame pointer.

Having performed these steps, the verifier asserts that the sequence of page accesses is equivalent in all the enumerated paths. Furthermore, to avoid trusting the compiler, the verifier also proves validity of the typing annotations using a set of simple rewrite rules.

6 EVALUATION

Benchmark	Code Size (no PAO)	Code Size (PAO)	Data Size (no PAO)	Data Size (PAO)
k-means	1638 B	1668 B	1784 B	1784 B
Decision Tree	1058 B	3082 B	416 B	416 B
SVM	1368 B	1408 B	2008 B	2008 B
CNN Classifier	1637 B	1667 B	45312 B	45312 B
IDCT (1 dim)	4574 B	10725 B	592 B	592 B
IDCT (2 dim)	7424 B	13575 B	664 B	664 B
AES Blk Cipher	5173 B	5203 B	488 B	488 B

Table 1: Summary of results.

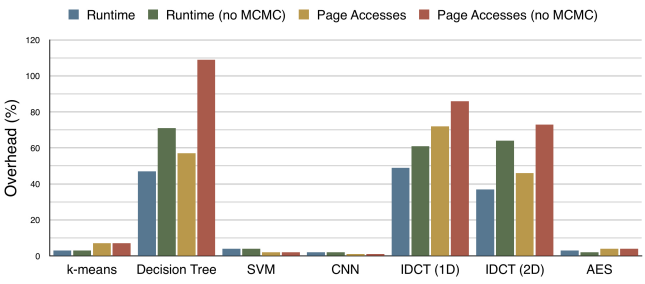


Figure 8: Overhead in Runtime and Page Accesses

To study the performance impact of automatic PAO enforcement, we implement an open-source toolchain consisting of a compiler from EncLang to TAL, and a verifier for certifying PAO on the output TAL. Along with the TAL, the toolchain produces executables that we run natively on a 3.2 GHz 6th Generation Intel CPU (with SGX instruction set enabled and 96 MB available for enclave memory).

We evaluate this toolchain on several enclave programs that compute on sensitive data. We sample standard machine learning algorithms: k-means clustering, training of SVM classifier (based on a cache oblivious algorithm from [14]), decision tree evaluation (Figure 1), and a convolutional neural network (CNN). In the case of k-means clustering, the input points and the k=10 trained clusters must be kept secret. The training of SVM classified must ensure that the learned weights are kept secret. The CNN (trained offline) must ensure that the image (to be classified) must be kept secret. We also experiment with the inverse discrete cosine transform (IDCT) routine (both one and two dimensions) from a JPEG decoder, whose page fault patterns were exploited by Xu et al. [21] to infer edges in the secret input images. To compare with [17], we also evaluate on the AES block cipher (encryption only).

The compiler takes roughly 2-3 seconds to compile each of these benchmarks, most of which is spent in the MCMC optimization. The verifier uses the typing annotations generated by the compiler in order to produce the proof, and this takes under 1 second for all of these benchmarks. Table 1 compares vanilla compilation with PAO-enforcing compilation, with respect to memory consumption for code and data pages. Across all the benchmarks, we observe an average of 81 % increase in code size, and 0 % increase in memory requirements because the compiler successfully rearranged local objects without introducing any padding space. That being said, the runtime overhead is significantly lower than code size overhead (81 %) because the added x64 instructions are distributed across multiple branches.

Figure 8 reports the performance overheads for runtime and number of page accesses, using standard datasets (e.g. UCI Repository [1]) for machine learning programs and $10^6$3 invocations with randomly generated inputs for IDCT and AES programs. The overhead denotes the increase in these metrics when enforcing PAO, and is reported after averaging over 10 runs on a large dataset, or a million invocations . We observe a non-negative overhead (%) because our algorithm instruments dummy accesses, which lead to additional data and code accesses (to fetch the added instructions). In general, we find that programs with more conditional branches (in a secret context) incur higher code size overheads because the EncLang compiler places dummy code and data accesses to determinize the page access across all branches, which leads to additional x86 instructions. Specifically, IDCT shows higher runtime overheads because the libjpeg implementation skips the complex computation when the input image satisfies a condition [21], and such optimization violates PAO because the attacker observes different page accesses based on secret input. Similarly, the decision tree classification incurs a high overhead because the oblivious implementation must perform a fixed number of tree traversals for all inputs. On the other hand, we find that k-means, SVM, CNN classification, and AES incur low overheads as the computation is data-intensive and primarily occurs outside of secret conditionals. Specifically, CNN and AES show negligible overhead as all array accesses use public indices, and loops use constant bounds; our compiler infers that they have no secret-dependent code or data accesses.

7 RELATED WORK

Moat [19] and SlashConf [18] verify the enclave binaries to prove an absence of explicit leaks; side channels are out of their scope.

Shinde at al. [17] develop an instrumentation scheme (at the LLVM IR level) for enforcing page fault obliviousness — computational security definition as opposed to non-interference — of programs written in a subset of C/C++. Their approach, termed deterministic multiplexing, copies all code and data blocks at the same level of the execution "tree" to a temporary page, and dynamically selects the appropriate code and data based on the current program path. Although [17] addresses the same side channel attack as this paper, there are several important differences. First, we protect against a stronger adversary that can observe all page accesses, whereas the attacker in [17] only observes changes in page accesses i.e. the attacker cannot count accesses within a single page. A realistic attack scenario can invalidate a page table entry

at any time (e.g. OS getting interrupted by a device). Therefore, the defense in [17] potentially leaks a bit of information on each page access, which is problematic for long-running enclaves. Second, we develop a binary verifier to certify PAO. On the contrary, [17] makes unspecified assumptions such as memory safety — a control flow exploit can bypass their instrumentation, and worse, spill secrets directly to untrusted memory. Furthermore, [17] incurs a large software TCB comprising the C compiler, LLVM assembler, and their instrumentation scheme. Our software TCB contains just the verifier. Finally, [17] obliviates by transforming the compiled LLVM program. While this allows them to target mainstream languages, their scheme incurs 705x average runtime overhead — they reduce this overhead using developer annotations, which may compromise soundness. We show that by carefully designing the semantics and compiler of EncLang, we can optimize the PAO enforcement to incur an average 49% overhead across various benchmarks.

Ohrimenko et al. [14] manually develop machine learning algorithms that guarantee data-obliviousness at a cache-line granularity; their ideas inspired our primitives for oblivious dummy accesses. Cache side-channel defenses are complementary to our work because pages and cache sets are addressed by disjoint bits in a virtual address.

Oblivious RAM (ORAM) [5] protects against side channel leaks via the program's memory access patterns. Liu et al. [11] formalize memory trace obliviousness, and develop a compiler for producing memory trace oblivious programs by partitioning code and data across multiple ORAM banks for efficiency; in a follow-up work, Liu et al. [12] develop ObliVM to compile high-level source programs to an oblivious representation that leverages ORAMs to efficiently perform dynamic memory accesses (in lieu of performing a linear scan). GhostRider [10] presents a co-designed compiler and hardware ORAM for memory trace oblivious execution. Although these techniques provide stronger guarantees than our work, they require a novel hardware platform with ORAM support, whereas we target commodity SGX machines. An interesting question arises whether an ORAM controller can be implemented within an SGX enclave in our threat model. As [17] explains, ORAM constructions require a private stash for shuffling data blocks, whereas our attacker can observe accesses to all pages in memory. Furthermore, to adapt the ORAM algorithm (e.g. Path ORAM) in our compiler, we need to "compose" the ORAM logic with the enclave program, effectively evaluating the ORAM logic on each instruction. Our compiler achieves obliviousness efficiently by using the type system and program behavior to guide several optimizations.

8 CONCLUSION

We formalize page-access obliviousness (PAO) for enclave programs, and develop a toolchain for enforcing PAO automatically. The toolchain comprises 1) a compiler for EncLang that automatically enforces PAO, 2) a stochastic optimization to reduce the runtime overhead of the compiler, and 3) a binary verifier to certify the output machine code. The toolchain provably guarantees PAO while achieving a tiny trusted computing base, which only includes the SGX processor and the verifier's implementation.

ACKNOWLEDGMENTS

This research was supported by the NSF STARSS grant 1528108 and SRC contract 2638.001. We gratefully acknowledge the anonymous reviewers for their insightful feedback.

REFERENCES

[1] UCI Machine Learning Repository. https://archive.ics.uci.edu/ml.
[2] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, July 2008.
[3] I. S. Diatchki and M. P. Jones. Strongly typed memory areas programming systems-level data structures in a functional language. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 72–83, New York, NY, USA, 2006. ACM.
[4] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt free ivory. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 189–200, New York, NY, USA, 2015. ACM.
[5] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.
[6] Intel Software Guard Extensions Programming Reference. Available at https://software.intel.com/sites/default/files/329298-001.pdf, 2014.
[7] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.
[8] Joanna Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. https://github.com/Cr4sh/ThinkPwn.git.
[9] Lenovo ThinkPad System Management Mode arbitrary code execution 0day exploit. Available at https://github.com/Cr4sh/ThinkPwn.git.
[10] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, Mar. 2015.
[11] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, CSF '13, pages 51–65, Washington, DC, USA, 2013. IEEE Computer Society.
[12] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA, 2015. IEEE Computer Society.
[13] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.
[14] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, Aug. 2016. USENIX Association.
[15] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGPLAN Not.*, 48(4):305–316, Mar. 2013.
[16] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *S&P*, 2015.
[17] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 317–328, New York, NY, USA, 2016. ACM.
[18] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 665–681, New York, NY, USA, 2016. ACM.
[19] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS*, 2015.
[20] R. Sinha, S. Rajamani, and S. A. Seshia. A compiler and verifier for page access oblivious computation. Technical Report UCB/EECS-2017-124, EECS Department, University of California, Berkeley, Jul 2017.
[21] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.