On Building the Data-Oblivious Virtual Environment

Tushar M. Jois*, Hyun Bin Lee†, Christopher W. Fletcher†, and Carl A. Gunter†
*Johns Hopkins University: jois@cs.jhu.edu
†University of Illinois at Urbana-Champaign:
{lee559, cwfletch, cgunter}@illinois.edu

***Abstract*—Users can improve the security of remote communications by using Trusted Execution Environments (TEEs) to protect against direct introspection and tampering of sensitive data. However, for applications coded in high-level, interpreted languages such as R, Python, and Ruby, this creates a trade-off between programming convenience versus the risk of attacks using microarchitectural side channels. In our NDSS '21 paper, we introduced the *Data-Oblivious Virtual Environment (DOVE)*, which leverages a *Data-Oblivious Transcript (DOT)* that is explicitly designed to support computation that excludes side channels to bridge a complex programming environment (like R) with evaluation on a Trusted Execution Environment (TEE).**

In this companion work, we take a more experiment-centric lens to the creation of DOVE. First, we show how our experiments revealed a number of subtle side-channel vulnerabilities in the R language. Then, we discuss how these experiments impacted the design of DOVE, the first side-channel resistant R programming stack. Finally, we experimentally evaluate the correctness, expressiveness, data-obliviousness, and efficiency of our prototype DOVE implementation, showing how DOVE can provide practical support for complex programming languages with usable performance and high security assurances against side channels. Along the way, we highlight relevant details about our methodology and lessons we learned through the process.

I. Introduction

Recent commercially-available Trusted Execution Environments (TEEs) such as Intel SGX [22], [36] and ARM TrustZone [4] have enabled significant progress towards the outsourcing of secure computation. Consider for example three competing drug companies investigating genomic factors for bipolar disorder. These companies would like to share their proprietary genome data and run a controlled study that releases only agreed-upon information to the three participants. TEEs enable such use cases, without requiring trust in remote administrator software stacks such as operating systems, using a combination of hardware-level isolation and cryptography.

The long-term vision pursued by TEE-based software systems (e.g., [7], [17]) is to bring TEE-level security to the masses where it can be used by data scientists familiar with existing high-level languages such as R, Ruby, and Python, but who may not have much background in security [14].

Here, we face a challenging problem. To achieve complete security from untrusted software, it is well known that TEE software must be hardened to block a plethora of microarchitectural side channels (e.g., [11], [65], [73], [76]). Yet, existing software-based techniques to block these channels—coming from a rich line of research in data-oblivious/constant-time programming [8], [20], [51], [59]—fall short of protecting existing high-level language stacks such as R, Ruby and Python. Specifically, these techniques typically require experts to manually code core routines [8], [9], require the use of custom domain-specific languages [13], [63], or only apply to close-to-metal compiled languages [51], [59].

Modern high-level languages, however, require complex stacks to support interpreted execution, just-in-time compilation, etc. As a case-in-point, the popular R stack features almost a million lines of code written in a combination of C, Fortran, and R itself [58]. Subtle issues in any of this code create security holes.

In our NDSS '21 paper [42], we aimed to tackle this problem, with the goal of extending data-oblivious/constant-time techniques to apply to existing high-level, interpreted languages, thus enabling TEE-level security for non-experts. We arrived at our solution, the "Data-Oblivious Virtual Environment (DOVE)", after discovering a number of subtle side-channel vulnerabilities in the R language through experimentation and careful study. These bugs arose in R code that, at first blush, appeared to be data-oblivious and thus harderned to side-channels. The results of our analysis of R guided the development of DOVE, as well as the experiments we performed to validate it.

Our key strategy and insight in DOVE is this: *if key observable features of a computation are truly independent of sensitive data, then that computation can be carried out with a collection of stand-ins ("pseudonyms") for the data*.

To capitalize on this idea, we perform computation in DOVE in two phases. In the first phase, we run the target computation on pseudonyms in the chosen high-level language, like R or Python. Since there is no sensitive data present, this stage cannot leak sensitive information. We instrument the programming stack so that this evaluation on pseudonyms outputs what we call a "Data-Oblivious Transcript (DOT)". The DOT is akin to a straight-line code representation of the original program, i.e., the transcript of operations performed when the program is evaluated on the pseudonyms. In the second phase of our computation, we evaluate the DOT on a small Trusted Computing Base (TCB) that runs within a TEE. This TEE contains the sensitive data, which is used in place of the pseudonyms. Protecting sensitive data *after* the DOT is constructed is relatively straightforward. Since the DOT is similar to straight-line code, the TEE need only apply simple transformations to evaluate it in a data-oblivious

Learning from Authoritative Security Experiment Results (LASER) 2021
25 February 2021, Virtual
ISBN 1-891562-71-1
https://dx.doi.org/10.14722/laser-ndss.2021.23056
www.ndss-symposium.org

fashion on real hardware. In the worst case, where the original computation was actually data dependent on the pseudonyms, the resulting computation in the TEE may be functionally incorrect but leaks no sensitive information.

Our proof-of-concept DOVE implementation[1] is two-fold: a DOVE *frontend* that translates programs written in the R language to a DOT representation, and a DOVE *backend* that evaluates the DOT on sensitive data inside of an Intel SGX enclave. We validate DOVE in four domains: correctness, expressiveness, data-obliviousness, and efficiency. We experimentally compare DOVE's results in these domains to those of base R, using a third-party library of genomics analysis algorithms written in R [15] applied on a real-world genomic dataset consisting of three populations of honeybees [6].

1) This paper: We present this work as a supplement to our main research contributions in [42]. While the structure of this paper is largely similar to that of [42], content has been added, removed, and reorganized as to be more useful for an experiments-focused reader. We present the experimental techniques we developed for identifying side-channel vulnerabilities in R, and discuss how these vulnerabilities influence the design of DOVE. This work also contains more information about the experiments we used to validate the runtime security (i.e., data-obliviousness) of DOVE, as well as its expressiveness and efficiency. We also include a new section on the lessons learned in building DOVE. Please refer to our NDSS '21 paper [42] for additional details on content omitted from this work.

II. Background

A. Programming in R

R is a statistical language that provides convenient interfaces for computations on arrays and matrices. Most function calls including primitive operators like addition and subtraction perform element-wise operations on array-like values.

1) Computation in R: R is an interpreted language [58], and its interpreter is written mostly in C and to a lesser extent Fortran and R itself. Every object is represented with an S-expression [46] such that interpreter parses R statements into S-expressions. The S-expressions are then evaluated and dispatched to the corresponding library functions written in C. Each C function runs on hardware as a compiled binary object. Thus, analyzing code written in R is more complex than analyzing code that is directly compiled and run on hardware (e.g. C, C++).

2) Not Applicable (NA): R represents `null`-like, empty values with **`NA`**, the representation of which depends on the datatype. A real-valued S-expression in R is represented with a IEEE 754 **`double`**; `NA_REAL` is defined with the special double value **`NaN`** with a specific lower word (**`1954`**). The interpreter treats **`NA`** differently from other values, even from **`NaN`**. Integer and logical (i.e., boolean) S-expressions are implemented with an **`int`** type, so R reserves the lowest integer value `INT_MIN` for the representation of `NA_INTEGER` and `NA_LOGICAL`.

[1]Code and benchmarks from [42] are available at https://github.com/dove-project/

B. Microarchitectural Side-Channel Attacks

Microarchitectural (shortened as "$\mu Arch$") side-channel attacks are a class of privacy-related vulnerabilities in which a sensitive program's hardware resource usage leaks sensitive information to an adversary co-located to the same (or a nearby) physical machine [30]. Over the years, numerous hardware structures—cache architectures [56], [77], [79], [80], branch predictors [1], [27], pipeline components [3], [5], [33] and others [26], [32], [49], [57], [73], [76]—have been found to leak information in this way. Many of these attacks require that the attacker only share physical resources with the victim (e.g., Prime+Probe and the cache [45], [56] or Drama and the DRAM row buffer [57]), as opposed to sharing virtual memory with the victim (e.g. [79]).

C. Enclave Execution and Intel SGX

Enclave execution [68], such as with Intel SGX [36], protects sensitive applications from direct inspection or tampering from supervisor software. That is, the OS, hypervisor and other software are considered to be the attacker [11], [31], [35], [50], [55], [59], [62], [65], [73], [81], who will be referred to as the *SGX adversary* for the rest of the paper. To use SGX, users partition their applications into enclaves at some interface boundary. For example, prior work has shown how to run whole applications with a LibOS [7], [17], containers [64], and data structure abstractions [62] within enclaves. At boot, hardware uses attestation via digital signatures to verify the user's expected program and input data are loaded correctly into each enclave. Isolation mechanisms implemented in virtual memory protect enclave integrity and confidentiality during execution.

SGX uses the Enclave Page Cache (EPC) to store enclave application code and data. The EPC is stored in a protected region of memory known as Processor-Reserved Memory (PRM). The processor prevents other system components from reading the PRM with the help of another component, the Memory Encryption Engine (MEE), that provides encryption and integrity protection for the PRM [47]. The EPC has a fixed size of 64 or 128 MB, shared among all enclaves [38]. For applications requiring more memory, SGX uses an EPC paging mechanism supported by the SGX OS driver. Specifically, the OS can move pages out of/into the EPC and manipulate them as if they were regular pages from a demand-paging perspective. For security, pages moved out of/into the EPC are transparently encrypted/decrypted and integrity checked by the SGX hardware [36], [47].

1) Side-channel amplification: Despite providing strong virtual isolation, SGX enclave code is still managed by untrusted software. Prior work has shown how this exacerbates the side-channel problem described in Section II-B.

First, SGX does not provide any physical isolation. Thus, nearly all of the $\mu Arch$ side-channel attacks discussed in Section II-B immediately apply in the SGX setting.

Second, importantly, the OS-level attacker has significant control over the enclave's execution and the processor hardware and thus can orchestrate finer-grain, lower-noise attacks than would otherwise be possible. For example, controlled side-channel attacks [76] and follow-on work [73] provide a zero-noise mechanism for an attacker to learn a victim's memory access pattern at page (or sometimes finer) granularity. A

line of work has further shown how the attacker can effectively single-step, and even replay, the victim to measure fine-grain information such as cache access pattern and arithmetic unit port contention [11], [31], [34], [35], [50], [65], [72].

2) Threat model: Our goal is to prevent arbitrary non-SGX enclave software from learning anything about the users' data, other than non-sensitive information about the data such as its bit length. Given SGX's architecture, this implies protecting user data from leaking over arbitrary non-speculative $\mu Arch$ side channels (Section II-B), given the powerful SGX adversary described above.We do not defend against hardware attacks such as power analysis [40], EM emissions [53], compromised manufacturing (*e.g.*, hardware trojans [78]), denial of service attacks, or speculative execution attacks [39] beyond default SGX protections.

Note, when we refer to *trusted computing base* (TCB) we mean the DOVE software that must function as intended—i.e., be free of logic bugs and control-flow hijacking vulnerabilities—for security to hold.

D. Data-Oblivious Programming

Data-oblivious (sometimes called "constant-time" in the hardware setting) programming is a way to write programs that makes program behavior independent of sensitive data, with respect to the side channels discussed in Section II-B [2], [5], [8]–[10], [13], [16], [20], [23]–[25], [28], [44], [44], [48], [51], [52], [55], [59], [62], [63], [66], [67], [71], [74], [82]–[84]. In the hardware setting, what constitutes data-oblivious execution depends on the intended adversary. In the SGX setting, we must assume a powerful adversary that can monitor potentially any $\mu Arch$ side channel as described in Section II-C.

Thus, prior works that try to achieve data obliviousness in an SGX context [2], [25], [28], [48], [55], [59], [62], [63], [84] implement computation using only a carefully chosen subset of arithmetic operations (e.g., bitwise operations), conditional moves, branches with data-independent outcomes, jumps with non-sensitive destinations, and memory instructions with data-independent addresses. For example, an `if` statement with a sensitive predicate is implemented as straight line code that executes both sides of the `if` and uses a data-oblivious ternary operator (such as the x86 `cmov` instruction or the CSWAP operation) to choose which result to keep.

III. The (Lack of) Data-Obliviousness of R

Our goal is to protect R programs (and by extension scientific computing) from the SGX adversary. As a starting point, imagine we try to run secure R code by moving the whole R stack into the SGX enclave (which is the approach taken by prior work [7], [17]). If R were data-oblivious, we could have security against the SGX adversary. However, we show that security is not guaranteed, by demonstrating subtle $\mu Arch$ side-channel attack vectors that come up in this approach.

A. Case Study

To evaluate the data obliviousness of R, we worked with an application of genomic data sharing to accurately represent the kinds of R scripts data scientists use. The specific application [6] aims to understand from genomics why honeybees from Puerto Rico are *gentle*, like European honeybees, even though they descend from *aggressive* Africanized honeybees from South America. Genomes from 30 honeybees were collected from Puerto Rico, Mexico, and the United States to provide a total of 90 genomes.

Overall, this honeybee study simulates the idea that three parties would like to derive critical information from their combined data set without the need for a trusted third party to consolidate the data. We did not work with truly sensitive data in this study, but the characteristics of the data and the data-sharing arrangement are essentially the same as would have been used in the hypothetical bipolar disorder study mentioned in the introduction. The honeybee data and code is available for download and will be a good benchmark for future studies of secuirty for genomics.

We reproduce the study in [6] with this data using R, but truncate the total number of samples to 60 (due to machine limitations). The study relies on R code drawn from a set of 13 genetics research programs [15] that implement important statistical measurements found in the literature [29], [54], [70], [75], totaling 478 lines of R code [15]. We refer to these scripts as our *evaluation programs*. We evaluate 11 of these evaluation programs as a part of our analysis, as they operate on numeric values (rather than character strings or symbols). The functionality of each of these programs is explained below.

- `allele_sharing` A program to calculate the allele sharing distance between pairs of individuals [29].
- `EHHS` A program to calculate the EHHS values for a given chromosome [70].
- `hwe_chisq` A program to test the significance of deviation from Hardy–Weinberg Equilibrium (HWE) using Pearson's Chi-Squared test.
- `hwe_fisher` A program to test the significance of deviation from HWE using Fisher's Exact test.
- `iES` A program to calculate the iES statistics [70]. The code calls `EHHS` in computing its statistics.
- `LD` A program to calculate D, D', r, χ^2, $\chi^{2\prime}$, which are statistics based on the frequencies of alleles in the input.
- `neiFis_multispop` A program to calculate inbreeding coefficients, F_{is} [54], for each sub-population from a given set of SNP markers.
- `neiFis_onepop` A program to calculate inbreeding coefficients, F_{is} [54], for the total population from a given set of SNP markers.
- `snp_stats` A program to calculate basic stats on SNPs, including: allele frequency, minor allele frequency, and exact estimate of HWE.
- `wcFstats` A program to estimate the variance components and fixation indices [75].
- `wcFst_spop_pairs` A program to estimate F_{st} (θ) values for each pair of sub-populations [75].

We perform our analysis of the data-obliviousness of R using the code snippet in Figure 1 as a guiding example. This code is found in four of the 13 evaluation programs, and three more feature similar snippets. We use R version 3.4.4, compiled with default flags, on a Ubuntu Linux 18.04.4 machine for this study.

Fig. 1: R code snippet. geno is a sensitive diploid dataset.

```
1  geno[(geno!=0) & (geno!=1) & (geno!=2)] <- NA
```

B. Example Walkthrough

geno is a set of samples made up of diploid Single Nucleotide Polymorphism (SNP) sequences. The database of samples is represented as an m by n matrix, where each column is one of n samples, each of which has m SNP positions. Each position in the matrix has a genotype, denoted as an integer **0**, **1**, or **2**. The sensitive data is the contents of geno, namely which genotype each SNP is for each sample. The matrix dimensions (m and n) are non-sensitive.

The line of code in Figure 1 sanitizes the input database: any entry that is not one of the three allowed genotypes is replaced with the special value **NA** (Section II-A). This occurs in real data due to noise in the sequencing process; in particular, 1.5% of the SNP entries in the honeybee dataset [6] are marked as **NA**. The code first computes element-wise filters `geno != 0`, `geno != 1`, `geno != 2`, each of which produces a matrix of booleans (a mask) indicating whether the condition is satisfied for each SNP position in each sample. The logical AND (**&**) performs element-wise AND of these 3 masks (producing a new mask) which is used to conditionally assign elements in geno to **NA**.

Given the above code, the adversary's goal is to learn the genotype at each SNP position—that is, whether the value of each cell in geno is **0**, **1**, **2**, or **NA**. Importantly, given no additional information about R's implementation, *the R-level code in Figure 1 follows guidelines for achieving data-obliviousness* (Section II-D), which would seemingly prevent leaking the above information. For example, it applies simple arithmetic/logical operations element-wise over matrices of non-sensitive size, performs a count over a subset of samples with a non-sensitive length, etc. Thus, combining each mask with **&** entails performing a data-independent number of simple logical operations (**&**); this is traditionally regarded as safe.

Yet, this code is not data-oblivious thanks to the transformations it undergoes in the R stack before reaching hardware.

In particular, the R interpreter transforms the line of code from R into C calls. When R interprets **&**, it invokes the C routine given in Figure 2a. This snippet takes different code paths, depending on the values of x1 and x2, which the SGX adversary can detect by single-stepping [72] or by replaying the victim [65] and measuring time, branch predictor state, etc. We investigate the side-channel characteristics of this with two types of analyses: instruction-level and processor-level. The following analyses apply well-established principles for writing constant-time and data-oblivious programs (Section II-D).

C. Instruction-level Analysis

We wish to experimentally verify the presence of such side channels in the R codebase. We can identify them at the assembly instruction level, as the C code that R functions call runs as a part of a compiled library. We cover both the static analysis of individual opcodes in the R binary, as well as dynamic analysis of execution traces of the binary for different input values.

1) Static opcode analysis: We identify the opcodes in the R binary, libR.so, using the objdump utility. This converts the compiled machine code into a human-readable opcode format. Then, we sweep over the objdump output, looking for vulnerable operations over data. In particular, we wish to find branches on sensitive data, which can leak control flow information and help an attacker reconstruct the secret.

Consider Figure 2b, which is the assembly for Lines 1 to 2 in Figure 2a. We note that the assembly shows a comparison (cmp) between the values stored in `rbp-0x58` (x1) and **0x0**, and `rbp-0x54` (x2)) and **0x0**. This constitutes a branchon sensitive data, as the code will take different paths through the code depending on the result of the computation (je, jne). In this case, the attacker learns if one of x1 or x2 equals 0. Since this **&** is applied to each SNP position of each sample in Figure 1, this information is leaked for every SNP position.

2) Dynamic execution trace analysis: We now show how this static analysis can be leveraged at runtime to leak the secret. We use the branch-trace-store execution trace recording mechanism [37] on our Intel Core i3-6100 CPU to counts the number of instructions executed at the assembly level for different inputs. ranch-trace-store hooks in GDB allow us to step through the program, counting instructions between breakpoints. Figure 3 for each possible input to **&**, as reported by brance-trace-store. Confirming the above explanation, we see that the instruction count equals 45 if and only if x1 equals 0. Thus, the adversary learns whether this is the case if it can monitor a function of the instruction count. Other cases leak other pieces of information such as whether both x1 and x2 equal 1.

Fig. 2: The R interpreter implementation of the **&** operator.

(a) C source code snippet of the **&** operator implementation.

```
1  if (x1 == 0 || x2 == 0)
2      pa[i] = 0;
3  else if (x1 == NA_LOGICAL || x2 == NA_LOGICAL)
4      pa[i] = NA_LOGICAL;
5  else
6      pa[i] = 1;
```

(b) The Intel-syntax x86-64 assembly for Lines 1 and 2 of the C code in Figure 2a, lightly edited for clarity.

```
; x1 in [rbp-0x58], x2 in [rbp-0x54]
a8: cmp   DWORD PTR [rbp-0x58],0x0 ; x1==0
ac: je    b4 ; if true, jump to pa[i]=0
ae: cmp   DWORD PTR [rbp-0x54],0x0 ; x2==0
b2: jne   cf ; if false, jump to else if
b4: mov   rax,QWORD PTR [rbp-0x50]
b8: lea   rdx,[rax*4+0x0]
c0: mov   rax,QWORD PTR [rbp-0x8]
c4: add   rax,rdx ; calc addr of pa[i]
c7: mov   DWORD PTR [rax],0x0 ; pa[i]=0
cf: ...
```

D. Intel PCM Analysis

Opcode and execution trace analysis is not sufficient to cover the diverse (and undocumented) set of potential $\mu Arch$ side channels, such as timing differences. We wish to show that the data dependent execution visible at the opcode layer can be verified by an attacker with access to side-channel information.

Fig. 3: The associated x86-64 instruction counts for different permutations of `x1` and `x2` fed as input to `&` in R.

Expression	Value	Instruction Count
`0 & 0`	`0`	45
`0 & 1`	`0`	45
`1 & 0`	`0`	47
`1 & 1`	`1`	54
`0 & NA`	`0`	45
`1 & NA`	`NA`	57
`NA & 0`	`0`	47
`NA & 1`	`NA`	53
`NA & NA`	`NA`	53

Fig. 4: Intel PCM Functions used for dynamic analysis.

Function Name	Criterion
`getCycles`	cycle counts
`getCyclesLostDueL3CacheMisses`	cycle counts, cache H/M
`getCyclesLostDueL2CacheMisses`	cycle counts, cache H/M
`getL2CacheHitRatio`	cache H/M
`getL3CacheHitRatio`	cache H/M
`getL3CacheMisses`	cache H/M
`getL2CacheMisses`	cache H/M
`getL2CacheHits`	cache H/M
`getL3CacheHitsNoSnoop`	cache H/M
`getL3CacheHitsSnoop`	cache H/M
`getL3CacheHits`	cache H/M
`getBytesReadFromMC`	bytes from/to MC
`getBytesWrittenToMC`	bytes from/to MC
`getIORequestBytesFromMC`	bytes from/to MC

Intel Processor Counter Monitor (PCM) is an Application Programming Interface (API) to monitor performance of Intel processors [21]. PCM offers various performance metrics, some of which are direct indicators of side-channel vulnerabilities. Such $\mu Arch$ measurements include cycle counts and L2/L3 cache hits. We leverage this API to experimentally check data-obliviousness of R function implementations.

We examined every performance metric that can be collected from PCM and chose metrics (listed in Figure 4) that are relevant for $\mu Arch$ side-channel detection. These metrics cover one or more of three criteria: cycle counts, cache hits/misses and bytes from/to the memory controller. These API functions all begin with prefix `get` and are followed by the metric they measure.

We illustrate an example using one of these measurements, cycle counts. In this simple experiment, we show how such small differences in instruction count from Figure 3 translate into measurable effects. We measure the number of cycles taken to evaluate one million iterations of expression `0 & 0` against those of `1 & 0`. Having access to a large number of measurements may occur naturally, *e.g.*, if the sensitive data is accessed in a loop, or if the attacker performs a $\mu Arch$ replay attack [65]. Note that the difference of execution length between two expressions is only two x86-64 instructions in Figure 3.

Figure 5 visualizes 100 trials of cycle count measurements against the aforementioned two sets of inputs in boxplots. The left box shows distribution of 100 measurements for each million iterations of expression `0 & 0` and the right box represents measurements for expression `1 & 0`. On average, it took $\mu_{00} = 73.9$ million cycles ($\sigma_{00} = 441$k) for `(0 & 0)`,

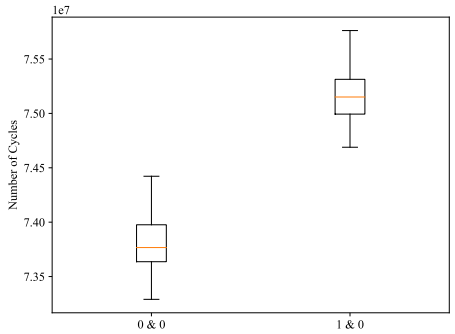


Fig. 5: Number of cycles taken to run one million iterations of 0 & 0 and 1 & 0. Each boxplot represents 100 measurements of each expression.

but it took $\mu_{10} = 75.2$ million cycles ($\sigma_{10} = 416$k) for `(1 & 0)` on average. The cycle count differences vary by a noticeable margin in the evaluation of these two expressions; even in the box plot, there is a clear separation between the experimental cycle counts of `0 & 0` and `1 & 0`.

E. Discussion

These examples are only a small subset of the parts of R that leak sensitive information. We discovered similar issues for other logical operators `|` and `xor()`, as well as functions like `sum()` found in its standard library. This, of course, does not preclude vulnerabilities arising from data-dependent R code. For example, an `if` statement with a sensitive predicate can reveal that predicate to the SGX adversary [1], [27] in R as well. Making matters worse are vulnerabilities due to timing side channels of just-in-time compilation [12], the timing differences of primitive C operations on floating point numbers [5] (such as `fdiv`, used throughout R), and the use of data-dependent `glibc` C library functions (e.g., `pow(y,x)` and `log(x)`).

R is a large code base comprising 992,564 lines of code, and is composed of hundreds of API functions and other features, implemented in a combination of R, C and Fortran [58].[2] Thus, all existing $\mu Arch$ side-channel attacks on C and Fortran applications must be considered when assessing security of R stack.

Side-channels in R present a serious security problem. Many data scientists and statisticians use R to compute on sensitive data every day. Clearly, it is not tractable for these users to understand the security implications of the code they write. At the same time, R's large code base makes manually patching data leaks inherently haphazard and error prone, even for security experts. As a result, experts have hitherto focused on replicating R's functionality in a new language/stack [63]. While these techniques add security, they trade-off expressiveness and usability by forcing data scientists to rewrite their code for a new programming stack.

In the next section, we address this challenge by designing the first secure R stack, where data scientists can program in

[2]Specifically, there are 388,141 lines of C, 345,547 lines of R and 258,876 lines of Fortran in the version of the R source we used for this paper.

(nearly) unchanged R, interact with the same R functionality with which they are familiar, and have strong confidence there are no latent side channels.

IV. The Data-Oblivious Virtual Environment

We now describe our solution to these problems, the Data-Oblivious Virtual Environment (DOVE). This begins with a discussion of our design principles and solution overview (Sections IV-A and IV-B). Section IV-C discusses the Data-Oblivious Transcript (DOT), which serves as the link between high-level programming and data-oblivious execution. Section IV-D discusses the DOVE frontend, which is a set of classes that convert R code into the DOT, using pseudonyms instead of sensitive data. Finally, Section IV-E describes the DOVE backend, an SGX enclave that converts the DOT operations on pseudonyms to data-oblivious computation on the actual sensitive data. For more details on the design and implementation of DOVE, please refer to our NDSS '21 paper [42].

A. Design Principles

To be a practical, yet secure, programming environment for outsourcing scientific computation, DOVE requires the following:

- *Correctness*. It is necessary to provide some evidence that computed values are correct, at least for a basic collection of computations. Importantly, R code run in DOVE must have the same output as R code run outside of DOVE.
- *Expressiveness*. It is important to demonstrate that it can code enough interesting cases to be worthwhile. DOVE should be able to handle enough R functionality to be a reasonable system for data science. Additionally, DOVE should not require any changes or modification for a user's library of data processing scripts. In other words, DOVE should be transparent to the user.
- *Data-Obliviousness*. Data-oblivious computation techniques defend computation from the SGX adversary described in Section II-C. DOVE should attempt to defend against all known $\mu Arch$ side-channels, such as the ones described in Section III, but be modular enough such that it can be easily patched in case a new class of side-channel is found.
- *Efficiency*. DOVE computations must sufficiently limit computational overhead. Some overhead is to be expected due to side-channel hardening and use of SGX, but it should not be so much as to prevent real data-science applications.

We developed this set of principles as a result of our experiments on R. We wanted to combine the expressiveness of R scripts with a data-oblivious core, while maintaining the efficiency required for data science (and, of course, the correctness). Our DOVE design aligns to these goals, and we evaluate our success in achieving them in Section V.

B. Overview

DOVE's security objective is to evaluate programs written in high-level (e.g., interpreted) languages in a data-oblivious manner (Section II-D). The key insight is that an operation that is truly data oblivious does not require the actual data to be present. Instead, the operation can take place on a *pseudonym* of the data. These pseudonyms have the same interface as normal data of the same type and support the same operations. For example, matrices are replaced with matrix pseudonyms, and matrix pseudonyms can be computed upon using the same operations as normal matrices (e.g., element-wise addition, matrix multiplication). However, the pseudonym contains no sensitive data, i.e., all of its data entries are replaced with $\perp$. This pseudonym is constructed solely through non-sensitive information specified for each pseudonym, such as, for matrices, the number of rows and columns. However, since the pseudonym does not actually have the data, any operation on the pseudonym is functionally equivalent to a NOP, i.e., $* \oplus \perp \rightarrow \perp$ where $*$ is a wildcard for any data value and $\oplus$ is an operation on the data. Instead, the operation performed is appended to a log. This log, which we call a *Data-Oblivious Transcript (DOT)*, is thus akin to a straight-line representation of the execution of the input program. The DOT can then be replayed on the *actual* data, executing the same operations as the input program.

With this in mind we propose the following architecture, shown in Figure 6. Our architecture is broken into two components, making up a *frontend* and *backend*. Each of N clients runs the same input — a common (non-sensitive) high-level program — in their local environment ("frontend"). The frontend replaces any references to sensitive data with pseudonyms and generates a DOT of the input program. Although only a single DOT needs to be generated for evaluation later on, each client can optionally compute its own DOT for program integrity-checking purposes (see Section IV-D for more information). This TEE ("backend") hosts the DOVE virtual machine, which is built with data-oblivious primitives. The virtual machine checks that all DOTs are equivalent (optional, for integrity) and runs the operations listed on the actual data.

Intuition for security comprises two parts. First, because the DOT is conceptually an execution trace, the backend TEE evaluates the same operations in the same order as the R program input to the frontend, regardless of the sensitive data provided to the backend. Importantly, the DOT was not created using any sensitive data, so the functions listed in the DOT are inherently independent/oblivious of that data. Second, we will architect the backend to ensure each operation is data oblivious, using well-established techniques for constant-time/data-oblivious execution.

The above architecture is general. The frontend can be adapted for different high-level languages (e.g., R, Python, Ruby), and the backend can be implemented for a variety of TEEs (e.g., SGX, TrustZone). For the rest of the paper, we explain, design, and evaluate ideas assuming the frontend input language is R and the TEE is SGX.

C. Data-Oblivious Transcript (DOT)

> *Relevant design principles*: expressiveness, data-obliviousness, efficiency.

The Data-Oblivious Transcript, or DOT, forms the core of the DOVE architecture, bridging an input program written in a

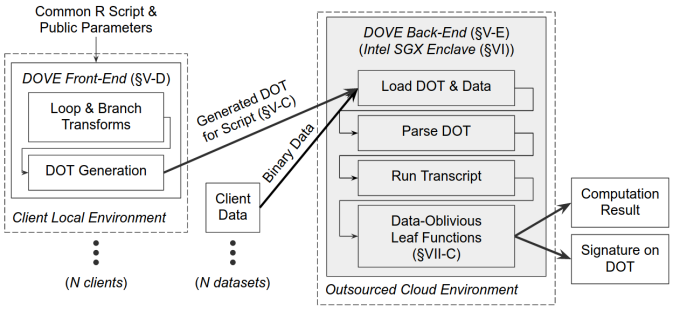


Fig. 6: High-level overview of DOVE. Bold-face arrows between nodes represent communication over (mutually-authenticated) TLS, while thinner ones are intra-process communication within a component. Shading indicates the location of our trusted computing base (TCB).

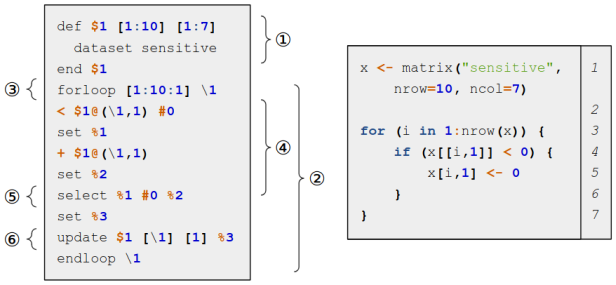


Fig. 7: A DOT (left) and its associated R program (right). The matrix `x` corresponds to the pseudonym `$1` in the DOT, and the loop index `i` with `\1`. ① corresponds to line 1 of the program, ② the `for` loop on line 3, ③ the `if` statement on line 4, and ⑥ the assignment in line 5. Intermediate values are stored in variables marked with `%`, and constants are declared using `#`.

high-level language with data-oblivious execution on a secure enclave. The DOT is designed to be built using only parameters related to the computation that are non-sensitive (such as data size). Because DOTs in DOVE are generated automatically, the client programmer does not need to learn the DOT language to write data-oblivious code. Once generated, the DOT is sent to the backend, where it is used to "replay" the same operations on the actual data (Section IV-E).

What to include in the DOT semantics strongly influences the TCB size in the backend and DOVE's overall performance. The structure of the DOT is similar to straight-line code where every operation is evaluated in the order it appears. Conditionals, data-dependent loops, etc. must be emulated with predicated, bounded execution as described below. Then, what primitive operations to include in the DOT semantics becomes a security/performance trade-off, because the cost to parse and run each operation in the DOT incurs non-negligible overhead in our current implementation (Section V-C). For example, DOVE might implement a transcendental function such as `sin` as a single primitive operation in the DOT or as a sequence of simpler operations in the DOT (such as bitwise operations). The former design is higher performance but requires a larger TCB: the backend parses a single DOT operation and evaluates that operation using a dedicated data-oblivious implementation of `sin` in the target Instruction Set Architecture (ISA), e.g., x86-64. The latter has the opposite characteristics: the backend parses each bitwise operation yet only needs dedicated support to implement data-oblivious bitwise operations. In these situations, we decide what operations to include in the DOT semantics on a case-by-case basis, described below and in Section IV-D.

We now discuss DOT semantics in more detail, using Figure 7 as a running example. We break the discussion into two parts, first describing data creation and operations on said data, and second describing (data-oblivious) control flow. A formal EBNF grammar for the DOT can be found in the paper's extended version [41].

1) Data creation, types and operations: We first discuss variable declarations, types and primitive operations.

a) Data types: When the frontend transcribes a program into a DOT, the DOT grammar only allows program inputs to be (1) fixed, concrete values or (2) pseudonyms. The two basic types of pseudonyms are matrices and scalars, with matrices being composed of $m \times n$ scalar (i.e., numeric) elements. Each operation on a matrix is usually decomposed into an operation on (1) its rows, (2) its columns or (3) its elements. Thus, in the case where matrix dimensions are non-sensitive, the sequence of operations needed to compute on actual matrix data is fully captured in the DOT.

b) Operations on data: Core functions comprise the set of primitive operations available to the DOT, including mathematical and logical operators (e.g. `+`, `==`), common mathematical functions (e.g. `exp`, `sin`), and summary operations (e.g. `sum`, `prod`).

There are two flavors of operations supported in the DOT, shown in first two rows of Figure 8. The Safe DOT/Core category contains operations deemed safe to operate on sensitive data in the backend. Every operation in this set must be implemented data-obliviously by a compliant backend, i.e., its evaluation must result in operand-independent resource usage on the target microarchitecture (see Section II-D). Each operation in this set has the following type signature: *if at least one operand is a pseudonym, the result is a pseudonym.* This is similar to taint algebras in information flow [60], [69] where if one operand is tainted, the result is tainted.

The Unsafe DOT/Core category contains operations which the DOT deems not safe to operate on sensitive data. For example, the `forloop` construct. These operations are only allowed to take non-pseudonyms as operands. Importantly, the selection which operations are marked Unsafe is a design choice. An alternate set of DOVE semantics can specify a Safe variant of any Unsafe operation, subject to the constraint that the backend must support a data-oblivious implementation of said Safe operation.

To summarize, we have:

- *Rule 1:* If an operation's operand(s) are pseudonyms, the result is a pseudonym.
- *Rule 2:* Safe operations may take pseudonyms or non-pseudonyms as inputs. Safe operations must be implemented data obliviously by the DOVE backend.
- *Rule 3:* Unsafe operations may only take non-pseudonyms as inputs.

This is analogous to the Data-Oblivious ISA policy *Confidential data↛Unsafe instruction*, which is analogous to the classic policy *High↛Low* in information flow. If a DOT follows the above rules, we call it a *valid DOT*. Whether a DOT is valid is checked before the DOT is evaluated by the backend (Sections IV-E), and invalid DOTs are disallowed.

2) Control flow: For reasons discussed above, the DOT disallows traditional control-flow constructs such as `if`, `while`, and `goto`, but supports predicated execution and bounded-iteration loops (similar to the program counter model [51]).

a) Bounded iteration: The DOT provides a `forloop` iteration primitive that only allows non-sensitive/non-pseudonym predicates. This primitive further does not support infinite loops. Loop indices are declared as non-pseudonyms. We note that supporting `forloop` is purely a performance/DOT size optimization. Equivalently, the loop could have been unrolled and the `forloop` construct removed.

b) Predicated conditionals: The DOT supports a `select` primitive that takes a pseudonym-typed predicate and returns one of two pseudonym operands based on the value of the predicate. `select` supports both scalar (i.e., logical `0` and `1`) and matrix predicates. Matrix predicates are transformed into element-wise select operations between the predicate and result/operand matrices. Thus, the predicate and its operands must have the same dimensions.

D. Frontend

Relevant design principles: correctness, expressiveness, efficiency.

The frontend takes R program with non-sensitive parameters as input and outputs a DOT. We develop our prototype frontend for R, but stress that the structure of the DOT is language-agnostic. As in a traditional compiler stack, one could design a different frontend for a different language that likewise compiles into the DOT representation.

Before initialization, clients share non-sensitive information, such as names and dimensions of datasets, with each other. The data within each dataset is considered sensitive and is not shared. To create a DOT, a client sources the DOVE frontend, which loads the names and dimensions for each sensitive input and creates a pseudonym for each in the R environment. The client then runs their program, performing operations as normal. Instrumentation in the R interpreter (see below) records each operation into the DOT, translating each dataset to primitives supported by the DOT semantics (e.g., scalar and matrix types). Clients can access elements, assign new values, apply operators, and run functions, all while dealing only with pseudonyms. Because the frontend does not have the actual data, this transcription is sensitive data-oblivious by design.

Our DOVE implementation ensures interface compatibility with base R in the implemented functions of the frontend. We use R's S3 method dispatch to overload functions in base R for pseudonyms. This requires no modification to the R interpreter, as clients merely have to import the DOVE frontend in their existing programs; in most cases, no programmer intervention is necessary.

Figure 8 lists all functions available to programmers. The Safe and Unsafe "DOT/Core" group of functions are those included in the DOT semantics (see previous section). To provide a richer library for clients, we also provide a "Supplemental" group of functions which are built using only the operations in "DOT/Core". For example, `colSums` calls the DOT function `sum` in a loop over the columns of a matrix. We provide these functions to enhance the user programming experience and to show that our DOT functions are sufficient primitives to develop more complex functions. Note that the "Supplemental" functions do not add to size of the TCB. They do not require changes to DOT semantics and therefore do not change the backend implementation.

1) Construct-specific handling: We now describe how the frontend translates different R programming constructs to the DOT semantics from Section IV-C.

a) Bounded iteration: Native R's `for` loop is not DOT-aware, so it just repeats the body of the loop m times. Instead, the frontend automatically transforms such bounded loops to use the `forloop` DOT construct. In our testing, we observed a $> 99\%$ decrease in frontend runtime using the DOT's `forloop` loops over normal `for` loops for compute-heavy $O(m^2)$-complexity programs. Early loop termination (e.g., `break`) is transformed in a manner similar to those of prior works [13], [44].

b) Predicated conditionals: The frontend must translate conventional if-then-else structures into the predicated execution model supported by the DOT (Section IV-C). For this, we implement an if-conversion transformation that is similar to prior works [20], [59]: an if-else with a sensitive predicate is converted into straight-line code where both sides of the if-else are unconditionally evaluated and a DOT `select` operator is used to choose the correct results at the end. Our frontend automatically converts R `if` statements to use the `select` primitive (discussed in Section IV-C) in the DOT. The whole expression is then recorded into the DOT directly; since the frontend does not have access to the actual data, the DOT must necessarily record both sides of the condition.

c) Disallowed constructs: Overall, the frontend's job is to translate R semantics into DOT semantics. Sometimes this is not possible, in which case the frontend signals an error. We explain two such cases (which are also common issues in related work). First, the frontend does not allow loops where the predicate depends on a pseudonym. Second, the frontend does not allow running operations with unimplemented types e.g., string-based computation or symbol-based computation. For example, one genomic evaluation program named `geno_to_allelecnt` in Section III-A receives a matrix of characters as a sensitive input. This program calls string operations like substring search or string concatenation.

Importantly, mentioned before, the frontend may contain a bug that results in an invalid DOT that contains an illegal construct such as those mentioned above. Such non-compliant DOTs are checked at parse time in the backend and rejected before being run.

Fig. 8: DOVE functions/operations. Functions in group "DOT/Core" are implemented directly in the DOVE backend and are included in the DOT semantics. Functions in the group "Supplemental" are implemented using operations in "DOT/Core" and exposed to the user as library functions. Safe functions require a data-oblivious implementation in the backend as they may receive pseudonyms as operands. Unsafe functions do not require a data-oblivious implementation, but can only take non-pseudonyms (non-sensitive) data as operands.

Group	Functions						
Safe DOT/Core (in TCB)	`abs`	`sqrt`	`floor`	`ceiling`	`exp`	`log`	`cos`
	`sin`	`tan`	`sign`	`+`	`-`	`*`	`/`
	`^`	`%%`	`%/%`	`>`	`<`	`>=`	`<=`
	`==`	`!=`	`|`	`&`	`!`	`all`	`any`
	`sum`	`prod`	`min`	`max`	`range`	`is.na`	`is.nan`
	`is.infinite`	`select`	`%*%`	`cbind`	`rbind`		
Unsafe DOT/Core (in TCB)	`forloop`	`dim`	`[`	`[[`			
Supplemental (not in TCB)	`fisher.test`	`pchisq`	`mean`	`colMeans`	`colSums`	`rowMeans`	`rowSums`
	`is.finite`	`as.numeric`	`as.matrix`	`apply`	`lapply`	`unlist`	`which`
	`data.frame`	`matrix`	`split`	`pmin`	`pmax`	`nrow`	`ncol`
	`len`	`t`					

E. Backend

Relevant design principles: correctness, data-obliviousness, efficiency.

The backend is a trusted SGX enclave (optionally, with attestation support) that runs the DOVE virtual machine that parses the DOT and runs the instructions contained within on the clients' sensitive data. Code in the backend ensures that only valid DOTs are run (Section IV-C), and includes implementations of all operations in the DOT semantics, i.e, those listed under Safe and Unsafe "DOT/Core" in Figure 8. Each client securely uploads (e.g., over TLS) the DOT of their R program. All clients additionally upload their shares of the sensitive dataset to the backend as well, in preparation for processing, as shown in Figure 6.

The scope of DOVE is to block all non-speculative $\mu Arch$ side channels (Section II-C2). For this purpose, the backend provides a data-oblivious implementation for operations in Safe "DOT/Core" of Figure 8. To implement these operations, we rely on a subset of the x86-64 ISA and well-established coding practices [20] for implementing constant-time/data-oblivious functions (see Section V-B for details). For example, we implement the `select` operation using the x86-64 `cmov` instruction, and all floating-point arithmetic functions are implemented using libfixedtimefixedpoint (libFTFP), a constant-time fixed-point arithmetic library created as a work-around for timing issues on floating-point hardware [5].

Importantly, what hardware operations (e.g., machine instructions) open $\mu Arch$ side channels depends on the $\mu Arch$. For example, two x86-64 processors can implement `cmov` differently: one in a safe way, one in an unsafe way (e.g., by microcoding the `cmov` into a branch plus a move [81]). DOVE is robust to new leakages found in specific $\mu Arch$ because to block a newly discovered leakage, it is sufficient to make a backend change. For example, if a vulnerability is found in `cmov`, the backend can opt to implement the DOT `select` operation using a CSWAP (bitwise operations) or other constructs.

V. Experimental Evaluation

We evaluate DOVE by designing experiments to validate its four design principles of correctness, expressiveness, data-obliviousness, and efficiency. We aim to use R as a baseline, comparing its results to those of DOVE. In this way, we validate DOVE using R as a reference. Our evaluations were performed on a machine with an Intel Skylake Core i3-6100 CPU, 1 TB HDD, and 24 GB of RAM, of which 19.37 GB was allocated to the SGX enclave. The machine was running Ubuntu 18.04.4 LTS and SGX software version 2.9.1 with EPC paging support. Thus DOVE's memory is not limited to EPC size, but this mechanism adds performance overhead when it is required. The frontend ran under R interpreter version 3.4.4, and the backend was compiled against `g++`, toolchain version `7.5.0-3ubuntu1~18.04`.

A. Correctness and Expressiveness

We combine our experiments to verify the correctness and expressiveness properties of DOVE. We first perform unit tests to check that individual R functions have correct output. We then proceed to use examples to show that DOVE can express solutions to real-world problems, and does so correctly.

1) Unit tests: For correctness, we confirm that what we get from DOVE is the same as what we would get from R. We perform simple unit tests. First, we ensure that frontend generate the correct DOTs, transliterating R functions into the appropriate DOT primitives. Then, we verify that the DOT primitives are processed correctly on the backend and output the expected result. Finally, we validate the end-to-end functionality of the system, checking that R output and DOVE (frontend-DOT-backend) output are equivalent.

2) PageRank: We begin with an introductory case study on the PageRank algorithm that is used as a case study on a custom data-oblivious programming language [63]. A large proportion of this algorithm is composed of matrix multiplications, which other works choose as primary performance benchmarks [43], [59]. Our DOVE implementation of this algorithm is found in Figure 9. Note that Line 4 is syntactic sugar to generate a random matrix in the backend, without putting those values in the DOT.

3) Evaluation scripts: We demonstrate that we can conveniently (and accurately) create DOTs from R code for our evaluation programs, as described in Section III-A. Using DOVE, we were able to transform (in the frontend) and run (in the backend) 11 out of the 13 evaluation programs, totaling 326

Fig. 9: The DOVE-compatible implementation of PageRank in R.

```
page_rank <- function(M) {
    d <- 0.8
    N <- nrow(M)
    v <- matrix(nrow = nodes, ncol = 1, rand =
      TRUE)
    norm_one <- sum(abs(v))
    v <- v / norm_one
    M_hat <- (M * d) + ((1-d) / N)
    iters <- 40
    for(i in 1:iters) {
        v[,] <- M_hat %*% v
    }
    v
}
```

lines of R code. The first program that we could not implement, `geno_to_allelecnt`, works on character data instead of numeric data, and as such is not supported by the current types available in the DOT. The second program, `gwas_lm`, performs a Genome-Wide Association Study (GWAS) using support in R for linear models. We were not readily able to implement this; R provides parameters to models as a formula of symbols, not values. DOVE currently does not support this paradigm, but we believe that DOVE can be extended to do so in the future.

Ten of the remaining 11 evaluation programs were automatically transformed by the frontend into data-oblivious code. Only one program, `LD`, required manual intervention, as it was written entirely in a data-dependent style. For this program we: (1) replaced some functions that are intrinsically data-dependent with data-oblivious primitives and (2) changed lines that required sensitive data-dependent array indexing with worst-case array scans. Future implementations could alternatively use an oblivious memory, e.g., [62], to avoid such worst-case work.

B. Data-Obliviousness

It is critical that the backend is secure against $\mu Arch$ side channels. Our backend is implemented in a data-oblivious style, only using constructs that are known to the side-channel free on the x86-64 ISA. To avoid side channels that can arise due to floating point numbers, we use a previously-evaluated fixed-point library, libFTFP [5], designed to provide computation on decimal numbers in constant time.

In our backend architecture, the only place we perform computation on sensitive data is in what we term *leaf functions*. These functions are at the "leaf" of our call tree, and implements a specific DOT operation on data. Up to that point in the call tree, our backend only operates on the DOT, performing instruction fetch and setting up pointers to data for the leaf functions. Only leaf functions dereference these pointers, and then read and modify sensitive data. Verifying data-obliviousness of these functions is a crucial assessment of DOVE's security promises. For more information on our backend architecture, please refer to our NDSS'21 paper [42].

Based on the above discussion, we now scrutinize whether these leaf functions enable our security guarantee, i.e., uphold Rule 2 from Section IV-C. We use the same set of experiments we used in Section III to verify the data-obliviousness of DOVE. For this, we manually disassemble and analyze every binary object file associated with DOVE functions, and verify that the subset of instructions which operate on sensitive data are instructions that do not create $\mu Arch$ side channels as a function of their operands. We also inspect the PCM characteristics of DOVE to identify any missed side channels.

Fig. 10: Intel-syntax assembly for `BitwiseAndOp::call`, the DOVE backend equivalent of `&` in R. This snippet has been lightly edited for clarity.

```
; [snip] push current register state
; initialize registers
mov     r13,rcx
mov     r12,rdi
mov     rbp,rdx
mov     rdi,rsi
mov     rbx,rsi
; convert fixed point number to int
call    fixed_to_int(fixed)
mov     rdi,rbp
mov     r14d,eax
call    fixed_to_int(fixed)
and     eax,r14d ; the actual operation
mov     rsi,r13
movsx   edi,al
; place `and` result into fixed
call    place_bool_in_fixed(int8_t, fixed*)
mov     rcx,r13
mov     rdx,rbp
mov     rsi,rbx
mov     rdi,r12
; [snip] pop previous register state
; supercall to data-obliviously check for NAs
call    BinaryOp::call(fixed, fixed, fixed*)
```

1) Static opcode analysis: We disassembled the object files generated during compilation and manually looked at every function that performed an operation on sensitive data. The machine instructions that run in these functions are of relevance to the security of DOVE, since insecure instructions may leak information about the data. A slightly truncated example of this disassembly can be found in Figure 10 for the backend's `&` operator used in our previous examples (e.g., Figure 1). Note the lack of branches on conditional data, as compared to the disassembly in Figure 2b. Compilation on different platforms can provide different results, so this analysis may have to be reapplied.

We first analyze the leaf function instructions that take sensitive data as operands. These instructions are shown in Figure 11. We determined this set by inspecting instruction dependencies in the `objdump` disassembly. All but one of the opcodes in Figure 11 is considered to be a data-oblivious instruction by libFTFP, our constant-time fixed-point arithmetic

Fig. 11: All x86-64 opcodes that operate on sensitive data in the leaf functions of DOVE. Those marked with * are those not found in libFTFP.

add	and	cdqe	cmovne*	cmp	imul
lea	mov	movabs	movsd	movsx	movsxd
movzx	mul	neg	not	or	pop
push	sar	sbb	seta	setae	setbe
sete	setg	setl	setle	setne	shl
shr	sub	test	xor		

library. We refer to its authors' analysis for its security [5]. The one instruction not found in libFTFP, `cmovne`, is used for conditional moves of sensitive data in the backend. This instruction is likewise shown to be data oblivious in [59]. We further verify that the above instructions use the direct register addressing memory mode for each operand, if the value stored in the register for that operand is sensitive (which also follows standard practice for writing data-oblivious code).[3] Thus, we conclude that the machine instructions operating on sensitive data in the backend do not create $\mu Arch$ side channels.

Beyond the instructions in Figure 11, there are other instructions in the leaf functions that *do not* operate on sensitive data. Examples include jumps to implement loops with non-sensitive iteration counts, checks to validate dimensions on operations, sanity checks for **`nullptr`**, and instructions associated with implementing polymorphism. Some of these are not data oblivious (e.g., jumps), but do not impact security because they operate on non-sensitive data such as matrix dimensions.

2) Dynamic execution analysis: To further corroborate our static security analysis, we also looked at runtime instruction statistics, as we did for R in Section III-C2. We used the branch-trace-store execution trace recording [37] of the DOVE backend execution, varying the input data. We found that the sequence of non-speculative dynamic instructions executed was independent of the data passed to the backend: that is, the backend satisfies the PC model [51]. Security follows from these two analyses: (a) that the backend follows the PC model and (b) that each individual instruction that operates on sensitive data consumes operand-independent hardware resource usage (previous paragraphs).

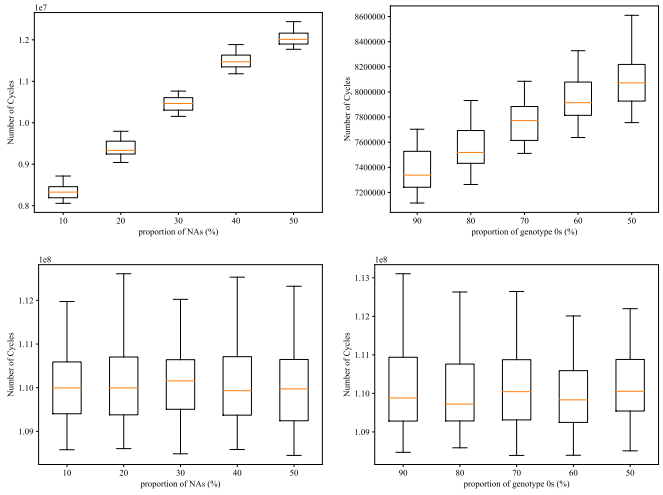


Fig. 12: Cycle count measurements for runtime Intel PCM analysis against Line 1 of Figure 1. Plots above are measurements from vanilla R and plots below are from DOVE. The plots on the left are tested against varying proportions of **`NA`**, and plots on the right are tested against varying proportions of **`0`**.

[3]x86-64 operands can utilize one of several flavors. For example, `rax` denotes a register file read and `[rax]` denotes a memory de-reference. The former is considered safe for use in constant-time/data-oblivious programming, while the latter creates memory-based side channels.

3) Intel PCM: We additionally validate DOVE's data-obliviousness by repeating the PCM experiments we conducted on the R interpreter in Section III-D. We performed PCM tests on every function in the backend that correspond to the Safe DOT/Core groups in Figure 8 on 14 metrics that previously described.

For these tests we generate and compare synthetic matrices with different data distributions, i.e., pairs of D and D'. As we have seen from our instruction analyses, base R implementations handle corner cases for missing data (**`NA`**) and/or **`0`** (e.g., `log()`, `exp()`), such that varying the proportions of these two values in the input matrix results in noticeable differences in execution at the $\mu Arch$ level. Thus, we test our functions with varying amounts of these two values. The first set tests whether the function is data-oblivious against **`NA`** or not. This set consists of matrices with five different proportions (10%, 20%, 30%, 40%, 50%) of **`NA`**. A second set tests whether the function is data-oblivious against **`0`** or not. This set also consists of matrices with five different proportions (90%, 80%, 70%, 60%, 50%) of **`0`**. We generate 100 matrices of each proportion randomly in both sets for our testing. The size of each matrix is 1,000 by 60.

Figure 12 shows boxplots that illustrate 100 trials of cycle count measurements against the aforementioned two sets of inputs against Line 1 of Figure 1. Each box in the figure represents 100 measurements of random input set with varying proportions of either **`NA`** or **`0`**. When Figure 1 was run on vanilla R, the cycle counts differ drastically when the input's proportion of **`NA`** (top left) or **`0`** (top right) is varied. Both plots at the top shows a linear increase in cycle counts as the proportion changes, but measurements from DOVE do not show such a trend against **`NA`** (bottom left) or **`0`** (bottom right).

C. Efficiency

We define efficiency in terms of performance, which we measure primarily through the execution time of DOVE. This is the most important metric in defining the practicality and scalability of a solution like DOVE in a data-science context.

One run of our performance benchmark is as follows. We first record the runtime of vanilla (insecure) R with data and a program. Then, we run the DOVE frontend on the same program, generating the DOT and writing it to disk. We then initialize the backend, read in the DOT, parse it, and execute the DOT instructions. Our evaluation of the DOVE implementation discusses two measures. First, we wish to consider if our frontend primitives are sufficient to express complex programs. Second, we examine the performance of DOVE when compared to its base R counterpart.

To highlight the overheads inherent to SGX and libFTFP, the external data-oblivious fixed point library [5], we ran performance benchmarks on three configurations of DOVE: (1) backend outside an SGX enclave and without libFTFP, (2) backend outside an SGX enclave and with libFTFP, and (3) backend inside an SGX enclave and with libFTFP (our default configuration). SGX-related overheads include SGX's memory encryption and access protections that isolate the enclave from the rest of the machine [22]. In particular, EPC paging (discussed in Section II-C) is a significant overhead, especially for large datasets. These overheads were exacerbated

Fig. 13: Absolute runtimes and sizes of the evaluation programs. Programs marked with an * were run on a reduced dataset due to test system limitations. Program `iES` calls `EHHS`, so we include the lines of code from `EHHS` when measuring lines of code for `iES`. FE are measurements for frontend, NEBE are for measurements with backend without SGX, and EBE are for the backend with SGX. F indicates the use of libFTFP, the data-oblivious floating point arithmetic library that we used on our DOVE implementation. LoC stands for Lines of Code for the original R program whereas DOT size represents the size of the counterpart DOT file in bytes. Finally, the DOT overhead represents the relative overhead of the DOT's file size relative to the size of the original R program.

Program	Vanilla R (s)	FE (s)	NEBE (s)	NEBE w/ F (s)	EBE w/ F (s)	LoC (lines)	DOT size (bytes)	DOT Overhead
`EHHS`*	18.9	3.85	1104.43	2131.65	3575.46	40	1538	0.51
`iES`*	23.48	6.43	1106.34	2161.95	3625	15 + 40	159853	105.44
`LD`*	1787.58	3.64	2869.48	9040	32264	54	5610	0.98
`allele_sharing`	283.41	5.6	650.03	1841.28	29733	12	419	0.28
`hwe_chisq`	38.48	4.56	113.98	262.23	853.49	21	5295	4.35
`hwe_fisher`	690.2	4.98	141425	154194	234054	12	10287	3.92
`neiFis_multispop`	85.85	16.88	111.82	278.42	1077.44	38	5311	4.09
`neiFis_onepop`	39.13	4.9	55.85	192.53	764.38	19	7381	2.43
`snp_stats`	692.73	11.21	142783	155840	236644	33	1980	1.35
`wcFstats`	55.27	8.21	79.38	186.27	757.38	35	6624	1.58
`wcFst_spop_pairs`	74.05	15.43	206.55	458.26	1343.51	45	18606	5.21

by increases in the working set of the enclave application. The libFTFP instructions' relative performance overhead is measured against its Streaming SIMD Extensions (SSE) counterpart; the overhead varies depending on the instruction, ranging from 1.2× for `neg` (operand negation) to 208× for `exp` (exponential function evaluation) [5].

We utilize the dataset from the honeybee study [6] to perform performance benchmarking. We run the full 2,808,570 × 60 (≈ 1.3 GB) dataset for all programs with space complexity of $O(m * n)$ where m is the number of rows and n is the number of columns. However, some of the evaluation programs could not run on this dataset due to machine limitations. Specifically, some programs with space complexity of $O(m^2)$ refuse to run even in vanilla R at full size. To address these limitations, we run a subset of programs with the first 10,000 rows of the honeybee dataset. Some related work also runs performance benchmarks on genomic data with similar sizes to that of our reduced dataset [18], [19], [61].

To normalize benchmark results run on datasets of different sizes, we present a relative overhead metric: runtime for DOVE (DOT generation, disk reading/writing, DOT evaluation) divided by runtime in vanilla R. This relative overhead metric is shown as stacked bar graphs in Figure 14, while raw numbers can be found in Figure 13 Each part of the bar represents the overhead contributed by a component of the backend, categorized by three factors: the DOVE runtime's data-oblivious implementation itself, constant-time fixed point operations (libFTFP), and the use of the SGX enclave. Overall, each factor provides additional security at the cost of increased overhead. We separate our programs into two bins: programs that run on the full honeybee dataset, and programs that run on a reduced dataset due to machine limitations (marked with * across the subfigures).

The min/avg/max size overhead of each DOT relative to its R script is 0.284x/10.8x/105x. Note, the DOT may be smaller than the original program because of the DOT instruction set. We expect that the DOT can be significantly compressed. Case in point, the current DOT is represented in ASCII which is space inefficient.

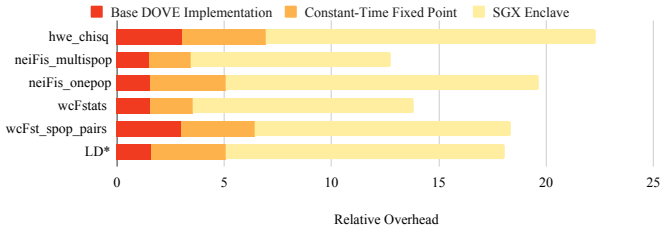
We now provide more detailed analysis for several programs with noteworthy performance characteristics.

1) Programs with quadratic space complexity: The relative overhead with DOVE is 120.7× against vanilla R on average for programs `EHHS`, `iES`, and `LD`. These three programs run statistics based on pairwise SNPs, i.e., a row is compared to each other row in the dataset. They operate in $O(m^2)$ space, or, quadratic in the number of rows m. The large relative overhead in the base DOVE implementation for `iES` and `EHHS` is due to data-oblivious transformations. Namely, the vanilla R versions of these programs benefit from early `breaks` in the loop body that occur depending on sensitive values. DOVE does not directly allow such behavior for security reasons. Hence, the backend must iterate through the entire matrix, regardless of the data, causing potentially high overhead.

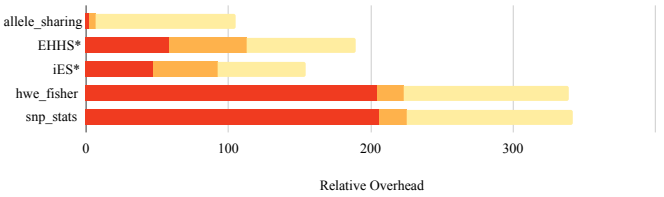
2) Statistical programs: The programs `hwe_chisq` and `hwe_fisher` each call a base R statistics function: `pchisq` (Chi-Square distribution) and `fisher.test` (Fisher's exact test), respectively. The program `snp_stats` calls both functions. In base R, the implementation of `fisher.test` is written in R itself whereas `pchisq` is written in C. We implement both as supplemental group functions in R (Figure 8), to provide a fair comparison and to reduce TCB size. When called, the frontend will convert the call into a series of equivalent DOT operations.

We note that, to achieve data obliviousness, our implementations of these functions are somewhat different than their vanilla R counterparts. For instance, computing a factorial of a sensitive value is intrinsically data dependent, but it is required to compute Fisher's exact test (in R, `fisher.test`). To implement factorial data obliviously, we implement it as an oblivious table lookup over a pre-determined domain of inputs, noting that other data-oblivious implementations are possible.

While `hwe_chisq` has reasonable performance overhead given our data-oblivious implementation of `pchisq`, both `hwe_fisher` and `snp_stats` show large performance overheads. These programs call the `fisher.test` function $O(m)$ times. The insecure version of this function takes $O(n)$ time. Our data-oblivious implementation takes $O(n^2)$ time due to inefficient oblivious-memory reads. As mentioned before, a more efficient oblivious-memory primitive would reduce overhead.



(a) Programs with less than 25× relative overhead.



(b) Programs with greater than 25× relative overhead.

Fig. 14: Performance evaluation results for the evaluation programs. Each stacked bar represents a measurement for each program. Each stack represents relative overhead of DOVE against vanilla R caused by generic data-oblivious computation, libFTFP and SGX from left to right. Programs marked with * run on reduced dataset due to machine limitations.

3) Remaining programs with linear space complexity: The remaining programs do not incur a significant performance penalty, as both the insecure and data-oblivious codes run in $O(m)$ time. The average overhead with DOVE is 28.3× relative to vanilla R for these programs. One program, `allele_sharing` (in Figure 14b), has a notably larger performance overhead than others when running inside the SGX enclave. We believe this is due to EPC paging costs. Specifically, this program has a larger working set size than SGX has EPC/PRM (2 GB vs. 64-128 MB). It further makes column-major traversals for a matrix that is stored in row-major order in memory, which leads to low spatial locality and therefore, we hypothesize, a high EPC fault rate.

VI. Lessons Learned

We now discuss some of the lessons we learning during the creation of DOVE. Our intermediate results helped us refine our design in three areas: expressiveness, data-obliviousness, and efficiency. We hope these observations will be useful to the community.

1) Automating expressiveness: An early version of DOVE was implemented as an R library instead of directly into R's base functions. This required end-users to rewrite their code base using DOVE functions as provided by the library in order to generate a DOT. In retrospect, this was not a good design, as it restricts expressiveness to only those functions the user knows how to use with DOVE. Under this design, a user might as well learn a different, data-oblivious language. We knew that we wanted to somehow automate the transformation of R scripts into DOTs in the frontend. This is where the evaluation programs [15] were useful. These scripts contain usage of many different R constructs: S3 base functions, statistical routines, and control flow structures, among others. We used the evaluation programs as a benchmark to ensure that our automatic conversion of scripts to DOTs was expressive enough, leading to the frontend architecture in DOVE today. The DOVE frontend transparently rewrites the S-expressions for a parsed R script (Section II-A) to use data-oblivious primitives. The end-user, in our implementation now, does not need to manually write DOVE-compliant R code, and instead can just run scripts out-of-the-box, thanks to the guidance provided by having a real-world evaluation set.

2) Data-oblivious statistics: R has a rich set of statistical functions baked into its standard library. To build some of this functionality into DOVE, we pulled off-the-shelf open source implementations and placed it into our backend as an external library. One of this functions is `fisher.test`, which calculates Fisher's exact test of statistical significance. Our evaluation programs [15] use this function to calculate deviation of input data from the Hardy-Weinberg Equilibrium. We did not consider the security implications of having an external library, nor did we fully understand the implementation of `fisher.test`. However, after applying data-oblivious tests to it, we noticed that it failed our instruction tests for branches on sensitive data. This is due to `fisher.test`'s use of factorials, which are data-dependent. We decided that the best way of implementing `fisher.test` was to rewrite it in data-oblivious R. The function calculates a large lookup table of factorials, and data-obliviously retrieves the correct value when needed. Thus, whenever a DOT running in the frontend calls `fisher.test`, it calls the function in the *frontend*, which itself is then transcribed into the DOT. This guarantees the security of the statistical function (and shrinks the TCB), but at a significant performance cost: 4.9× overhead for the insecure variant versus 315× for the (current) secure variant. This underscores the importance of inspecting the data-oblivious characteristics of the entire TCB.

3) Efficient looping: Our original DOT design did not have a `forloop` primitive. This meant that any loops used in input R code would be fully unrolled, its instructions copied into the DOT for each loop iteration. The DOT would become size $O(n)$ for loops of size n, which we initially thought was reasonable – ostensibly, data science systems have sufficient RAM to hold a large DOT in addition to the data upon which to operate. Our experiments showed that this was a flawed assumption. Evaluation programs that combined matrices together explicitly had reasonable performance in DOVE, while evaluation programs that explicitly looped through matrices had terrible performance. The size of the data from [6], combined with the overhead due to SGX EPC paging (Section V-C), slowed evaluation to a crawl. We realized that our backend had to minimize RAM in order to have reasonable performance, and we thus implemented it, reducing DOT complexity to $O(1)$ for loops of arbitrary size. We had to make additional efficiency jumps in order for our data-oblivious code to be performant, even if correctness and security properties were already guaranteed, in order to run real-world workloads.

VII. Conclusion

Our architecture and implementation, DOVE, provides strong security in the face of the strong adversary. This

adversary clearly can attack modern data science applications, as evinced by the myriad of side-channels experimentally visible in base R. The security of DOVE, on the other hand, is not at the cost of expressibility or of usability; we implement the vast majority of the mathematical functions and operators in base R, allowing us to transform and run non-trivial genomic programs on real data. Rigorous evaluation on our implementation shows our system is both truly data-oblivious and feasible in practice. In this way, DOVE represents the best of both worlds, spanning the gap between the security potential of data-oblivious programming and the usability of existing, insecure stacks. We hope that the experimental approaches that we used in this work will be useful for future work in the data-oblivious computation space.

REFERENCES

[1] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting Secret Keys via Branch Prediction. *IACR'06*.

[2] Adil Ahmad, KyungTae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A Data Oblivious Filesystem for Intel SGX. In *NDSS'18*.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. IACR'18.

[4] T Alves and D Felton. TrustZone: Integrated hardware and software security. *ARM white paper*, 2004.

[5] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *S&P'15*.

[6] Arian Avalos, Hailin Pan, Cai Li, Jenny P Acevedo-Gonzalez, Gloria Rendon, Christopher J Fields, Patrick J Brown, Tugrul Giray, Gene E Robinson, Matthew E Hudson, et al. A soft selective sweep during rapid evolution of gentle behaviour in an Africanized honeybee. *Nature communications*, 8(1):1550, 2017.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM TOCS'15*.

[8] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *PKC'06*.

[9] Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In *FSE'05*.

[10] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIA CCS'13*.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *CoRR'17*.

[12] T. Brennan, N. Rosner, and T. Bultan. JIT leaks: Inducing timing side channels through just-in-time compilation. In *S&P'20*.

[13] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. *SecDev'17*.

[14] Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, and Mona Vij. Chapter 8 - trusted execution environment with intel sgx. In Xiaoqian Jiang and Haixu Tang, editors, *Responsible Genomic Data Sharing*, pages 161 – 190. Academic Press, 2020.

[15] Eva KF Chan. Handy R functions for genetics research. https://github.com/ekfchan/evachan.org-Rscripts, 2019.

[16] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. *IACR'17*.

[17] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC'17*.

[18] Feng Chen, Chenghong Wang, Wenrui Dai, Xiaoqian Jiang, Noman Mohammed, Md Momin Al Aziz, Md Nazmus Sadat, Cenk Sahinalp, Kristin Lauter, and Shuang Wang. PRESAGE: privacy-preserving genetic testing via software guard extension. *BMC medical genomics*, 10(2):48, 2017.

[19] Feng Chen, Shuang Wang, Xiaoqian Jiang, Sijie Ding, Yao Lu, Jihoon Kim, S Cenk Sahinalp, Chisato Shimizu, Jane C Burns, Victoria J Wright, et al. Princess: Privacy-protecting rare disease international network collaboration via encryption through software guard extensions. *Bioinformatics*, 33(6):871–878, 2016.

[20] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P'09*.

[21] Intel Corporation. Processor Counter Monitor. https://github.com/opcm/pcm.

[22] Victor Costan and Srinivas Devadas. Intel SGX explained. IACR'16.

[23] David Darais, Chang Liu, Ian Sweet, and Michael Hicks. A language for probabilistically oblivious computation. *CoRR'17*.

[24] Jack Doerner, David Evans, and abhi shelat. Secure stable matching at scale. *IACR'16*.

[25] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR'17*.

[26] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS '16*.

[27] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS'18*.

[28] Ben A. Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using Intel SGX. In *CCS'17*.

[29] Xiaoyi Gao and Joshua Starmer. Human population structure detection via multilocus genotype clustering. *BMC genetics*, 8(1):34, 2007.

[30] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. IACR'16.

[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *EuroSec'17*.

[32] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security'18*.

[33] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *ICISC'09*.

[34] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *S&P'11*.

[35] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC'17*.

[36] Intel®. Intel® software guard extensions programming reference. 2014.

[37] Intel®. Intel® 64 and ia-32 architectures software developer's manual volume 3b: System programming guide. 2020.

[38] Intel®. Intel® software guard extensions sdk for linux os developer reference. 2020.

[39] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P'19*.

[40] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO'99*.

[41] Hyun Bin Lee, Tushar Jois, Christopher W. Fletcher, and Carl A. Gunter. DOVE: A Data-Oblivious Virtual Environment. In *Arxiv eprint*.

[42] Hyun Bin Lee, Tushar M. Jois, Christopher W. Fletcher, and Carl A. Gunter. DOVE: A Data-Oblivious Virtual Environment. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[43] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *SIGPLAN Not.*, 50(4):87–101, March 2015.

[44] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *S&P'15*.

[45] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *S&P'15*.

[46] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. 1959.

[47] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP'13*.

[48] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *S&P'18*.

[49] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *CoRR'17*.

[50] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. *CoRR'17*.

[51] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. *IACR'05*.

[52] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. Graphsc: Parallel secure computation made easy. In *S&P'15*.

[53] Alireza Nazari, Nader Sehatbakhsh, Monjur Alam, Alenka Zajic, and Milos Prvulovic. EDDIE: EM-Based Detection of Deviations in Program Execution. In *ISCA'17*.

[54] Masatoshi Nei. F-statistics and analysis of gene diversity in subdivided populations. *Annals of human genetics*, 41(2):225–233, 1977.

[55] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security'16*.

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA'06*.

[57] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security'16*.

[58] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2019.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security'15*.

[60] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[61] Md Nazmus Sadat, Md Momin Al Aziz, Noman Mohammed, Feng Chen, Shuang Wang, and Xiaoqian Jiang. SAFETY: Secure GWAS in federated environment through a hybrid solution with Intel SGX and homomorphic encryption. *arXiv preprint arXiv:1703.02577*, 2017.

[62] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. Zerotrace : Oblivious memory primitives from intel sgx. In *NDSS'18*.

[63] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *CCS'17*.

[64] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with sgx enclaves. In *NDSS'17*.

[65] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA'19*.

[66] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *S&P'15*.

[67] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *CCS'13*.

[68] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *CCS'17*.

[69] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS'04*.

[70] Kun Tang, Kevin R Thornton, and Mark Stoneking. A new approach for using genome scans to detect recent positive selection in the human genome. *PLoS biology*, 5(7):e171, 2007.

[71] Shruti Tople and Prateek Saxena. On the trade-offs in oblivious execution techniques. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer'17.

[72] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *SysTEX'17*.

[73] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS'17*.

[74] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. *IACR'14*.

[75] Bruce S Weir and C Clark Cockerham. Estimating F-statistics for the analysis of population structure. *evolution*, 38(6):1358–1370, 1984.

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P'15*.

[77] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE S&P*, 2019.

[78] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. A2: Analog Malicious Hardware. In *S&P'16*.

[79] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security'14*.

[80] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *IACR'16*.

[81] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS'19*. https://eprint.iacr.org/2018/808.

[82] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *S&P'13*.

[83] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *IACR'15*.

[84] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI'17*.