DOVE: A Data-Oblivious Virtual Environment

Hyun Bin Lee*, Tushar M. Jois†, Christopher W. Fletcher*, and Carl A. Gunter*
*University of Illinois at Urbana-Champaign:
{lee559, cwfletch, cgunter}@illinois.edu
†Johns Hopkins University: jois@cs.jhu.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. This can even be done with applications coded in high-level languages with complex programming stacks such as R, Python, and Ruby. However, this creates a trade-off between programming convenience versus the risk of attacks using microarchitectural side channels.**

In this paper, we argue that it is possible to address this problem for important applications by instrumenting a complex programming environment (like R) to produce a *Data-Oblivious Transcript (DOT)* that is explicitly designed to support computation that excludes side channels. Such a transcript is then evaluated on a Trusted Execution Environment (TEE) containing the sensitive data using a small trusted computing base called the *Data-Oblivious Virtual Environment (DOVE)*.

To motivate the problem, we demonstrate a number of subtle side-channel vulnerabilities in the R language. We then provide an illustrative design and implementation of DOVE for R, creating the first side-channel resistant R programming stack. We demonstrate that the two-phase architecture provided by DOT generation and DOVE evaluation can provide practical support for complex programming languages with usable performance and high security assurances against side channels.

I. Introduction

Recent commercially-available Trusted Execution Environments (TEEs) such as Intel SGX [28], [47] 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 cryptographic mechanisms.

The long-term vision pursued by TEE-based software systems (e.g., [10], [20]) 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 [17].

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23056
www.ndss-symposium.org

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., [14], [81], [90], [93]). Yet, existing software-based techniques to block these channels—coming from a rich line of research in data-oblivious/constant-time programming [11], [25], [66], [75]—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 [11], [12], require the use of custom domain-specific languages [16], [79], or only apply to close-to-metal compiled languages [66], [75]. 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 [74]. Subtle issues in any of this code create security holes.

The goal of this paper is to extend data-oblivious/constant-time techniques to apply to existing high-level, interpreted languages, thus enabling TEE-level security for non-experts. The key strategy and insight 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 for the data*. We call these stand-ins "pseudonyms".

To exploit this idea we perform computation 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 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.

Conceptually, the DOT plays a role similar to a compiler intermediate representation. Our approach can be characterized as a *frontend* translating high-level evaluation to the DOT and a *backend* evaluating the DOT data-obliviously, on sensitive data. A key benefit of this decoupled approach is that only the backend (importantly, *not* the frontend) is part of the TCB. This provides a powerful strategy for protecting complex, high-level programming stacks against side channel attacks. In addition to a reduced TCB, the decoupling provides modularity and extensibility benefits similar to those found in modern compilers. For example, to add support for a new high-level language, we need only change frontend code. Likewise if a security vulnerability is found in the TEE, or we wish to deploy different TEEs to protect execution for different processor microarchitectures, we need only change backend code.

Putting it all together, we design and implement an instance of the above architecture, called the "Data-Oblivious Virtual Environment (DOVE)". As a proof-of-concept, we develop a DOVE frontend that translates programs written in the R language to a DOT representation and design a backend that evaluates the DOT on sensitive data inside of an Intel SGX enclave.

To validate DOVE, we show how to support a third-party library of genomics analysis algorithms written in R [18], which we call the *evaluation programs*. Out of 13 evaluation programs, DOVE can run 11 of them, with these 11 totaling 326 lines of R code. For 10 of the 11 above programs, our frontend can automatically convert the unmodified R program into the DOT language; converting the remaining case required manual user intervention because of a programming construct not yet supported by our frontend. We collect performance benchmarks on these programs with a real-world genomic dataset consisting of three populations of honeybees [8].

Summary of contributions.

1) We identify a number of subtle side-channel vulnerabilities in the R language.
2) We design DOVE, the first architecture that runs existing high-level interpreted languages and is demonstrably resistant to side channels.
3) We provide an implementation of DOVE for R, creating the first side-channel resistant R programming stack.
4) We evaluate the security and performance of DOVE against evaluation programs drawn from the genomics literature. Relative runtime overheads of DOVE against vanilla R on these programs range from 12.74$\times$ to 341.62$\times$.

Source code for the DOVE frontend and backend prototype is available at https://github.com/dove-project.

Finally, the extended version of this paper [52] additionally includes 1) a grammar for the DOT language, 2) more details on the evaluation programs and 3) more details regarding the use of the Intel Performance Counter Monitor (PCM) APIs for our security evaluation.

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. Figure 1 is an R code example from our evaluation programs that includes such operations.

Computation in R. R is an interpreted language [74], and its interpreter is written mostly in C and to a lesser extent Fortran and R itself. Every object is represented with a symbolic expression (S-expression) [61] 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++).

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`.

S3 method dispatch. The most common object-oriented programming system in R is S3 method dispatch. For each function call on an object, the S3 object system calls the correct method associated with that object. For example, for `print(x)`, when `x` is a scalar, S3 calls `print.numeric(x)`; when `x` is a matrix, S3 calls `print.matrix(x)` instead. A programmer who wishes to add their custom type `myObject` to `print` would define a function `print.myObject(x)`. This paradigm makes it easy to supply new types of objects to existing functions, making the differences in implementation transparent to the end user. S3 is the OOP system used in the base R, making it especially useful to override commonly used functions.

B. Microarchitectural Side-Channel Attacks

Microarchitectural (shortened as "$\mu Arch$") side-channel attacks are a class of privacy-related vulnerabilities where a sensitive program's hardware resource usage leaks sensitive information to an adversary co-located to the same (or a nearby) physical machine [38]. Over the years, numerous hardware structures—a variety of cache architectures [71], [94], [96], [97], branch predictors [1], [34], pipeline components [3], [5], [43] and other structures [33], [42], [64], [73], [90], [93]—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 [60], [71] or Drama and the DRAM row buffer [73]), as opposed to sharing virtual memory with the victim (e.g. [96]).

C. Enclave Execution and Intel SGX

Enclave execution [84], such as with Intel SGX [47], 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 [14], [41], [45], [65], [70], [75], [78], [81], [90], [98], 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 [10], [20], containers [80], and data structure abstractions [78] 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 [62]. The EPC has a fixed size of 64 or 128 MB, shared among all enclaves [49]. 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 [47], [62].

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 [93] and follow-on work [90] 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 [14], [41], [44], [45], [65], [81], [88].

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], [11]–[13], [16], [19], [25], [30]–[32], [36], [59], [59], [63], [66], [67], [70], [75], [78], [79], [82], [83], [87], [91], [99]–[101]. 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], [32], [36], [63], [70], [75], [78], [79], [101] 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. Threat Model

In this paper we consider a setting where one or more users submit data to an untrusted server that computes on said data in a high-level language such as R. The server hosts SGX as well as a regular software stack outside of SGX. The user(s) and SGX hardware mechanism are trusted. The program computing on user data, like the R interpreter and evaluation programs, is assumed to be non-sensitive. No software running on the remote host outside of an SGX enclave is trusted—this includes the supervisor software stack, disks, the connection between client and server, and the other hardware components besides the processor hosting SGX. Per the usual SGX threat model (Section II-C), we assume the OS is compromised and may run concurrently on the same hardware, such as in adjacent hyperthread/SMT contexts, on neighboring physical cores, etc.

Security goal. 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 in Section II-C. This is formalized in our security analysis (Section VII).

Security non-goals. We do not defend against hardware attacks such as power analysis [51], EM emissions [68], compromised manufacturing (*e.g.*, hardware trojans [95]), or denial of service attacks. Also, our current implementation does not have mechanisms to mitigate speculative execution attacks [50] beyond default SGX protections (e.g., flushing branch predictor state on context switches [24]). If additional protection is needed, our backend (an SGX enclave) can be re-compiled with a software-level protection such as speculative load hardening [40].

Functional correctness. While we designed DOVE to preserve semantic equivalence (functional correctness) between the input high-level program and output DOT plus its subsequent execution, we do not have a formal proof that our implementation does indeed preserve semantic equivalence. This is in line with other data-oblivious compilation frameworks (e.g., [59]) and we consider such end-to-end verified compilation to be important future work. Note that even if the DOVE frontend has bugs, leading to functionality- or security-related issues, our security guarantee in Section VII still holds.

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.

IV. Attack Examples

A major problem this paper addresses is how to protect R programs 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 [10], [20]). We demonstrate subtle $\mu Arch$ side-channel attack vectors that come up in this approach, using the code snippet in Figure 1 as a guiding example. This code is found in 4 of 13 evaluation programs that form a public repository of code for genomics research. Three additional programs from these evaluation programs feature a similar snippet. We explain what these programs are in Section VIII-B. The program takes as input a set of samples made up of diploid Single Nucleotide Polymorphisms (SNP) sequences and outputs the number of samples that express a given genotype for each SNP position. We use R version 3.2.3 to illustrate these attack examples.

A. Example Walkthrough

The program represents the database of samples as `geno`, 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.

Computationally, the code works as follows. Line 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 that we use in Section VIII-B 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**. Then, the code in Lines 3–5 produces three vectors `n0,n1,n2`, where R "applies" the **sum()** function on each row (specified by the second argument **1**) such that each vector is the count (sum on rows) of the number of samples that express each genotype.

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. Yet, as we now show, this code nonetheless leaks privacy through $\mu Arch$ side channels.

B. Logical Operators

We start with Line 1 in Figure 1, specifically the logical **&** operations performed between the masks. At the level of R code, these look like safe data-oblivious operations

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

```
geno[(geno!=0) & (geno!=1) & (geno!=2)] <- NA
geno <- as.matrix(geno)
n0 <- apply(geno==0,1,sum,na.rm=T)
n1 <- apply(geno==1,1,sum,na.rm=T)
n2 <- apply(geno==2,1,sum,na.rm=T)
```

(Section II-D). Recall that the dimensions of `geno` are non-sensitive. 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.

First, the code is transformed from R into C calls by the R interpreter, shown in Figure 2a. 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 [88] or by replaying the victim [81] and measuring time, branch predictor state, etc. (see below). 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, this information is leaked for every SNP position.

Second, the compiler compiles the resulting C into assembly, which leaks additional information. Consider Figure 2b, which is the assembly for Lines 1 to 2 in Figure 2a, Note that the C standard requires short-circuit evaluation for the logical **||** operator such that if the left operand is true, the right operand is not evaluated. Depending on the outcome of the left predicate `x1 == 0`, the code at the assembly level will again take different paths. Hence, the attacker learns not only whether one of `x1` or `x2` equals 0, but also learns information about *which* one of them equals 0.

Figure 3 counts the number of instructions executed at the assembly level for each possible input to **&**. 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.

To test how small differences in instruction count translate into measurable effects, we conduct a simple experiment. We measure the number of cycles taken to evaluate one million iterations of expression **0 & 0** against those of **1 & 0**. Note that the execution length of these two expressions only differ by two x86-64 instructions in Figure 3. 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 [81]. We make 100 trials of such measurements against R with the Intel Performance Counter Monitor (PCM) [27]. On average, it took $\mu_{00} = 73.9$ million cycles ($\sigma_{00} = 441$K) for **(0 & 0)**, 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.

Similar issues exist for other logical operators **|** and **xor()**. In fact, **xor()** is implemented using R-level **&** and **|**

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

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

```
if (x1 == 0 || x2 == 0)
    pa[i] = 0;
else if (x1 == NA_LOGICAL || x2 == NA_LOGICAL)
    pa[i] = NA_LOGICAL;
else
    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: ...
```

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

operators. Even binary comparison operators such as == and != have similar issues. For example, R's implementation of both these operators uses branches at the R level to first check if either operand is NA.

C. Functions

Aside from R-level primitive operators, R also has a large library of functions written in either R or C. In Figure 1, we see base R functions as.matrix(), apply(), and sum(). The as.matrix() function simply converts geno to a matrix object, similarly to dynamic_cast in C++.

In Line 3, geno==0 produces a matrix of booleans (a mask) similar to those created on Line 1. Then, apply() invokes the sum() function for each row (dimension 1) that counts the occurrence of TRUE in each row of the argument matrix. Calls to apply() in Lines 4-5 have similar issues.

sum() for integers and booleans is implemented in C as isum() in the R source [26]. Interestingly, this code does perform accumulations data-obliviously; that is, each boolean is treated as 0 or 1 and accumulated without a branch that checks if (TRUE) or if (FALSE). Yet, the code *does still branch* based on whether the current value is NA before accumulation, once again leaking which entries are NA.

D. Data-Dependent Constructs

Finally, any code construct that is not data-oblivious in C is more-than-likely not data-oblivious in R. For example, an if statement in C with a sensitive predicate can reveal that predicate to the SGX adversary [1], [34]. Likewise, an if statement in R with a sensitive predicate causes an even larger (easier to measure) perturbation in program execution, due to the additional steps taken to execute that branch on hardware.

E. Discussion

These examples are only a small subset of the parts of R that leak sensitive information. R is a large code base comprising 992,564 lines of code with sophisticated runtime mechanisms such as just-in-time compilation [15], and is composed of hundreds of API functions and other features, implemented in a combination of R, C and Fortran [74].[1] Not to mention, even when an R script makes it to assembly code, we must still worry about microarchitecture-specific unsafe instructions that modulate hardware resources as a function of their input operands (e.g., [5], [25], [43], [98]).

This presents 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 [79].

In the next section, we address this challenge by designing the first secure R stack, where data scientists can program in (nearly) unchanged R, interact with the same R functionality with which they are familiar, and have strong confidence there are no latent side channels.

V. Design

We now describe the Data-Oblivious Virtual Environment (DOVE). This begins with a design overview and summary of design benefits (Sections V-A, V-B). Section V-C discusses the Data-Oblivious Transcript (DOT), which serves as the link between high-level programming and data-oblivious execution. Section V-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 V-E describes the DOVE backend, an SGX enclave that converts the DOT operations on pseudonyms to data-oblivious computation on the actual sensitive data.

A. Design Overview

As stated in the threat model (Section III), DOVE's security objective is to evaluate programs written in high-level (e.g., interpreted) languages in a data-oblivious manner. 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

[1]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.

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 4. 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 V-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.

Sensitive & non-sensitive data. Our goal is to make execution independent of sensitive data from a $\mu Arch$ side channel perspective. However, like other systems enforcing a similar information flow policy, DOVE allows for *non-sensitive* (i.e., public) data to influence attacker-visible execution. This is an important performance optimization. For example, matrix operations on rows and columns are common in our target domain; such operations' performance is largely a function of the matrix dimensions, which are usually non-sensitive. DOVE performs optimizations based on non-sensitive data during DOT construction (in the frontend) by generating the DOT with concrete non-sensitive inputs. This includes concretizing dimension arithmetic, loop bounds and control flow—when they are not a function of sensitive data. As we discuss in Section V-D, certain constructs such as loops with sensitive loop bounds are disallowed.

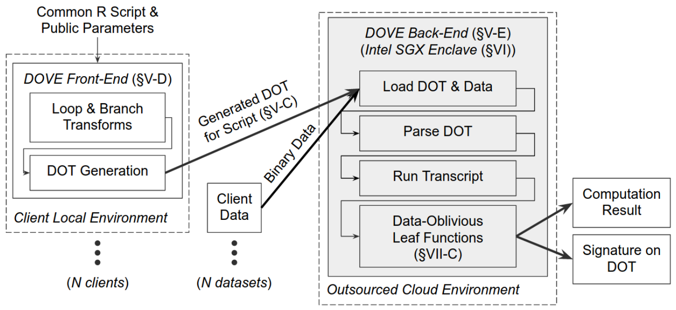


Fig. 4: High-level overview of DOVE. Bold-face arrows between nodes represent communication over (mutually-authenticated) TLS, while thinner ones are intra-process communication within a component. Shading indicates the location of our trusted computing base (TCB).

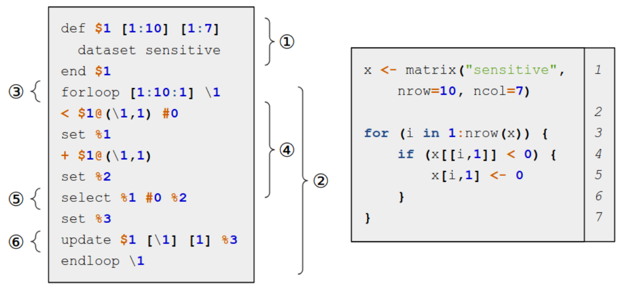
B. Benefits of Proposed Architecture

The above two-phase architecture has the following security, performance, and extensibility benefits.

- **Small trusted computing base.** The only part of the DOVE architecture that actually handles sensitive data is the backend, which is made up of a relatively small C/C++ codebase featuring 7,001 lines of code, 4,295 of which consist of a previously-vetted, external data-oblivious fixed point library [5]. Importantly, the R stack (with its almost 1 million lines of code [74]) is not in the trusted computing base.
- **No use of cryptographic encrypted computation.** Our design performs data-oblivious computation without resorting to encrypted computation techniques (such as homomorphic encryption and garbled circuits).
- **Minimal changes to programmer-facing interface.** The DOVE frontend performs a set of automated transformations (e.g., if-conversion, converting loops with early exits to guarded loops [25], [59]) to make input R code compatible with DOT semantics. As we show in our evaluation, the client can typically submit their unmodified R programs to the frontend and therefore is not required to learn a new language.
- **Extensibility to other languages.** Because the DOT decouples *R semantics* from *backend semantics*, DOVE can in principle support additional languages by implementing a new frontend without rewriting the backend.
- **Extensibility to other threat models.** Because the DOT decouples *backend semantics* from *R semantics*, if a new $\mu Arch$ side channel is discovered that undermines backend security, the backend can be patched locally without necessarily making a change to the frontend.

C. Data-Oblivious Transcript (DOT)

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



```
def $1 [1:10] [1:7]
  dataset sensitive
end $1
forloop [1:10:1] \1
< $1@(\1,1) #0
set %1
+ $1@(\1,1)
set %2
select %1 #0 %2
set %3
update $1 [\1] [1] %3
endloop \1
```

```
x <- matrix("sensitive",          1
    nrow=10, ncol=7)
                                  2
for (i in 1:nrow(x)) {            3
    if (x[[i,1]] < 0) {           4
        x[i,1] <- 0               5
    }                             6
}                                 7
```

Fig. 5: 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 V-E).

What to include in the DOT semantics strongly influences the TCB size in the backend and DOVE's overall performance. We designed the DOT semantics to follow the program counter (PC) model [66], at the granularity of primitive operations supported by the DOT. That is, 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. We chose this design because while it can be difficult to transform normal programs to the PC model, it is generally much simpler to turn PC model programs into constant-time/data-oblivious programs.[2] For example, to convert an if-else style conditional to the PC model, a compiler (or similar) needs to convert the conditional to a predicated execution abstraction, which can be complex depending on whether the conditional is nested, etc. However, converting predicated code into data-oblivious code usually entails simple transformations such as replacing point instructions with other side channel-resistant instructions such as `cmov`. Thus, since the frontend is not in the TCB and the backend is in the TCB, we have pushed the complex program transformation tasks into the frontend, and therefore out of the TCB.

Then, what primitive operations to include in the DOT semantics becomes a security/performance trade-off, because the cost to parse each operation in the DOT incurs non-negligible overhead in our current implementation (Section VI). 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 V-D.

[2]We note that our current implementation implements data memory-trace obliviousness [58] in a simplistic fashion. For example, if a data memory access has a sensitive address, we implement that access as a naive "scan memory" Oblivious RAM-style lookup. Depending on parameters, future work can improve this using a poly-log overhead contant-time Oblivious RAM client [75], [78].

We now discuss DOT semantics in more detail, using Figure 5 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 [52].

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

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. For example, in Figure 5, the input **nrow**(x) of an R program (right) is translated into a concrete value **10** by the frontend, used as a fixed-loop bound in ③ for a DOT (left). This is possible because **nrow**(x) is fixed as **10** in line 1 of the original R program. Likewise, a concrete value **0** that is being assigned to `x[i,1]` is transcribed with a prefix **#** in ⑤ in order to indicate that it is a concrete value.

The two basic types of pseudonyms are matrices and scalars, with matrices being composed of $m \times n$ scalar (i.e., numeric) elements. ① in Figure 5 shows the definition of such a matrix, with $m = 10, n = 7$. Matrices are indicated with **$**, scalars with **%**. 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.

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 6. 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 Sections III and VII). 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 [76], [85] 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. Certain constructs, such as `forloop`, are difficult (and sometimes impossible) to implement data-obliviously with respect to their arguments—which motivates why we place them in the Unsafe category. Thus, the trade-offs in deciding whether each operation is Safe vs. Unsafe are analogous to those for deciding which instructions should be made Safe vs. Unsafe in the Data-Oblivious ISA [98].

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$\not\rightarrow$Unsafe instruction*, which is analogous to the classic policy *High$\not\rightarrow$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 V-E, VI), and invalid DOTs are disallowed.

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 [66]).

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. ② in Figure 5 corresponds to the body of the loop. In the example, ③ defines the bounds of the loop (from 1 to 10 in steps of 1), along with the loop index, `\1`. 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.

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.

This flow is shown in ④ in Figure 5. This predicated execution model, similar to that used in prior work [25], [75], facilitates data-oblivious branching. That is, once conditionals are re-written to `select`, it is relatively straightforward to further convert them to backend-specific data-oblivious operators such as `cmov` (Section II-D).

Going back to our example, in ④, the condition is a **`<`** comparison between `sensitive` at row `\1` (loop index), column 1, with the scalar value 0. This condition is a pseudonym, and thus it cannot be evaluated directly using the DOT alone. The `select` (⑤) uses this condition, and if it is true, returns a scalar 0. Otherwise, it returns the value already in that location (**`$1@(\1,1)`**). This result value, stored in **`%3`**, is placed back into the **`[\1,1]`** position (⑥). In other words, the location is updated with either 0 or itself, based on the condition, in a data-oblivious fashion.

D. Frontend

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, as described in Section II-A 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 6 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.

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

Bounded iteration. Native R's **`for`** loop is not DOT-aware, so it just repeats the body of the loop m times. The frontend will naively record repeated invocations of the loop body every iteration; this results in the DOT size being proportional to

Fig. 6: 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	`abs`	`sqrt`	`floor`	`ceiling`	`exp`	`log`	`cos`
(in TCB)	`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	`fisher.test`	`pchisq`	`mean`	`colMeans`	`colSums`	`rowMeans`	`rowSums`
(not in TCB)	`is.finite`	`as.numeric`	`as.matrix`	`apply`	`lapply`	`unlist`	`which`
	`data.frame`	`matrix`	`split`	`pmin`	`pmax`	`nrow`	`ncol`
	`len`	`t`					

complexity of the loop. Instead, the frontend automatically transforms such bounded loops to use the `forloop` DOT construct. As discussed in Section V-C, explicitly defining the `forloop` is purely a performance enhancement. 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.

We also note that many loops are written with early termination (e.g., **`break`**) conditions. When a **`break`** statement is encountered, the frontend first examines a predicate associated with the break condition and its associated operations. Then the frontend performs a transformation to each statement in the loop to mask out architectural state updates, using `select`, once the break condition has been tripped. This transformation is similar to those of prior works [16], [59].

Predicated conditionals. The frontend must translate conventional if-then-else structures into the predicated execution model supported by the DOT (Section V-C). For this, we implement an if-conversion transformation that is similar to prior works [25], [75]: 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 V-C) in the DOT. The branch in Figure 5 is converted to ④ through this process. Updating the matrix value at the current position with either 0 or itself retains the semantics of the original **`if`** statement while making the operation explicitly data oblivious. 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.

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 VIII-B 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 (Section VI).

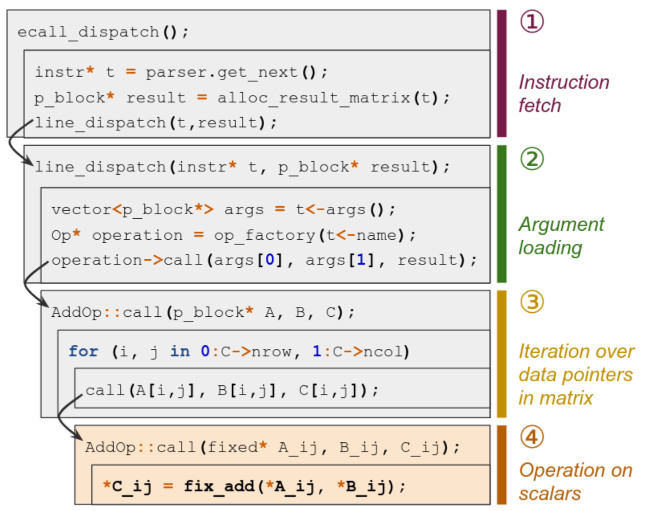
Support for integrity protection. While our primary focus is privacy, DOVE can achieve integrity through SGX’s attestation support. Specifically, each client can run the frontend and generate the DOT locally. These DOTs, or their hashes, can be checked against the DOT evaluated on the server side, by a DOVE backend that is attested to perform such checks.

E. Backend

The backend is a trusted SGX enclave 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 V-C), and includes implementations of all operations in the DOT semantics, i.e, those listed under Safe and Unsafe “DOT/Core” in Figure 6. 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 4.

The scope of DOVE is to block all non-speculative $\mu Arch$ side channels (Section III). For this purpose, the backend provides a data-oblivious implementation for operations in Safe “DOT/Core” of Figure 6. To implement these operations, we rely on a subset of the x86-64 ISA and well-established coding practices [25] for implementing constant-time/data-oblivious functions (see Section VII 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.,



```
ecall_dispatch();
  instr* t = parser.get_next();
  p_block* result = alloc_result_matrix(t);
  line_dispatch(t,result);
line_dispatch(instr* t, p_block* result);
  vector<p_block*> args = t<-args();
  Op* operation = op_factory(t<-name);
  operation->call(args[0], args[1], result);
  AddOp::call(p_block* A, B, C);
    for (i, j in 0:C->nrow, 1:C->ncol)
      call(A[i,j], B[i,j], C[i,j]);
      AddOp::call(fixed* A_ij, B_ij, C_ij);
        *C_ij = fix_add(*A_ij, *B_ij);
```

Fig. 7: A simplified graph representing the flow of `ecall_dispatch()` for the addition instruction **+**. The final, bold-face portion is the only block that dereferences sensitive data.

by microcoding the `cmov` into a branch plus a move [98]). 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.

VI. IMPLEMENTATION

We now discuss the backend implementation, the C++ codebase that evaluates DOT operations and forms the DOVE TCB. We will rely on this description during our security analysis (Section VII). We eschew detailing our frontend implementation as it is outside the TCB. As noted previously, the backend code runs in an SGX enclave. The three functions that run in this secure enclave (or `ECALL`s in SGX parlance) are associated with the three phases of the backend: loading sensitive data into secure memory (`ecall_load_data`), parsing the DOT into an abstract syntax tree (`ecall_parse`), and evaluating DOT operations based on said tree (`ecall_dispatch`).

Loading the data. The enclave loads client data, as it is received, into the SGX EPC (Section II-C). The binary blobs received consist of 8-byte, little-endian **`double`** values along with metadata about the dataset format (*e.g.*, tensor shape). The dataset is then copied into enclave memory via `memcpy`, and converted into a fixed-point integer representation. The dataset in memory is stored in a `p_block` data structure, which consists of a matrix of pointers to the scalar data values in memory and the matrix dimensions it represents.

Parsing the DOT. This phase involves recursive-descent parsing of the DOT into an abstract syntax tree (AST). Conceptually, this tree is akin to a list of instructions (see Figure 5) to be interpreted by the enclave. Recall, the DOT is created by the frontend without access to sensitive data. The AST, by extension, is not a function of sensitive data.

Importantly, the parsing process verifies that the DOT complies with DOT semantics. Specifically, that no disallowed construct appears in the DOT (Section V-D) and that each DOT operation type checks (Section V-C). The latter ensures that pseudonyms cannot be downgraded to non-pseudonyms (Rule 1, Section V-C) and that pseudonyms are not passed as operands to Unsafe operations (Rule 3, Section V-C).

Evaluating the DOT. After loading datasets and parsing the DOT, DOVE is ready to run instructions from the DOT on the data. Broadly, this phase occurs in four steps per instruction across four functions in the backend. Figure 7 depicts a simplified call graph for the addition instruction (**+**). Running different instructions entails a similar call graph.

During instruction fetch (① in Figure 7), the `ecall_dispatch()` function is the top-level call that fetches the next instruction from the DOT to be run and also allocates a placeholder datatype for use with the results of the instruction: in our running example, this is a matrix `C` of type `p_block*`. The argument loading step (②) loads pointers to instruction datatypes that form the arguments to the instruction. Our example loads two matrices: `p_block* A` and `p_block* B`. The iteration step (③) utilizes polymorphism to dispatch the backend operation corresponding to the instruction.

When one or more operands are matrices, as in Figure 7, we must perform the addition operation over all matrix elements. So, the final step is to iterate over all elements in the matrices and pass each element's pointer to a subcall to actually operate on the scalars in the matrix.

In this final step, control reaches a *leaf function*, the highlighted, bottom level of the graph in Figure 7. This is the only step when the scalar elements of the sensitive data matrix are dereferenced and utilized in the operation. At this point in our example (④), we use the external libFTFP library to perform a data-oblivious operation, addition in our example, on actual scalar values.

VII. SECURITY EVALUATION

We first present our formal definition of security under the SGX adversary first discussed in Section III. We then argue that this security definition holds given our DOVE backend implementation, for the setup and instruction execution phases.

A. Security Definition

In order to analyze the security properties of DOVE, we first formalize our security definition. We denote an execution of the R interpreter as $R(S, D)$, where S is an R program, and D is the data on which the program S is run. The SGX adversary's view of $R(S, D)$ (i.e., the leakage trace) is denoted $\mu Arch$. As discussed in Section III, this view includes hardware resource usage at a fine spatial- and temporal-granularity. For example, the attacker can monitor contention for the cache or other hardware resources, the program runtime, etc. However, due to the virtual isolation provided by the SGX TEE, the view does not include the enclave memory itself.

From our analyses in Section IV, it is clear that $\mu Arch[R(S,D)] \neq \mu Arch[R(S,D')]$ for certain D and D'. That is, the adversary's view of the $\mu Arch$ side channels of the execution of the R interpreter on a program S is different for different datasets D, D'. In practice, this means that the adversary can glean information from the datasets via these side channels, implying the computation for the given view is data dependent.

Now, we consider the notation for DOVE. We define the frontend as $A : S \mapsto T_S$, a compiler A that translates S into a DOT T_S. This computation is vacuously data-oblivious, since no data is passed as a parameter to A. We then define the backend as $V(T_S, D)$, a virtual machine that runs the DOT on the data. We aim to show that DOVE is secure against an SGX adversary, with the following definition.

Security Definition. We say a Data-Oblivious Virtual Machine backend V is *secure in the SGX adversary model* if for any pair of datasets D, D', and DOT T_S compiled from a program S, we have the following equation.

$$\mu Arch[V(T_S, D)] = \mu Arch[V(T_S, D')]$$

where $\mu Arch$ denotes the adversary's view in the SGX adversary model.

This is equivalent to a non-interference property, where high (sensitive) state is the backend input data and low (non-sensitive) state is other architectural state in the processor (across all running programs). We show that the DOVE implementation meets the above security definition through an analysis of the flow of sensitive data through the backend and compiled object code the backend uses.

B. Security Argument

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. We analyze this further in Section VIII.

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`. For our security analysis, we ran DOVE without SGX enabled for easier inspection of potential side channels. Our security evaluation related to side channels is independent of SGX, with the enclave technology being an implementation choice to guard against direct introspection/tampering by supervisor software. We provide a performance evaluation of DOVE in Section VIII.

The backend forms our trusted computing base, and it consists of 7,001 lines of code, of which 4,295 lines is the libFTFP library which we adopt from prior work [5]. Since the frontend has no access to sensitive data, the security evaluation of DOVE reduces to the evaluation on the remaining 2,706 lines of code in the backend. We now examine each step of DOVE's workflow prior to the leaf function call (described in Section VI). Finally, Section VII-C examines the leaf functions.

Creating the DOT. Each client runs their R program on their local frontend to produce a DOT. The adversary can only learn non-sensitive information (e.g., dataset dimensions) explicitly given to the frontend.

Transferring the data and DOT. After DOT creation, the DOT and sensitive data is sent by the user to the server through a secure channel whose endpoint is within the TEE [7]. The secure channel and SGX-provided attestation ensures that the correct DOT is run and that the data is privacy/integrity-protected in transit. In the case of multiple users submitting data and DOTs, this process is applied to each user, after which the DOTs are hashed and compared to ensure the computation is consistent with that requested by each user (also see Section V-D).

Loading the data. Once datasets arrive, the SGX backend stores the data in the enclave by passing the datasets through `ecall_load_data`. As mentioned in Section VI, each dataset is a binary blob which is copied into enclave memory via `memcpy`. This operation consists of a sequence of data-oblivious `mov` operations. Prior to the copy, we convert floating point values in each sensitive dataset to fixed point numbers (to be compatible with libFTFP [5]). This conversion process is also data-oblivious; further, the number of bits in the fixed-point representation is independent of the underlying value.

Parsing the DOT. As noted in Section V-C, the DOT contains no sensitive information. By extension, the DOT parsing phase—whereby the DOT is parsed into an AST—cannot leak sensitive information. Recall that the parsing process also ensures the DOT complies with DOT semantics (Section VI), which ensures that Rules 1 and 3 are enforced (Section V-C). The DOT grammar is simple, so type-checking the DOT is also simple.

Evaluating the DOT. This phase occurs in four steps, as shown in Figure 7: instruction fetch, argument loading, matrix iteration and operation on dereferenced sensitive data. The first three phases do not perform operations on the underlying sensitive data and only operate based on the DOT, which is a function of non-sensitive information as discussed above. Specifically, until the runtime reaches the operation in the leaf function, pointers to the sensitive data are passed around, but the sensitive data values are never accessed.

C. Leaf Functions

Based on the above discussion, only leaf functions read and modify sensitive data. Thus, we now scrutinize whether these leaf functions enable our security guarantee, i.e., uphold Rule 2 from Section V-C. 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. The following analysis applies well-established principles for writing constant-time and data-oblivious programs (Section II-D).

We first analyze the leaf function instructions that take sensitive data as operands. These instructions are shown in Figure 8. We determined this set by inspecting instruction dependencies in the `objdump` disassembly. All but one of the

Fig. 8: 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		

opcodes in Figure 8 is considered to be a data-oblivious instruction by libFTFP, our constant-time fixed-point arithmetic 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 [75]. 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 8, 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.

To further corroborate our static security analysis, we also looked at runtime instruction statistics. We used the branch-trace-store execution trace recording [48] 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 [66]. 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). Additional details regarding dynamic security analysis of the DOVE backend can be found in the paper's extended version [52].

VIII. Experimental Evaluation

We now turn to the experimental evaluation of DOVE in three areas: (1) correctness, (2) expressiveness, and (3) computational efficiency. It is necessary to provide some evidence that computed values are correct, at least for a basic collection of computations. Since DOVE works with a subset of R, it is also important to demonstrate that it can code enough interesting cases to be worthwhile. Moreover, DOVE computations must sufficiently limit computational overhead. We carry out the validation via two case studies, using the same machine configuration that we introduced in Section VII for experimental evaluation. The first echoes prior work [79] by coding and analyzing applications of the PageRank algorithm [72]. The second examines a suite of programs [18] for genomic analysis and a case study using it for the analysis of honeybee genomes [8]. It is easier to work with this type of data than, say, genomic data of people with bipolar disorder, while it illustrates similar issues of scale and the potential value of controlled data sharing.

For correctness, we confirm that what we get from DOVE is the same as what we would get from R. That is, using the notation of Section VII-A: $Q[V(T_S, D)] = Q[R(S, D)]$ where Q denotes the calculated output for a given execution. For expressiveness, we demonstrate that we can conveniently create DOTs from R code for each case study. As such, we devote most of the section to the evaluation of computational efficiency.

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, 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 [28]. 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$\times$ for `neg` (operand negation) to 208$\times$ for `exp` (exponential function evaluation) [5].

A. 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 [79]. A large proportion of this algorithm is composed of matrix multiplications, which other works choose as primary performance benchmarks [57], [75].

We use the Wikipedia vote network (WikiVote) [53] and Astro-Physics collaboration network (ca-AstroPh) [54] datasets from the Stanford Network Analysis Project [55] for this case study. Both datasets are converted to adjacency matrices, where WikiVote has 7,115 nodes ($\approx$ 405 MB) and ca-AstroPh has 18,772 nodes ($\approx$ 2.8 GB). Figure 9 shows the runtimes for the PageRank algorithm on two datasets where the DOVE frontend took on average of 3.34 seconds for each dataset. Note that we evaluate several configurations for the DOVE backend (SGX, libFTFP) as discussed before.

Vanilla R ran faster than DOVE, even without the SGX enclave and without libFTFP. This is expected for several

[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.

Fig. 9: Runtimes for running PageRank algorithm on different configurations. All measurements are in seconds, and the measurements are sums of frontend and backend runtimes. The frontend took on average of 3.34 seconds.

Configurations	WikiVote	ca-AstroPh
Vanilla R	6.91	46.88
DOVE w/o libFTFP, w/o SGX	23.70	122.18
DOVE w/ libFTFP, w/o SGX	137.00	951.62
DOVE w/ libFTFP, w/ SGX	509.04	2,254.46

reasons. First, R uses the highly-optimized Fortran BLAS library for matrix multiplication, while DOVE does not. Second, DOVE code (with or without libFTFP) disables compiler vectorization for safety reasons. Finally, DOVE uses the data-oblivious x86-64 `cmov` for any conditional statement on sensitive data whereas the R interpreter is written with unsafe branch statements (Section IV).

Enabling libFTFP increases runtime overhead of DOVE by around 6×, and enabling SGX on top of libFTFP incurs 3× additional overhead. The PageRank implementation shows that DOVE is expressive enough to handle a common data-processing algorithm without severe performance degradation.

B. Genomic Analysis

To further validate DOVE, we work with an application that performs a controlled study on honeybee genomic data [8]. The study relies on R code drawn from a set of 13 genetics research programs [18] that implement important statistical measurements found in the literature [37], [69], [86], [92], totaling 478 lines of R code [18]. These programs, in addition to the coding of PageRank, constitute a practical illustration of the expressiveness of DOVE. The paper's extended version [52] provides more details about the programs' applications to genomics.

Using DOVE, we were able to transform (in the frontend) and run (in the backend) 11 out of the 13 evaluation programs, totaling 326 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., [78], to avoid such worst-case work.

We utilize the dataset from the honeybee study [8] 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 [22], [23], [77].

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 10.[4] 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).

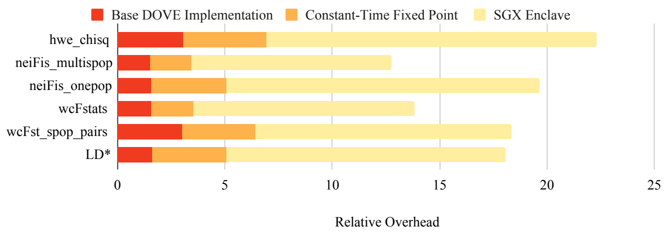
The min/avg/max size overhead of each DOT relative to its R script is 0.284x/10.8x/105x. Note, the DOT may be smaller than the original program because of the DOT instruction set. We expect that the DOT can be significantly compressed. Case in point, the current DOT is represented in ASCII which is space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

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

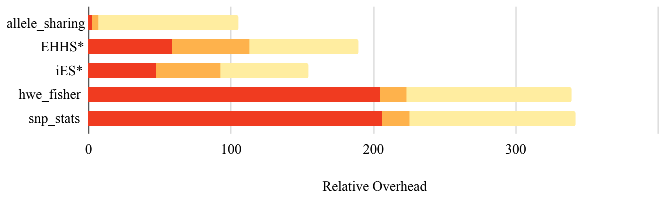
Programs with quadratic space complexity. The relative overhead with DOVE is 120.7× against vanilla R on average for programs `EHHS`, `iES`, and `LD`. These three programs run statistics based on pairwise SNPs, i.e., a row is compared to each other row in the dataset. They operate in $O(m^2)$ space, 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 **`break`**s 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.

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 6), 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.

[4]Numbers for each program's absolute runtime are given in the paper's extended version [52].



(a) Programs with less than $25\times$ relative overhead.



(b) Programs with greater than $25\times$ relative overhead.

Fig. 10: 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.

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.

Remaining programs. 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$\times$ relative to vanilla R for these programs. One program, `allele_sharing` (in Figure 10b), 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.

IX. Discussion and Future Work

Current prototype limitations. The DOT has been designed to provide functionality for real-world data science tasks. Some features such as multi-threading and networking, that are present in general-purpose languages, are not currently supported in either the DOT or DOVE more generally. This is not fundamental. Additional functionality can be added to the DOT, as long as there exists a data-oblivious implementation of said functionality.

Handling loop bounds (and related constructs) that depend on sensitive data. In the current DOVE prototype, loop bounds (and related constructs such as recursion depth) must be a function of non-sensitive data. For example, we consider matrix dimensions to be non-sensitive and matrix dimensions determine loop bounds in our evaluation scripts. An interesting direction for future work would be to add either static or dynamic program analysis to enable such control-flow information to be a function of sensitive data. For example, if a given loop iterates i_1 or i_2 times depending on a sensitive value, one would like an analysis to discover i_1 and i_2, set the loop's bound to $\mathsf{max}(i_1, i_2)$ and add the instrumentation from Section V-D to mask out architectural state updates when the actual input requires fewer loop iterations.

Possible performance optimizations. Finally, as our primary objective was to demonstrate a proof-of-concept of DOVE's security benefits, we believe many performance optimizations are possible. For example, there are performance-optimized data-oblivious implementations of several key primitives (e.g., sensitive array lookup [78], matrix multiply [79]) which could be integrated into our backend to improve performance without changing the DOVE architecture. Finally, as mentioned in Section V-C, users can add frequently-used routines as DOT primitives implemented in the backend, to trade-off performance and TCB size.

X. Related Work

A. SGX Programming

Our work is related to prior efforts in running/partitioning/-managing general purpose applications in SGX [6], [10], [20], [35], [39], [46], [56], [78]–[80], [89]. The four most relevant axes for comparison are: (i) whether the application running is untrusted, (ii) whether the proposal runs interpreted code such as R, (iii) what is the threat model (in particular, does it include defense against $\mu Arch$ side channels) and (iv) whether the proposal requires a new custom programming language. We show a comparison along these axes in Figure 11. The takeaway is that no prior proposal, to our knowledge, simultaneously runs (i) untrusted code, (ii) high-level interpreted code such as R, (iii) provides broad protection against $\mu Arch$ side-channel attacks, (iv) supports existing languages. Moreover, our work makes a distinct conceptual- and design-level contribution, namely to orchestrate computation through existing high-level languages without showing those languages the sensitive data.

The works most similar to our proposal are TrustJS [39] and SGXBigMatrix [79]. The former runs untrusted JavaScript code but assumes a weaker adversary that cannot monitor fine-grain application behavior over, e.g., side channels. The latter provides a data-oblivious matrix API, but requires programmers to adopt a custom scripting language for performing computation. We view this work as complementary: our work strives to enable general-purpose data oblivious computing on *existing high-level languages*, but could benefit from the performance optimizations made to matrix computations in SGXBigMatrix.

B. Data-Oblivious Programming

There is a rich literature that studies how to write and run different applications in a data-oblivious fashion on today's ISAs. For example, application-centric works propose data-oblivious cryptography [11], [12], machine learning [70], [79], databases [32], [63], [101], memory and datastructures [2], [78], general purpose code [25], [29], [35], [66], [75], utilities [87] and floating point functions [5]. Many of these [29], [32], [35], [36], [63], [70], [75], [78], [79], [101] were designed for SGX-enabled applications to block the $\mu Arch$ side channels discussed in Sections II-B-II-C. Programming language, compiler and runtime works study how to write (e.g., [16], [30]) and compile (e.g., [59], [82], [100]) programs to software circuits. ISA abstractions study how to design interfaces usable by both software designers and hardware architects to uphold data-oblivious security guarantees [98]. These efforts are backed on the theory side by studies on how to run different algorithms and data structures in the circuit model [13], [19], [31], [59], [67], [82], [83], [91], [99].

Our work differs from these application-centric works by targeting high-level interpreted programming stacks such as R as opposed to low-level C. As discussed in Section IV, R introduces significant new challenges in establishing confidence that code is actually data-oblivious. Our work differs from the PL, runtime, compiler work because it focuses on hardening mostly-unmodified R code, as opposed to creating a new end-to-end stack with a custom language; our work differs from ISA work, such as the Data-Oblivious ISA extensions [98] (OISA), by not requiring ISA-level changes to the underlying machine. Finally, our work is complementary to data-oblivious algorithm and data structure design, as our backend can leverage these algorithms to implement specific operations.

C. Privacy-Preserving Genomics

Our case study is based on the genomics of honeybees collected from three different locations [8]. Genomics has been a promising test case for privacy-preserving application of SGX before in other areas. These include at least the privacy-preserving computation of admixtures [21], Genome Wide Association Studies (GWAS) [77], and analysis of rare diseases (viz. Kawasaki disease) [23]. There is also an SGX-based study of privacy-preserving queries on genomic data [22]. There is a survey [9] that discusses approaches to genomic privacy based on SGX, on cryptography, and on a hybrid of both.

XI. Conclusion

DOVE offers an approach to achieve data-oblivious computation within a TEE for programs originally written in languages with complex stacks such as R. The approach takes as input a high-level program and transforms it to an intermediate representation (DOT) that can be more easily reasoned about with respect to providing data obliviousness on a constrained TCB. This gives the advantage of being able to program in a familiar and convenient language while providing a very strong security guarantee. The trade-off for achieving these benefits, in general, is limits on the high-level programs that can be processed. We have demonstrated a design and implementation that can cover a significant range of programs with efficiency that is an acceptable trade-off for the benefits. This provides a foundation for future study using our methodology, such as expanding to richer high-level programming constructs and languages.

Fig. 11: Related works on application partitioning/management in SGX.

Name	Untrusted Apps?	Interpreted Code?	Blocks μarch Side Channels?	Supports existing languages?
Haven [10]	✗	✓	✗	✓
Graphene-SGX [20]	✗	✓	✗	✓
Scone [6]	✗	✓	✗	✓
Panoply [80]	✗	✗	✗	✓
Glamdring [56]	✗	✗	✗	✓
TrustJS [39]	✓	✓	✗	✓
ScriptShield [89]	✗	✓	✗	✓
Ryoan [46]	✓	✗	✗	✓
SGXBigMatrix [79]	✓	✓	✓	✗
ZeroTrace [78]	✗	✗	✓	N/A
Felsen et al. [35]	✓	✗	✓	N/A
DOVE (This paper)	✓	✓	✓	✓

Acknowledgments

This work was supported in part by NSF CNS 13-30491 (THaW), NSF CNS 19-55228 (SPLICE) and NSF CNS 18-16226 (OISA). The views expressed are those of the authors only.

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] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *OSDI'16*.

[7] Pierre-Louis Aublin, Florian Kelbert, Dan O'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and transparent TLS termination inside SGX enclaves. *Imperial College London, Tech. Rep*, 5:2017, 2017.

[8] 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.

[9] Md Momin Al Aziz, Md Nazmus Sadat, Dima Alhadidi, Shuang Wang, Xiaoqian Jiang, Cheryl L Brown, and Noman Mohammed. Privacy-preserving techniques of genomic data—a survey. *Briefings in Bioinformatics*, 20(3):887–895, 2017.

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

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

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

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

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

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

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

[17] 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.

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

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

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

[21] Feng Chen, Michelle Dow, Sijie Ding, Yao Lu, Xiaoqian Jiang, Hua Tang, and Shuang Wang. PREMIX: Privacy-preserving estimation of individual admixture. In *AMIA ASP'16*.

[22] 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.

[23] 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.

[24] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *EuroS&P'19)*.

[25] 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*.

[26] R core team. R source, summary.c. https://github.com/wch/r-source/blob/tags/R-3-2-3/src/main/summary.c.

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

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

[29] Cybernetica. Sharemind HI. https://sharemind.cyber.ee/sharemind-hi/, 2020.

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

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

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

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

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

[35] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and private function evaluation with Intel SGX. In *SIGSAC'19*.

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

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

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

[39] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. TrustJS: Trusted client-side execution of javascript. In *EuroSec'17*.

[40] Google/LLVM. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

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

[42] 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*.

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

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

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

[46] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI'16*.

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

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

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

[50] 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*.

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

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

[53] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *SIGCHI'10*.

[54] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM TKDD'07*.

[55] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[56] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *USENIX ATC 17*.

[57] 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.

[58] Chang Liu, Michael Hicks, and Elaine Shi. Memory trace oblivious program execution. In *CSF'13*.

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

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

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

[62] 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*.

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

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

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

[66] 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*.

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

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

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

[70] 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*.

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

[72] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

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

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

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

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

[77] 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.

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

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

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

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

[82] 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*.

[83] 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*.

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

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

[86] 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.

[87] 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.

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

[89] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian Zhang. Running Language Interpreters Inside SGX: A Lightweight,Legacy-Compatible Script Code Hardening Approach. In *Asia CCS'19*.

[90] 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*.

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

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

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

[94] 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.

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

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

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

[98] 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.

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

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

[101] 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*.