© 2023 Hyun Bin Lee

LOGS AND SIDE CHANNELS

BY

HYUN BIN LEE

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Doctoral Committee:

Professor Carl Gunter, Co-Chair
Assistant Professor Christopher Fletcher, Co-Chair
Professor Nikita Borisov
Professor Adam Bates
Research Assistant Professor Timothy Pierson, Dartmouth College

ABSTRACT

While system administrators would prefer to configure their systems so that their loggers capture as much details as possible, such details may include sensitive information. In particular, system audit logs aim to record information about system events that are triggered during any incident. Hence they inevitably contain sensitive information that needs to be kept secret. Redacting and obfuscating such information may help when such information is written explicitly on logs, but implicit information, content that is seemingly not secret but indirectly leaks secret information, cannot be handled with such measures. Unfortunately, researchers have overlooked risks of leaking implicit secret content.

This thesis first introduces *Termite attacks*, a new class of side channel attack that exploits such implicit content to learn secret information. We demonstrate how logs can be used to create both membership inference attacks and keystroke timing attacks, and show practical exploits against a common setup: a LEMP (Linux, ENginx, MySQL, PHP) webserver with SSH daemon that is audited by the Linux Audit System (LAS). Then we demonstrate how to launch a concurrency-based timing side channel attack. Contrary to the belief that log information is too coarse-grained to be exploited against modern microarchitectural side channel attacks, we demonstrate how adversaries can craft timing side channels that are as fine as 100s of nanoseconds through system audit logs.

To address such vulnerabilities associated with logs leaking sensitive information via microarchitectural side channels, we propose two different countermeasure schemes. First scheme involves data-oblivious computation and Trusted Execution Environments (TEEs) against adversarial systems. We demonstrate how to produce a side-channel resistant log by instrumenting a complex programming environment (like R) to produce a *Data-Oblivious Transcript (DOT)*. The DOT is designed so that any sensitive data from computation is decoupled from the transcript. Such transcript is later evaluated on a TEE containing the sensitive data using a small trusted computing base called the *Data-Oblivious Virtual Environment (DOVE)*.

While DOVE allows us to protect logs, deploying DOVE requires several prerequisites. In particular, another countermeasure is needed when one cannot completely decouple timing information from logs. As we have shown how to exploit such timing information to steal secrets from Termite attacks, our last study investigates how to mitigate such attack vector against aforementioned system audit logs stored on honest system that adversaries temporarily break-in. This study observes tradeoffs between the number of timestamps in logs

versus the utility of audit logs. We analyze existing utility functions associated with audit logs and choose a reachability function as a utility standard for this study. We use DARPA's Trusted Computing Dataset for this study to conduct experiments. In order to guarantee complete security against timing side channels, the logs must not contain any timing information such that no timestamp exists and the ordering of all log entries are randomized. We call such logs *Timeless Logs*. Unfortunately our reachability analysis showed such extreme measures lead to very poor utility according to our experiments against the dataset. Hence, we introduce an alternative method, *Batched Timeless Logs*. This method divides logs into n batches and make each batch *timeless*. We preserve ordering among batches, so there are n different time information stored in the logs. The size of each batch is randomized to augment security against clock-edge attacks between batches. Our experimentation on a public dataset shows that when batches are sufficiently small (batch size of around 1,000 entries or smaller), there is only an 1.47 percent increase in the number of false positive reachable nodes from randomly picked source nodes on average.

We argue that side-channel attacks against logs can leak very fine-grained information across any vulnerable application but such threats have been overlooked. Our study on such attacks and proposed countermeasures highlights some future work that should to be addressed.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First I would like to thank Prof. Carl Gunter and Prof. Chris Fletcher for advising throughout my Ph.D. When I made a big decision to change my research interests from differential privacy to side channels, Carl encouraged me to a new research problem that supported my decisions, and Chris provided valuable guidance on this unfamiliar topic. I had doubts about my research directions, but their advice helped me overcome such doubts and make valuable scientific contributions throughout my Ph.D. program.

Second, I am grateful to have Prof. Carl Gunter, Prof. Chris Fletcher, Prof. Nikita Borisov, Prof. Adam Bates and Prof. Timothy Pierson in my thesis committee. I would like to thank them for their insightful questions, helpful suggestions, and valuable feedback.

In addition, I thank Tushar Jois who collaborated with me on most of my research projects. His summer visit to UIUC was the most crucial turning point of my Ph.D. program, and it was a great pleasure to work with him on various research projects. I also would like to thank members of Security and Privacy Research at Illinois (SPRAI) and SPLICE (Security and Privacy in the Lifecycle of IoT for Consumer Environments). It was an honor to work with these great talents.

Next, I would like to thank Prof. Grigore Rosu and Prof. Michael Bailey. Prof. Rosu was my undergraduate mentor and gave me words of courage during a mentorship meeting. Thanks to his advice, I did not give up my dream. I began my graduate program with a teaching assistanceship under Prof. Bailey. While working with him, I learned what it means to be a security researcher.

Finally, I thank my parents for their continuous trust and support. With unconditional love and sacrifice, they gave me an opportunity to study in Illinois. I was able to successfully finish this journey with their continuous support.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Contributions and Roadmap 3
1.2 Background 4
1.3 Related Works 11

CHAPTER 2 TERMITE ATTACKS 16
2.1 Introduction 16
2.2 Threat Model 16
2.3 Coarse-Grained Side Channel Attacks 21
2.4 Fine-Grained Side Channel Attacks against Ordering of Log Entries 29
2.5 Conclusion 37

CHAPTER 3 DOVE: DATA-OBLIVIOUS VIRTUAL ENVIRONMENT 38
3.1 Introduction 38
3.2 Threat Model 39
3.3 Attack Examples 40
3.4 Design 45
3.5 Implementation 55
3.6 Security Evaluation 57
3.7 Experimental Evaluation 60
3.8 Conclusion 66

CHAPTER 4 TIMELESS LOGS 67
4.1 Introduction 67
4.2 Threat Model 68
4.3 Reachability 68
4.4 Timeless Logs 71
4.5 Batched Timeless Logs 74
4.6 Conclusion 79

CHAPTER 5 CONCLUSIONS 80
5.1 Discussion and Future Works 80

REFERENCES 83

CHAPTER 1: INTRODUCTION

Logging is an act of recording a sequence of events that occur in a computer system [1]. For example, system audit logging involves collecting information about system events for purposes ranging from post-mortem incident reconstruction to system diagnosis and troubleshooting. Such information is usually sensitive, so research focus has been on preventing loss of confidentiality using one of two techniques: *encryption* and *redaction*. Encryption-based techniques [2, 3, 4] rely on cryptography and specialized trusted hardware or remote trusted servers to protect log data. However, encryption-based schemes can be hard to implement due to performance overheads and interoperability issues with third-party log analysis applications [5]. Redaction-based schemes [6, 7], including obfuscation [8, 9], protect when secrets are written directly to logs. Redaction-based schemes force an inherent utility-security tradeoff. But, more critically, existing redaction-based schemes are not designed for blocking *implicit flows* from logs where where seemingly *nonsensitive* contents leak important system secrets [10]. In the past, the implicit information provided by audit logs has been studied for anomaly detection [11, 12, 13] and causality inference [14, 15, 16]. We posit that it can also be used for exploitation by enabling side channel attacks.

Attacking logs brings unique opportunities and challenges to side-channel research. On one hand, logs *persist* after the victim application runs. There is, in fact, incentive to maintain logs long after they are generated so as to better serve forensic analyses. Unlike traditional side channels, *e.g.,*, power attacks generating ephemeral power traces, this allows the attacker to learn ephemeral secret information even if said attacker is not actively monitoring the victim's execution as it runs. Such a characteristic adds big concerns associated with forward secrecy, a security feature that usually is not prioritized against micoarchitectural side-channel attacks due to ephemeral nature of execution traces. On the other hand, logs typically capture only *coarse-grained* information. For example, log entries typically correspond to major system-relevant events such as system calls. This makes it difficult to use log-based side channels to extract fine-grained information about a victim's execution (*e.g.,*, fine-grained hardware-resource usages) that is the foundation of many traditional side-channel attacks (*e.g.,*, microarchitectural attacks).

Thesis Statement: Logs are overlooked persistent sources of execution traces that are vulnerable against fine-grained side channel attacks, but appropriate countermeasures can mitigate risks against the attacks.

In this thesis, we exploit the above strengths and overcome the above weaknesses to reproduce several disparate attacks ranging from keystroke detection to fine-grained microarchitectural side channels. Our most surprising result is that, despite the fact that logs capture only coarse-grain information, *they can still encode extremely fine-grain events (on the order of 100s of nanoseconds to microseconds)*. We call these side channel attacks against logs as the *Termite attacks*[1].

To address concerns associated with Termite attacks against logs, we introduce two countermeasures. Our first countermeasure, the Data-Oblivious Virtual Environment (DOVE) [17] leverages Trusted Execution Environment (TEE) and data-oblivious programming to close microarchitectural side channel attacks against logs or transcripts that record computation of high level language interpreters. This work is originally designed to provide a secure programming environment for high-level programming languages against untrusted software including operating systems and hypervisors. The DOVE architecture is composed of two steps where a *frontend* instruments the high-level programming stack by recording computation done on input scripts into a execution log called "Data-Oblivious Transcript (DOT)". This log does not carry any sensitive data as these are replaced with stand-ins called "pseudonyms", and all operations recorded on the DOT are data-oblivious primitives. Such logs are then replayed by a *backend* with actual data supplied via a secure channel. The backend is composed only of data-oblivious functions such that computation itself is secured.

DOVE provides a strong security guarantee against a strong threat model, but its deployment may not always be suitable. In particular, DOVE assumes several attributes to be public including ordering of operations, but a clock-edge timing attack in Termite attacks exploit such "seemingly-public" information to extract system secrets. Our second countermeasure is designed under a different threat model compared to that of DOVE where we assume the system is honest but at some point is compromised by an adversary. This countermeasure is mainly designed against clock-edge attacks using a post-processing mechanism on logs. We propose a new scheme called *Timeless Logs*, a solution that started with a naive thought on removing every timing information from logs to protect logs from timing side-channel attacks. This can obviously make logs useless, but how can we quantify such utility loss? Our study begins with answering such questions with utility functions offered by prior work on log reduction techniques. Then, we introduce *Batched Timeless Logs*, a sequence of Timeless Log segments which offers reasonable utility-security tradeoffs based on our quantitative analysis against a public log dataset.

[1] Full title of the original paper that introduces these attacks is called "Termite Attacks: Gnawing on Logs to Extract Secret Information."

1.1 CONTRIBUTIONS AND ROADMAP

Overall, these are our contributions of this thesis:

- We introduce Termite attacks: side channel attacks that exploit implicit information contained in logs.
  - We systematize a threat model of possible attack vectors against system audit logs, and provide examples of potential Termite attacks that illustrate our threat model.
  - We illustrate how implicit contents written by system audit loggers may leak secrets with three scenarios.
  - We show how design of the Linux Audit System (LAS) is vulnerable to a concurrent execution side channel under normal system conditions.
  - We demonstrate a proof-of-concept NetSpectre attack with our concurrent timing side-channel against LAS to show that Termite attacks can encode extremely fine-grain events.
- We design DOVE, the first architecture that runs existing high-level interpreted languages and is demonstrably resistant to microarchitectural side channels.
  - We identify a number of subtle side-channel vulnerabilities in the R language.
  - We provide an implementation of DOVE for R, creating the first side-channel resistant R programming stack.
  - 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× to 341.62×.
- We introduce Timeless Logs, a post-processing scheme to address concurrent timing attacks against logs.
  - We study various log reduction techniques and assess utility functions suitable to quantify utility losses associated with our post-processing countermeasure scheme.
  - We conduct removal of timestamps on a public log dataset and analyze utility loss based on a reachability function, the utility function that we choose based on prior works.
  - We introduce Batched Timeless Logs that provide good utility-security tradeoffs on the same dataset under the same utility function.

1.2 BACKGROUND

This section contains background information required to understand ideas and concepts introduced throughout this thesis. We first begin this section with background on the topic of logging to give readers understanding of how modern logging systems are designed and implemented. Then, we explain key ideas associated with microarchitectural side channel attacks, the fundamental building blocks that comprises our thesis. Finally, we end with some background on R and Trusted Execution Environments to give readers important information to understand Chapter 3.

1.2.1 Logging

Application logging. System applications commonly include logging facilities that record events that occur while the application is running. Applications such as web servers [18] and databases [4, 19] all use logging to alert administrators to events. Typical events include errors in the application as it runs, as well as any security incidents. Logging information varies between applications, but usually includes a timestamp alongside the actual log message.

System audit logging. In addition to the logging frameworks of individual applications, the operating system retains its own logging functionality. Examples include Linux Audit System (LAS) on Linux [20] DTrace on FreeBSD [21] and ETW on Windows [22]. System audit logs provide a chronological record of all events that happened on an operating system at the granularity of system calls. These logs typically are stored for long periods of time as an intrusion prolongs over 188 days before the detection [23]. Functionality varies between logging frameworks, but typically they log interactions between applications and the operating system, such as user commands, file accesses, security events, and network accesses [20, 24]. System logs typically interleave logging information from different processes [25].

Linux Audit System. Linux Audit System (LAS) [20] is a system logger for Linux that is mainly comprised of two parts: a userspace daemon, and the kernel's audit subsystem that processes system calls. The architecture of LAS is denoted in Figure 1.1. Audit rules dictate what LAS monitors, and rules can be specified to capture system calls, file system behaviors, and LAS behavior itself. In system call monitoring, when a userspace application requests a system call, the framework sends information about the system call to a kernelspace

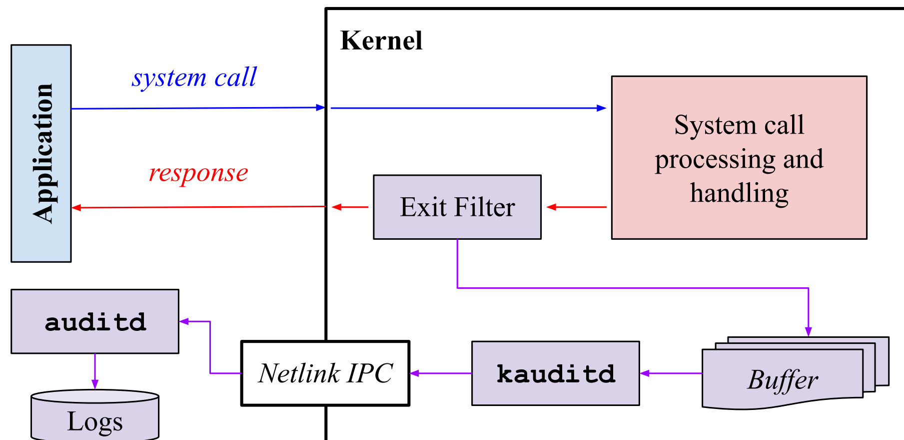


Figure 1.1: Syscall auditing in the Linux Audit System (LAS).

daemon, `kauditd`. This daemon asynchronously processes the entries in a buffer, first-in, first-out. `kauditd` then uses a Netlink IPC socket to communicate information to the userspace component, `auditd`, which in turn stores the information on disk for a user to recover. LAS logs the time a system call was executed, its syscall number, its parameters, and information about the application that executed it. It also records the raw pointer values (*e.g.*,, `void *`) and other primitive values (*e.g.*,, `int`) in system call parameters. Note that this architecture introduces some latency, as LAS handles the logging of system calls at *filters* that intercept system call control flow and checks whether the call matches with audit *rules*.

Listing 2.2 contains a LAS entry for the system call `recvfrom` when PHP receives a MySQL response.The system call `recvfrom(2)` is depicted in each entry. Some important aspects include:

- The *type* of event, which is a `SYSCALL`.
- The *timestamp*, in the `msg` field. This is a UNIX epoch time with millisecond granularity, followed by a serial number for the log entry.
- The *system call number*, `syscall`, which is 45, or `recvfrom(2)`.
- The *exit code*, `exit`, which is the return value of the system call. For `recvfrom(2)`, this is the number of bytes received.
- The *system call parameters*, in fields `a0` through `a3`. All of them are in hexadecimal.

```
-a exit,always -F exe=/usr/sbin/php-fpm7.2 -F arch=b64 -S read -S readv -S write
↪  -S writev -S sendto -S recvfrom -S sendmsg -S recvmsg -S mmap -S mprotect -S
↪  link -S symlink -S clone -S fork -S vfork -S execve -S open -S close -S creat
↪  -S openat -S mknodat -S mknod -S dup -S dup2 -S dup3 -S bind -S accept -S
↪  accept4 -S connect -S rename -S setuid -S setreuid -S setresuid -S chmod -S
↪  fchmod -S pipe -S pipe2 -S truncate -S ftruncate -S sendfile -S unlink -S
↪  unlinkat -S socketpair -S splice
```

Listing 1.1: LAS rule to log forensically-relevant system calls.

- The *process identifier*, `pid`, along with several *user* and *group identifiers*, `*uid`, `*gid` of the process. This is not only useful in the context of an individual system's configuration: what is running, and who is running it, but also useful for an attacker to find a log entry of interest.

LAS is configured by *rules* that determine its behavior [26]. The rule can filter system calls with *fields*, such as executable path, system architecture, system call of interest, users, and more. Listing 1.1 contains the LAS rule used in the generation of the log in Listing 2.2. In this example, the system calls of interest are those deemed forensically-relevant by prior work on provenance auditing, one of which is `recvfrom`. We explain how these works utilize LAS in next subsection.

1.2.2 Provenance Auditing and Rule Configurations

Provenance auditing serves to understand the entire view of system execution so that it assists causal analysis of system activities. Provenance-based works primarily focus on reconstructing the chain of events that led to an attack (backward tracing) as well as the ramifications of the attack (forward tracing). Thus, most works on provenance auditing first converts logs into dependency graphs to find relationships between events. [27]

Provenance auditing serves a crucial role in attack detection and investigation against enterprise environments [28]. The Controlled Access Protection Profile common criteria requires audit trails for systems used by U.S. Government agencies [29]. LAS was introduced in Linux version 2.6 to achieve certification of the common criteria for Linux systems [5, 30]. Hence, many works on provenance auditing use LAS for collecting system logs.

Provenance auditing has been used for forensic analysis [31, 32] and attack detection [33, 34, 35]. Other applications tackle the dependency explosion problem of LAS [14, 36, 37]. Provenance auditing has also been applied for less traditional systems, such as the Linux components of distributed systems [38] and Android devices [39]. Finally, several works

address integrity of audit frameworks [5, 30, 40].

The aforementioned related works collect system logs from LAS and their LAS configurations are consistent with that of ours (Listing 1.1).

1.2.3 Microarchitectural Side Channels

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 [41]. Over the years, numerous hardware structures—a variety of cache architectures [42, 43, 44, 45], branch predictors [46, 47], pipeline components [48, 49, 50] and other structures [51, 52, 53, 54, 55, 56]—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 [42, 57] or Drama and the DRAM row buffer [55]), as opposed to sharing virtual memory with the victim (e.g. [43]).

In a timing side channel attack, the attacker uses the time it takes a victim to perform a certain task to infer functions of the victim's data [58]. μArch timing side channel attacks are a specific type of timing side channel attack where the attacker monitors when, for how long and how the victim uses different microarchitectural resources. Such resources include the cache [45, 59, 60, 61], arithmetic units [62, 63], etc.

Speculative execution attacks. The Meltdown [64] and Spectre [65] attacks introduced the speculative execution class of attack, which uses the processor speculation to create a side channel. In particular, these types of attack rely on the processor mispredicting an event (*e.g.,*, a memory access or branch) and exploiting this misprediction to extract data. The NetSpectre [62] attack allows for such an attack to be executed over the network. This attack is an active remote attack that requires attacker to actively interact with victim machine via network in order to open a Spectre-style side channel into a process. Furthermore, the attack requires a very fine-grained measurement of latency between departure and arrival of network packets to infer latency incurred by data-dependent operations from the victim.

The NetSpectre attack requires two Spectre gadgets. The leak gadget is manipulated by the adversary such that a malicious array offset would lead to accessing a secret bit. Then the bit is compared with a public parameter in a branch where if the branch holds true, a change in μArch state occurs. Listing 1.2 is an example of a leak gadget. First, the attacker sends multiple packets to victim that runs the leak gadget for packet processing. These packets are crafted by the attacker ensure that the packets pass bound checks on `x`.

This would train the branch predictor to predict future `x` values to be within the bounds. Then, attacker would send a packet where 1) the value `x` is larger than `array_length` and 2) `array[x]` is the secret bit that attacker is interested in. Due to speculative execution, the branch is mistrained to assume that `x` would be in bounds, allowing the memory access on `array[x]` to be speculatively executed. As a result, if the value of secret bit is 1, the `array2[1 * 1024]` would be speculatively accessed such that memory block associated with address of `array2[1024]` is cached. However, if the bit is 0, `array2[0]` would be speculatively accessed. This change in μArch state due to speculative execution won't be resolved.

A transmit gadget is where latency of running the gadget depends on the μArch state that is "leaked" by the leak gadget. A common example of such gadget would be accessing index 0 or 1024 of aforementioned `array2` that is tainted by the leak gadget to cause latency associated with the leaked μArch state. The attacker would exploit such a gadget to measure the latency and infer the μArch state changes to steal the secret bit.

```
if (x < array_length) {
    secret = array[x] & 0x1;
    array2[secret * 1024];
}
```

Listing 1.2: A leak gadget for the NetSpectre attack Bounds-Check Bypass.

Clock-edge attacks. While timing attacks measure time, they fundamentally only require the attacker to be able to measure whether a secret-dependent event occurs before or after a secret-independent reference event or clock edge [66]. For example, while a cache timing attack measures the time it takes an attacker's accesses to return from the cache, fundamentally the attacker need only determine whether its accesses exceed a threshold (the cache hit latency).

Reference events/clocks can be built using a variety of mechanisms. For example, through the use of explicit coarse-grained [66] or fine-grained [59] timer operations/instructions, or through the use of implicit clocks [66] such as measuring another reference program's execution time [67, 68]. Timeless Timing Attacks [68] run two processes concurrently (a sensitive process and a reference process) and checks which completes first to deduce information about how long it took the sensitive process to run.

1.2.4 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 3.1 is an R code example from our evaluation programs that includes such operations.

Computation in R. R is an interpreted language [69], 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) [70] 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.

1.2.5 Enclave Execution and Intel SGX

Enclave execution [71], such as with Intel SGX [72], 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 [53, 73, 74, 75, 76, 77, 78, 79, 80, 81], who

will be referred to as the *SGX adversary.* 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 [82, 83], containers [84], and data structure abstractions [73] 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 [85]. The EPC has a fixed size of 64 or 128 MB, shared among all enclaves [86]. 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 [72, 85].

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 1.2.3.

First, SGX does not provide any physical isolation. Thus, nearly all of the $\mu Arch$ side-channel attacks discussed in Section 1.2.3 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 [52] and follow-on work [53] 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 [77, 78, 79, 80, 81, 87, 88].

1.2.6 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 1.2.3 [48, 73, 75, 76, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111]. 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 1.2.5.

Thus, prior works that try to achieve data obliviousness in an SGX context [73, 75, 76, 92, 93, 94, 95, 96, 98] 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.

1.3 RELATED WORKS

This section encompasses related works that cover one or more chapters of this thesis. While majority of works are directly related with one chapter, several works cover multiple chapters so that we outline these works in this introductory chapter.

1.3.1 Program Behavior Analysis

Researchers have studied the topic of understanding implicit information stored in system audit logs for various reasons. System logs often serve as forensic artifacts, so there has been several works that focus on causality inferences. Authors analyze system calls transcribed in logs [14, 15, 16]. Our work, on the other hand, focuses on exploiting such events to extract secret information through side channels. Logs with system call vectors are also explored by previous works on outlier detection. Proposed solutions include process monitoring [13], malware detection [112], intrusion detection [11, 12], and data mining algorithms for application systems [113].

1.3.2 Log Confidentiality

Most researchers are well aware of rich information provided by system logs such that many works prioritize preserving integrity of system logs against tampering attacks [30, 114]. Nonetheless, several works address preservation of log confidentiality. These works mainly

work on encryption-based schemes [2, 115] or on trusted hardware [3, 116]. Overall, we believe this line of work is orthogonal to our Termite attacks. Alternative solutions target on redacting [6, 7] or obfuscating [8, 9] secret data written on the logs. Some of the data that is redacted from system logs are IP addresses [117, 118, 119, 120] and timestamps. Google released traces from their cluster management software and systems [121, 122] where these traces have their actual timestamps converted to relative timestamps such that exact time of events are hidden but the order is preserved. However, Termite attacks that exploit concurrency (Section 2.4) are resilient against such protection measures since the clock-edge attack succeeds only with information on the ordering of two concurrent events. Reiss et al. [123] introduced Google's guidelines on releasing production logs to public. While their work introduces several obfuscation techniques to achieve log confidentiality, their techniques are solely designed for Google's production cluster traces such that implementing their solutions may not be plausible for other systems. Furthermore, their work does not provide a quantitative analysis on the effectiveness of their obfuscation techniques.

1.3.3 Implicit Secrets from Logs

While aforementioned works try to achieve confidentiality for system logs, most works fail to account for risks associated with implicit secret information contained in the logs. Those works focus on protecting contents for each log entry such that they overlook secret leaking through log perturbations of multiple log entries. A few works, however, do address this problem. Long et al. [10] proposed a software testing-based approach to achieve the property of data-obliviousness [124] for shared logs. Their works offer game-based definition of the risk of exposing sensitive information through released logs. Meanwhile, their framework enforces strict restrictions such that it may be considered infeasible in practice. Dangwal et al. [125] proposes an approach of "wringing" program traces where authors propose ways of formalizing bounds of leaked information from compressed traces. Their evaluation showed that such approach is plausible against a class of existing side-channel attacks against AES.

1.3.4 Log Reduction Techniques

While provenance auditing's top priority is to capture behavior of entire system for potential future analysis, such auditing would often lead to generation of enormous volume of logs, gigabytes per day on a single machine [27]. Such sheer volume can reduce efficiency of audit log analysis, so there has been multiple works that focus on reducing volume of logs. As mentioned in Section 1.2.2, recent provenance auditing mostly focuses on converting logs into

provenance graphs for analysis. Thus, recent log reduction work often focuses on constructing such graphs and removing events, represented as edges, from the graphs. We summarize some of those works that are particularly related with our objectives in Chapter 4.

Causality Preserving Reduction (CPR) [126] is one of first works on log reduction topic. This work preserves information flows in provenance graphs but removes redundant events mainly caused by performing I/O related activities. CPR examines interleaved flows, a presence of new input flow between two output flows. If there exists such interleaving event pattern, then events are categorized as non-redundant (*i.e.* two output flows represent two different information flows).

Reachability analysis on a dependency graph checks whether a source node can reach a destination node by checking whether 1) there is a path from the source node to the destination node and 2) timestamps of edges on the path are sorted in increasing order. Such analysis ensures utility of logs for forward and backward tracing that we mentioned in Section 1.2.2. We discuss reachability analysis further in Section 4.3. Authors of CPR takes a step further from CPR and proposes an approximation-based reduction technique called Process-centric Causality Approximation Reduction (PCAR) [126]. In addition to aforementioned check of interleaved flows, this method approximates bursty processes (processes that handle a large of system calls in short interval of time) by aggregating such events. Authors test their method against naive aggregation that aggregates events within 10-second interval and runs reachability (authors use term connectivity instead in their paper) analysis from 20,000 randomly chosen nodes. This study is very similar to what we conduct in Section 4.5, so we explain this more in detail in that section.

Dependency Preserved Reduction (DPR) [127] is motivated from CPR but takes a step beyond CPR. This work checks global reachability between any two nodes in dependency graph to make reachability checks. Once such reachability is confirmed, then other edges that also constitute such path are considered redundant and are reduced.

Aforementioned works focus on reachability as the main reduction criteria, but Forensic Validity [128] augments systematic assessment of log utility. The authors of the work provide three criteria to evaluate the utility of reduction techniques. The first criteria is called lossless forensics that naively measures the number of log entries reduced from original log. This is motivated by a conservative view that any loss of log events leads to loss in semantics. The second criteria is called causality-preserving forensics and is motivated by CPR. This metric counts the number of "distinct" information flows and check if edges that constitute each flow still remain in reduced log. The final criteria is called attack-preserving forensics. This criteria assumes that there is a ground truth about events that actually constitute an attack. Then, it checks whether those events have been removed by log reduction techniques

or not.

While all metrics proposed by Forensic validity work well for log reduction utility, our goal is to measure utility for timing information and we found none of these metrics to be applicable for our work.

1.3.5 SGX Programming

Our work on DOVE is related to prior efforts in running/partitioning/managing general purpose applications in SGX [73, 82, 83, 84, 93, 129, 130, 131, 132, 133, 134]. 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 *μArch* side channels) and (iv) whether the proposal requires a new custom programming language. We show a comparison along these axes in Table 1.1. 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 *μArch* 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 exposing the sensitive data.

The work most similar to our proposal are TrustJS [131] and SGXBigMatrix [93]. The former runs untrusted JavaScript code but assumes a weaker adversary that cannot monitor fine-grain application behavior over 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.

1.3.6 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 Instruction Set Architectures (ISAs). For example, application-centric works propose data-oblivious cryptography [89, 90], machine learning [75, 93], databases [94, 95, 96], memory and datastructures [73, 98], general purpose code [76, 91, 99, 134, 135], utilities [97] and floating point functions [48]. Many of these [73, 75, 76, 92, 93, 94, 95, 96, 134, 135] were designed for SGX-enabled applications to block the *μArch* side channels discussed in Section 1.2.3 and 1.2.5. Programming language, compiler and runtime works study how to write (e.g., [100, 101]) and compile (e.g., [102, 103, 109]) programs to

Table 1.1: Related works on application partitioning/management in SGX.

Name	Untrusted Apps?	Interpreted Code?	Blocks μarch Side Channels?	Supports existing languages?
Haven [82]	✗	✓	✗	✓
Graphene-SGX [83]	✗	✓	✗	✓
Scone [129]	✗	✓	✗	✓
Panoply [84]	✗	✗	✗	✓
Glamdring [130]	✗	✗	✗	✓
TrustJS [131]	✓	✓	✗	✓
ScriptShield [132]	✗	✓	✗	✓
Ryoan [133]	✓	✗	✗	✓
SGXBigMatrix [93]	✓	✓	✓	✗
ZeroTrace [73]	✗	✗	✓	N/A
Felsen et al. [134]	✓	✗	✓	N/A
DOVE [17]	✓	✓	✓	✓

software circuits. ISA abstractions study how to design interfaces usable by both software designers and hardware architects to uphold data-oblivious security guarantees [74]. These efforts are backed on the theory side by studies on how to run different algorithms and data structures in the circuit model [102, 104, 105, 106, 107, 108, 109, 110, 111].

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 3.3, 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 [74] (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.

1.3.7 Privacy-Preserving Genomics

Our case study for DOVE is based on the genomic dataset of honeybees collected from three different locations [136]. Genomics has been a promising test case for privacy-preserving application of SGX in prior work. These include at least the privacy-preserving computation of admixtures [137], Genome Wide Association Studies (GWAS) [138], and analysis of rare diseases (viz. Kawasaki disease) [139]. There is also an SGX-based study of privacy-preserving queries on genomic data [140]. Additionally, a survey discusses approaches to genomic privacy based on SGX, on cryptography, and on a hybrid of both [141].

CHAPTER 2: TERMITE ATTACKS

2.1 INTRODUCTION

This chapter introduces Termite attacks, a new class of side-channel attack that exploits implicit information contained in logs. We show how attackers can steal system secrets by using a rich existing body of work on side-channel analysis, in several disparate attack scenarios. This chapter starts with a reproduction of two prior coarse-grained side channel attacks. The first attack is a membership inference attack against MySQL databases by inferring communication details between Nginx and the MySQL process by return values of system calls made by both processes. Then, we port a classic keystroke detection attack [142] against the SSH process with a timing side-channel attack. Finally we demonstrate first fine-grained microarchitectural side-channel attack against system audit logs. The key idea for this fine-grained microarchitectural side channels is inspired by clock-edge attacks [66, 68]; although log events themselves correspond to coarse-grain events, the *order* in which log entries are written to the log can be a function of fine-grain timing behavior across concurrent processes. We characterize the extent to which this idea can be used to enable prior microarchitectural attacks such as NetSpectre [62] and remote memory deduplication attacks [143]. In particular, our NetSpectre proof-of-concept (PoC) attack shows that our timing side-channel attack is capabled of steal a secret bit in 470 seconds with 81.25% success rate when an adversary can craft 12 memory loads [144].

Chapter outline. Since our attack is based on a potentially-unfamiliar threat model, we explain the threat model of our work in Section 2.2. Then, Section 2.3 introduces two side-channel attacks to illustrate how to exploit coarse-grained information contained in audit logs. The first attack is a membership inference attack against a LEMP (**L**inux, **E**Nginx, **M**ySQL, **P**HP) webserver's MySQL database and second attack is a keystroke inference attack against a server when an administrator victim remotely connects to the server via SSH. Finally, we demonstrate how to create a fine-grained timing channel in Section 2.4.

2.2 THREAT MODEL

We wish to categorize the different types of attacks that arise from logs. Assume a system with processes running concurrently, at least one of which interacts with some secret data x. We define a system log L as a series of log entries, each of which is a tuple including various

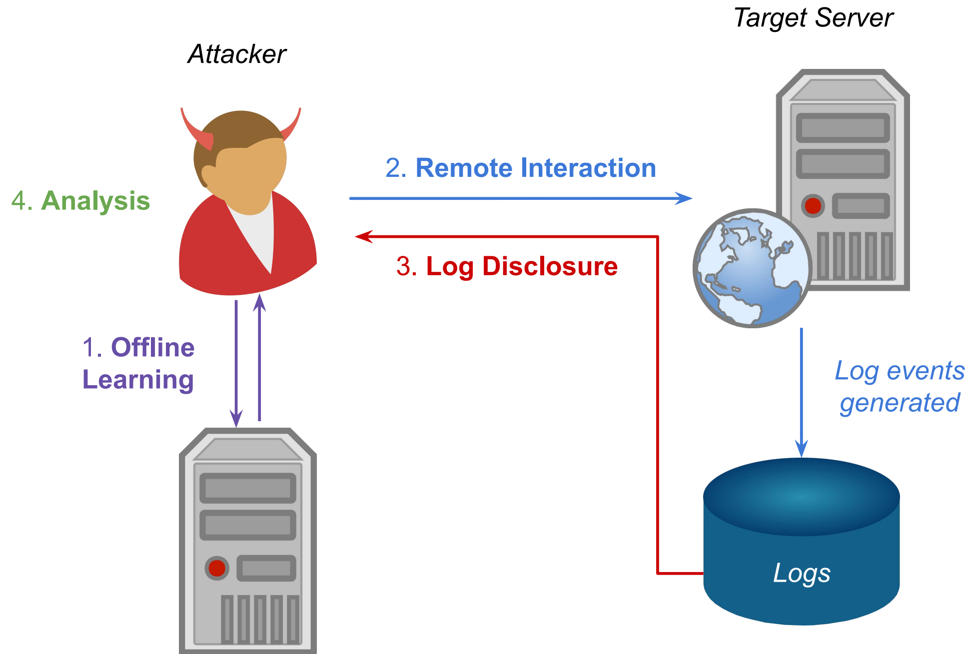


Figure 2.1: Workflow for a Termite attack.

pieces of metadata along side the log content. Exact metadata depends on the logger, but can include the process identifier and timestamp. There is a timer oracle that the logger queries to generate timestamps. The timestamps faithfully represent the order in which the timer oracle was queried, but events can later be written to the log in a different order. If log entries are written to disk out-of-order, we can use the timestamps to reorder them. Loggers track coarse-grained events, such that if two events sequentially happen in very short duration of time, they may share the same timestamp due to the coarse-grained timer. This means that it may be difficult to directly extract fine-grained information from log entries. As such, the model puts no requirements on log granularity.

When viewed in this fashion, logs share similar characteristics to program execution traces [74, 145]. Both are depictions of a program's behavior. If a program interacts with a secret, then information about the secret—or the secret itself—can be leaked by analyzing the output of the log or trace. In other words, logs are a *side channel*, just as execution traces are.

If we view logs like execution traces, the system's interactions with the secret x will project some information about x into the system log. The attacker's analysis of the system log leaks P, a set of possible values of secret x that can be inferred from L. The attacker's goal is to gather a sufficient number of logs from the system such that the range of P can be reduced: ideally for the attacker, $|P| = 1$. However, noise from system execution may cause outlier

measurements such that an intersection is infeasible in practice. Thus, we instead formulate this problem into a probability analysis, $Pr(\text{secret} = x \mid L) \geq \tau$: the attacker wishes to find the probability of secret being x for a given log L is greater than a threshold τ.

Attack scenario. We assume a system that runs a public-facing application of some sort as well as a system logger configured to record forensically-relevant system calls (Listing 1.1) such that we configure our logs to have comparable configurations to the audit logs generated by works described in Section 1.2.2. These logs are stored in the system for long periods of time (Section 1.2.1). The applications may have their own logger, but we will focus on the system logger. This application operates on secret data, and the attacker's goal is to gain access to the secret.

Generally, Termite attacks follow four phases, depicted in Figure 2.1. In the first phase, *offline learning*, the attacker locally initializes a server with similar configuration as the target and some value x' in place of the true secret. The attacker uses this local setup to measure and characterize the logs L' to develop a plan of attack for the target. This gives the attacker the insights necessary to find f and infer the true secret x. Prior literature in de-anonymization [146, 147] terms this "fingerprinting": comparing attributes of one object to those of an object the adversary knows.

After offline training, the attacker is able to connect to the server over its public interface during the *remote interaction* phase. Importantly, the server may or may not respond to the attacker's remote queries. For example, the server may display an identical error message on its public interface for any invalid input. However, the server will record log events that may contain additional information beyond its response.

At some point, remote interaction stops. Next, during *log disclosure*, the attacker receives the logs. Finally, the attacker proceeds to the *analysis* phase and can use the logs to attempt to infer the secret. The attacker uses their knowledge of application behavior to determine secret information from the disclosed logs.

2.2.1 Disclosure Assumptions

We assume that the log the attacker receives has been redacted according to some system policy to eliminate any explicit leakage of secrets not the log[2]. As such, we focus on implicit leakage, in which information about a secret is encoded in the sequence of log events. We assume that the secret is not directly disclosed to the attacker, *i.e.*, not directly written

[2]We compare log confidentiality techniques in Section 1.3

to somewhere accessible by an attacker. We further assume that the only information the attacker has available is the log and knowledge of the victim program and target platform.

We now discuss two possible forms of log disclosure.

Disclosure via privilege escalation. Since logs are typically stored in a protected directory (*e.g.*,, `/var/log/` for LAS), privilege escalation may be necessary to extract them. However, any privilege used to extract log data could be used to extract the secret directly, making log analysis unnecessary. Logs *are* useful when the secret is *ephemeral*, *i.e.*, only available during a process and not after. If this were not the case, then An ephemeral secret, on the other hand, is not disclosed with privilege escalation, because it no longer exists at the time of log disclosure.

For example, consider the case of a user inputting secret commands into an interpreter (*e.g.*,, [142, 148]). Even if an adversary were able to escalate privilege and access the user's system, the adversary would not be able to recover the commands inputted after the execution of the interpreter. However, the logs related to the user's session may contain implicit information about the commands.

A similar situation can arise, for example, when a *remote* resource is available during execution (*e.g.*,, [149]) but not after. In general, any temporarily-available secret information can be considered an ephemeral secret. We assume that if privilege escalation is used for log disclosure, the secret must be ephemeral to necessitate a Termite attack.

Disclosure via shared logs. An organization's log data can be used by other parties for scientific purposes. For example, enterprise logs [123, 150, 151] help guide academic research into systems architecture and security. Parties can share their logs among one another after applying obfuscation techniques (*e.g.*,, [10, 123, 146]), but this sharing can also act as log disclosure for a malicious party. If an attacker gains access to even obfuscated log data shared under such a model, they can still execute a Termite attack, without requiring privilege escalation on the target server.

2.2.2 Attacker Interaction

We define the types of interaction an attacker can perform prior to log disclosure. The most obvious is an the *passive* setting, in which the attacker cannot interfere with the normal operation of the system. Since log content is user-controlled, the attacker must use the offline learning phase to identify what relevant user data sequences in L look like. This in addition to the analysis required to reverse the log entry itself. The complement is the *active* attacker

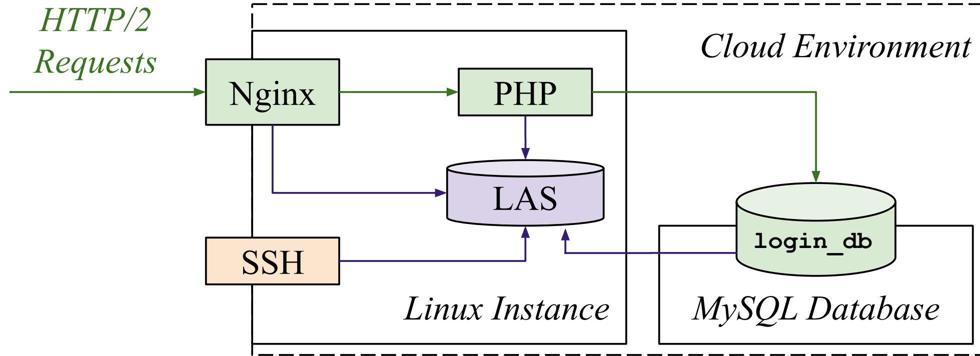


Figure 2.2: The architecture of our target system: a simple login application using the LEMP stack. The relevant system calls of Nginx, PHP, and SSH are logged using LAS.

setting. The attacker injects data into the system to influence what data is logged, in an attempt to add more information about the secret to the log.

Non-adaptive attack. Active Termite attacks are inherently similar to the non-adaptive adversary used in IND-CCA1 models [152]. Critically, the attacker only gets the logs after the attack, and cannot continue to interact with the server once its logs are disclosed. Unlike most attacks (*e.g.*,, [62]), the attacker gains no feedback from the log channel until after the attack is executed. This means the attacker cannot modify parameters while the attack is in progress. Known attacks do not immediately translate to the log setting; it is non-trivial to apply adaptive attacks to the non-adaptive model.

Process view. Because a system logger is operating on m concurrent processes, potentially from different users, it may include entries not relevant to the attacker. This can lead to log entry interleaving between processes, *i.e.*, for entries l_i, l_{i+1}, process identifiers $p_i \neq p_{i+1}$. This can result in noise that impedes the attacker, and, given sufficiently large m, slow down the system logger. However, this multi-process view can also provide additional information useful in deducing a secret shared by multiple applications.

2.2.3 Target System

To better describe Termite attacks, we develop a target system. Our attacks are general, and can be applied to any system logger running against a service. The example serves to build intuition about the impact of Termite attacks and show their practicality against realistic deployments.

Consider an Nginx HTTP/2 web server running on a Linux cloud instance. The server is configured to run PHP applications via its FastCGI process manager (PHP-FPM), which is a standard setup for using PHP on Nginx. Additionally, the server can connect to a remote MySQL database inside the cloud via PHP. This combination is commonly referred to as a "LEMP" stack. For system administration, the cloud instance is accessible over SSH. The cloud instance is configured to record forensically-relevant system calls for provenance auditing using LAS (Section 1.2.2) with rules such as the one in Listing 1.1. LAS monitors key system daemons, such as Nginx (`nginx`), PHP (`php-fpm7.2`), and SSH (`sshd`). A diagram of our target system can be found in Figure 2.2.

The web server implements a simple login service as a PHP application, with a username and password input field. When a user attempts to log in, the PHP application requests the password `hash` for a provided username from a database, using the SQL query in Listing 2.1. The resulting hash, if found, is a 60-byte value that is returned to the login application. If the hashed password matches the database hash, the login is successful. However, our application displays "Login failed." to the user either if the password does not match the hash (*i.e.*, is incorrect) or if no hash was returned (*i.e.*, the user is not found).

2.3 COARSE-GRAINED SIDE CHANNEL ATTACKS

Before we dive into a demonstration of fine-grained timing side channel attacks, we first present two coarse-grained side-channel attacks to explain how implicit information in logs can be leveraged to steal sensitive information. The first attack is membership inference attack that infers whether an entry is included in the MySQL database based on system calls made during communications between MySQL and Nginx. Then, our second attack demonstrates porting a classical keystroke timing attack [142] against SSH. This attack not only demonstrates a plausible timing side-channel attacks against audit logs, but also explains workflow of system audit logger so that readers can fully understand the sources of timing-related noises associated with logging. Then in Section 2.4, we show how we fully leverage such information to launch clock-edge timing attacks.

2.3.1 Attacking Log Content

We start by showing how log content can be used to extract secrets. We build an active membership inference attack against a database. In particular, we examine how LAS entries for socket interactions contain fields that leak the presence or absence of a value in a database.

```sql
SELECT hash FROM login_db.login_table WHERE username = ? LIMIT 1
```

Listing 2.1: SQL statement executed by our PHP login application. The `?` is replaced with a user-supplied value.

```
type=SYSCALL msg=audit(1651701011.121:54441): arch=c000003e syscall=45
↪  success=yes exit=87 a0=5 a1=7f8ce6a73067 a2=58 a3=40 items=0 ppid=3055
↪  pid=3056 auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33
↪  sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="php-fpm7.2"
↪  exe="/usr/sbin/php-fpm7.2"
```

Listing 2.2: Full LAS log entry for `recvfrom` when PHP receives a MySQL response that does not include a password hash.

The attacker's goal is to identify if the members of some set S are included in a column of a database table. In the context of our target system, S would be a set of usernames potentially in the login application's database. The attacker, in this case, is interested if the users of S are registered with our example service. While not traditionally considered a secret, the registration status of user is still potentially sensitive. For example, the service could be subversive (*e.g.*,, [153]) or taboo (*e.g.*,, [154]), and the attacker could use this knowledge against the user as extortion or blackmail.

A conventional attack against our target system would be to launch a credential stuffing attack [155], logging in with the usernames in S using the web interface (with random passwords) and reading back the server's response. The issue with this attack is the nature of the "Login failed." message, as described in Section 2.2.3. The attacker is unable to discern if the login failed due to the username not being in the login database, or if the username was in the database but the password was incorrect. Naive credential stuffing, therefore, fails to infer registration status.

Implementation. We now show how a Termite attack on log content successfully infers database membership.

Importantly, the SQL in Listing 2.1 will return the password hash *only if* the user is actually in the database. Thus, for an invalid user response r, and valid user response r', the lengths have relationship $|r| < |r'|$. Moreover, since the lengths of password hashes are fixed values [156], the difference between the lengths is $|r'| - |r| = l_h + l_m$, the length of the password hash l_h plus the length of any metadata l_m different between the responses.

Since the `php-fpm` daemon is monitored by LAS, the log will contain information about socket reads and writes, including those of its communication with the MySQL database.

```
type=SYSCALL msg=audit(1651701013.701:54492): arch=c000003e syscall=45
↪  success=yes exit=88 a0=5 a1=7f8ce6a73067 a2=58 a3=40 [snip]
↪  exe="/usr/sbin/php-fpm7.2"
type=SYSCALL msg=audit(1651701013.701:54493): arch=c000003e syscall=45
↪  success=yes exit=64 a0=5 a1=7f8ce6a73060 a2=5f a3=40 [snip]
↪  exe="/usr/sbin/php-fpm7.2"
```

Listing 2.3: Abridged LAS log entries for `recvfrom` when PHP receives a MySQL response that does include a password hash. The total number of bytes received is $88 + 64 = 152$.

LAS records return values of system calls in its `exit=` field; for the `recvfrom` system call, this is the number of bytes read from a socket for a response.

With the above in mind, membership in the database is associated with either a larger or a smaller response length; we can use this fact to infer it from LAS logs.

Offline learning. We first set up a local version of the login application and MySQL database, inserting some users from S and random hash values with a fixed length l_h into the database. We can then perform login actions against this local instance.

First, we send a value not in our local database. The associated LAS entry for the response r can be found in Listing 2.2. It is a `recvfrom` call (`syscall=45`) that reads 87 bytes (`exit=87`) from the socket associated with MySQL (`a0=5` in hex) into a buffer. This means an "empty" response $|r|$ is 87 bytes long.

We then send a value that is in the database. LAS records two entries for the response r', as seen in Listing 2.3, abridged for clarity. There are two `recvfrom` calls, one reading 88 bytes (`exit=87`), and the other reading 64 bytes (`exit=64`), for a total response length $|r'|$ of 152 bytes. We have two log entries because the first `recvfrom` call reads only up to its maximum buffer size of 88 bytes (`a2=58` in hex).

The difference in bytes between the two cases is $|r'| - |r| = 65$: 60 bytes of the found hash value (l_h), and 5 bytes of metadata (l_m). Thus, we now know that a value from S is not a member of the database when we see a response `recvfrom` of $|r| = 87$ bytes, and is a member when we see two response `recvfrom`s of a total of $|r'| = 152$ bytes.

Remote interaction. With this information in mind, we attempt login for all of the usernames $s \in S$, and recording a timestamp t_s when we do. This timestamp will be relevant in our analysis phase.

Log disclosure and analysis. After log disclosure occurs, we begin our analysis. We apply our knowledge from the local training phase to look for `recvfrom` calls for MySQL's responses to PHP: an 87 byte entry for a username s means that it is not in the database, while an 88 byte and 64 byte entry means that it is.

As discussed at the end of Section 2.2.1, the log entries of multiple processes could interleave, creating noise for our analysis. This is where our timestamp t_s for each username attempted s helps. Each log entry l_i has some timestamp t_i, and in LAS this is represented by the `msg=audit(...)` field, as seen in Listing 2.2 and 2.3. Additionally LAS entries have process identifier fields p_i, `exe=...`. We can then attempt to line up timestamps, searching for l_i such that $t_s \approxeq t_i$ and p_i is our target process, `php-fpm`.

2.3.2 Attacking Log Timestamps

Our next attack utilizes the timestamp information embedded in logs to passively reconstruct the keystrokes a user inputted into their SSH session. To best of our knowledge, this is the first keystroke timing attack using only LAS logs.

Our attacker wishes to extract secret information that users type while remotely logged in via SSH, such as passwords [142, 148, 157, 158]. We assume that forensically-relevant system calls are logged for SSH on the server. We additionally assume the users of the interest to the attacker have already performed their SSH session before logs are disclosed. In other words, user keystroke information is already in the log. The attacker wishes to use the timing between keystrokes to reconstruct the original typed information.

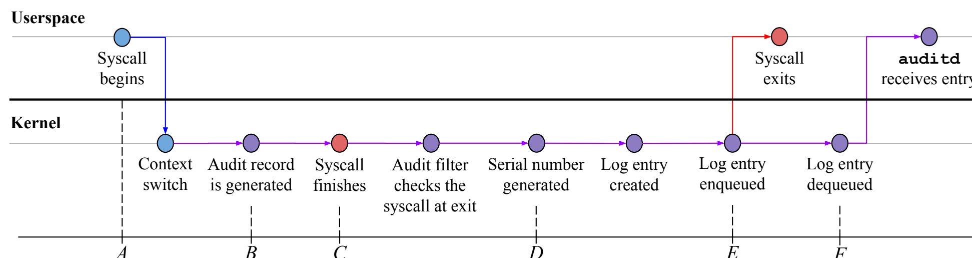


Figure 2.3: A workflow of system call logging, separated between userspace and the kernel. The timeline shows relative times associated with selected event to show delays associated with system call logging. For example, the delay and noise associated with timestamp generation is $B - A$, those of serial number is $D - A$, and those of log entry orders in a file $F - A$.

```
type=SYSCALL msg=audit(1651937384.644:1075): arch=c000003e syscall=0 success=yes
↪  exit=36 a0=3 a1=7ffc41df88e0 a2=4000 a3=7ffc41dfc850 [snip]
↪  exe="/usr/sbin/sshd"
type=SYSCALL msg=audit(1651937384.644:1076): arch=c000003e syscall=1 success=yes
↪  exit=1 a0=9 a1=56064ad23020 a2=1 a3=7ffc41dfc850 [snip] exe="/usr/sbin/sshd"
type=SYSCALL msg=audit(1651937384.644:1077): arch=c000003e syscall=0 success=yes
↪  exit=1 a0=b a1=7ffc41df87b0 a2=4000 a3=7ffc41dfc850 [snip]
↪  exe="/usr/sbin/sshd"
type=SYSCALL msg=audit(1651937384.644:1078): arch=c000003e syscall=1 success=yes
↪  exit=36 a0=3 a1=56064ad0f370 a2=24 a3=7ffc41dfc850 [snip]
↪  exe="/usr/sbin/sshd"
```

Listing 2.4: Abridged LAS log entries generated for a keypress of `v` over SSH.

Timestamps in LAS. A keystroke attack [142, 148, 157, 158] is fundamentally a timing side-channel attack. As such, we must understand uncertainties associated with timing measurements to ensure our channel is sufficient before we execute it. Our prior attack did not require such an analysis, as it was not dependent on the timing of events to execute.

The lifecycle of an LAS log entry is shown in Figure 2.3. We focus our analysis on LAS's properties. However, the logging mechanism of the LAS is also found in other system audit loggers [28, 159, 160, 161], so we believe our analysis on this logger can be generalized.

After a system call begins in userspace (time A in Figure 2.3), it makes a context switch into the kernel. An audit record is is then generated (time B in Figure 2.3). This record contains basic information like type of system call, values of system call registers, and a timestamp is polled from kernel's timekeeping function called `ktime_get_coarse_real_ts64()`. This function provides a coarse-grained UNIX timestamp ($\approx$ 3 milliseconds per tick) for performance reasons.

The event we wish to analyze happens at A, when the system call begins; however, we do not see a timestamp at B. Thus, For the purposes of building a timing attack, we must consider the latency between the two: $B - A$. We analyze the function trace of the system call entrypoint in the Linux kernel, `do_syscall_64()`. The first function entered is `syscall_trace_enter()`, which leads to the function `__audit_syscall_entry()`, in which the audit record and timestamp are generated. `__audit_syscall_entry()` is the first operation that occurs once the kernel is entered, so latency between A and B is minimal, and will not significantly impact our timing attack.

Implementation. Armed with our understanding of LAS timing, we can execute our keystroke attack against our target system, as described in Section 2.2.3. The attacker is

specifically interested in the keystrokes a user inputs into their Bash shell when logged in over SSH. Prior work has shown how operating system interfaces [158], network traffic [142] or hardware side channels [148, 157, 162] can be used to extract keystrokes; we adapt it, concretely, to the LAS setting.

The secret x in question is a user's typed content into a terminal, such as a password. This secret is ephemeral, and lasts only as long as the user's session is active, making it an attractive target for a Termite attack.

We must first understand how the log L is generated to extract the secret x. In an SSH session, a remote user uses an SSH client to interact with an SSH server, which mirrors input and commands from the client into a shell. When a user inputs a key, four key steps are occurring:

1. The SSH server daemon receives a keystroke from the user's SSH client as a network packet.

2. The SSH server sends the user's keystroke to the Bash shell interpreter that represents the user's remote login.

3. Bash, to provide the user with feedback, prints back the the keystroke as a character to its standard output. This is immediately read back by the SSH server.

4. The SSH server provides a response to the user, sending this printed character back to the SSH client.

All four of these steps require interaction with the kernel in the form of system calls, which means each step generates an entry: l_1, l_2, l_3, l_4, with associated timestamps t_1, t_2, t_3, t_4. All four steps are executed every time a key is pressed; thus, a subsequent keystroke may generate log entries l'_1, l'_2, l'_3, l'_4 with timestamps t'_1, t'_2, t'_3, t'_4. It follows then that the inter-keystroke timing is $t'_1 - t_1$, which can then be used to recover keystrokes [142]. The fingerprint is the sequence l_1, l_2, l_3, l_4 that indicates a user typed a keystroke. We then use a timestamp from this fingerprint, t_1, to establish inter-keystroke timing. We choose t_1 because the original attack of [142] uses network timing to recover keystrokes, and t_1 is related to network processing; in practice, any of the entries' timestamps can be chosen. In any case, inter-keystroke timing recovers x, as required.

Offline learning. We first identify what LAS logs when a key is pressed over SSH. Listing 2.4 contains the four entries generated in sequence for a user keystroke. Each of the four lines of Listing 2.4 corresponds to an entry l_1, l_2, l_3, l_4 from the workflow above. Step

(1) is represented by l_1, which is a `read` (`syscall`=0) of 36-byte client packet (`exit`=36) from the SSH server's socket (`a0`=3 in hex). Step (2) is represented by l_2, which is a `write` (`syscall`=1) of the keystroke itself (`exit`=1) to Bash's standard input (`a0`=9 in hex). Step (3) is represented by l_3, which is a `read` back of the now-printed keystroke (`exit`=1) from Bash's standard output (`a0`=`b` in hex). Finally, Step (4) is represented by l_4, which sends 36 bytes (`exit`=36) representing the printed character back to the client over the network.

Each entry also contains a Unix timestamp in its `msg`=`audit(...)` field: t_1, t_2, t_3, t_4. As discussed in Section 2.3.2, the LAS timestamp is relatively course grained – on the order of milliseconds – but is actually sufficient granularity to launch the machine learning-style analyses found in prior work [142, 157, 158, 162]. Intuitively, even advanced typists (*e.g.*,, 120 words per minute) need a few milliseconds to input a character, so LAS will record each key press.

Remote interaction. As a passive attack, we do not interact with the server as an attacker. To show the power of our attack, we instead design a experiment based on the classic experiments from [142]: measuring the inter-keystroke timing between different pairs of key presses, with the hope of distinguishing each key pair from another based on the timing differences.

We perform two experiments. In the first, we measure the inter-keystroke timing of two specific pairs – `v-o` and `v-b` – as done in [142]. Typically, `v-o` is typed with two different fingers, leading to a smaller inter-keystroke timing than that of `v-b`. We perform the experiment by logging into our server via SSH (as a user) and typing the character pair `v-o` 75 times. Then, we type the pair `v-b` 75 times. We will attempt to use our understanding of the log to differentiate between the two.

Our second experiment attempts to generalize the first, assessing the timing properties of several key pairs instead of just two. As a user, we once again login via SSH to our server. This time, we type the sentence "the quick brown fox jumped over the lazy dog" 32 times. This sentence is composed of 40 unique key pairs, each of which we hope to distinguish based on log information.

Log disclosure and analysis. After we retrieve the LAS logs from the server, we must go through the process of extracting timestamps from known keystroke log entries, as learned from our local instance. We show the results of our first experiment in Figure 2.4. The histograms show that LAS detects a clear difference in inter-keystroke timing between the two pairs. This reproduces the original result of Song et al. [142], showing the potential of LAS for this style of attack. Moreover, our attack did not require us to be collecting any

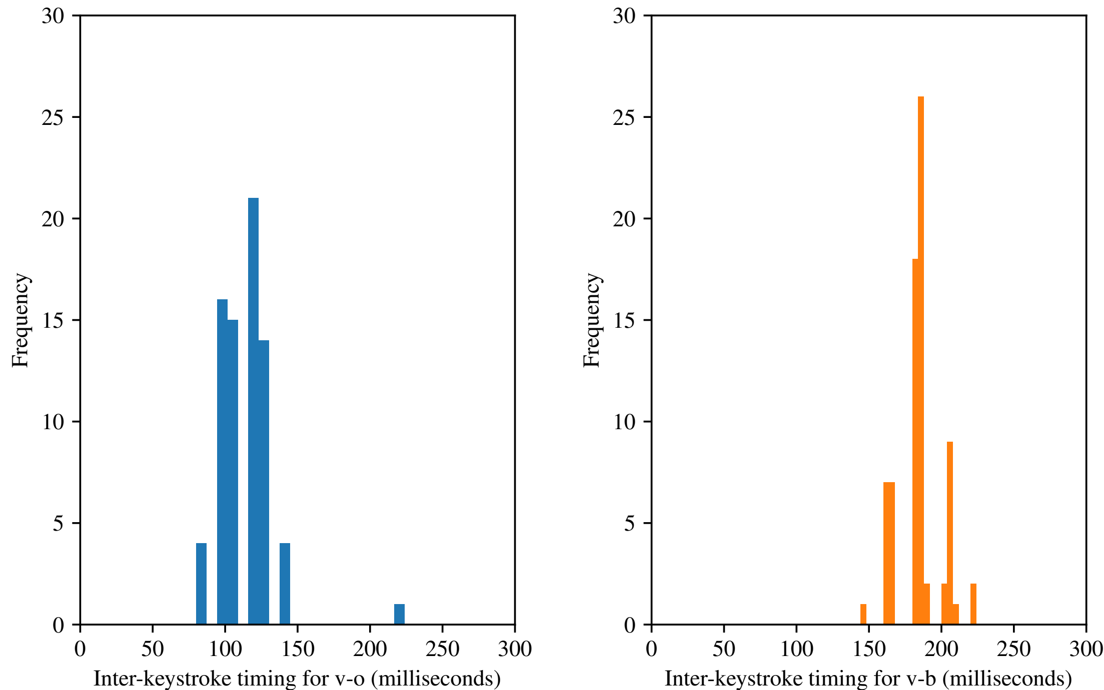


Figure 2.4: Inter-keystroke timing for two keypairs, captured via LAS log entries for SSH. Each keypair was typed 75 times.

data while the user is online; we can still extract keystroke timing well after the user has completed their session.

The results of our second experiment can be found in Figure 2.5. Each curve in the figure represents an estimated normal distribution over one of the 40 unique key pairs of the sentence. We were able to record each key press in the LAS log with zero false positives and zero false negatives. The distributions overlap, but are similar in form to the distributions collected and analyzed by Song et al. [142]; this means that the techniques from that work can be applied to LAS logs to reconstruct key press information.

We can consider an optimization to our attack when attempting to extract password information only. In the naive case, we reconstruct all keystrokes and look for password entries that way: for example, a password is likely entered after inferring the keystrokes `s-u-<RETURN>`. This may be too much work if the log is sufficiently long.

Instead, we can use the following insight to focus our analysis: passwords are typically not echoed back to a user, as a security measure. In practical terms, this means Bash writes nothing back to its standard output when receiving a password input. In turn, Step (3) in our interaction workflow does not happen, which means l_3 in Listing 2.4 is not found

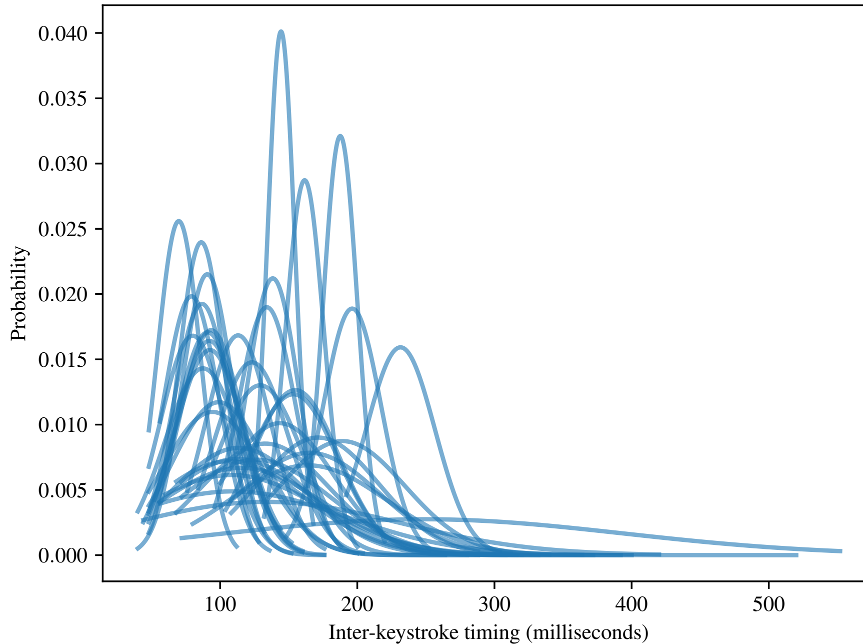


Figure 2.5: Estimated normal distributions for the unique key pairs of "the quick brown fox jumped over the lazy dog".

in a password keystroke's log entry sequence.[3] Thus, we know that if we see a sequence that looks like l_1, l_2, l_4, the user is likely entering a password; we can utilize our analysis techniques against the timestamps of those associated log entries.

2.4 FINE-GRAINED SIDE CHANNEL ATTACKS AGAINST ORDERING OF LOG ENTRIES

System auditing rarely requires that events have fine-grained timestamps. As such, logs produced by audit frameworks like LAS have coarse temporal resolution. Secrets can still be extracted from this coarse resolution, as the attack in Section 2.3.2 shows. Listing 2.2 shows an example of LAS records; note that the Unix epoch timestamps are only in millisecond granularity.

An intriguing question is if logs with such coarse timing can nevertheless be used to leak secrets through finer-grained side channels. In this section we show how a Termite attack can overcome the limits of coarse timestamps to construct a timing side channel based on microsecond or even nanosecond granularity. The demonstration is based on microarchitectural

[3]In our testing, Step (4) still occurs, regardless if there is a character to send or not.

side channels, and it is inspired by clock-edge attacks [66, 68]. The key idea in the attack is to exploit the ordering between log events created by two processes where the ordering leaks relative timing information between two events. This allows the attack to circumvent the use of coarse timestamps in system auditing. We show how this works against LAS.

We focus on the active but non-adaptive scenario described in Section 2.2 for this attack: the attacker can remotely interact with an application via the network and can influence what is written by the system logger. This interaction model is common for remote timing side-channel attacks [163]. Note that since the logs created by LAS are system-level logs, the logs may contain information about any application running on the system, not just the application of interest.

2.4.1 Attack Overview

We aim to build a timing side-channel attack to steal a secret bit. The attacker must first ensure that a victim runs a data-dependent computation that leaks timing information associated with the secret. For the remainder of this section, we call a code segment that satisfies this a *secret-dependent section*. An example of a secret-dependent section would be a branch on a secret flag bit (such as `if(secret)`). The secret flag determines whether the branch is taken or not, and the computation time that corresponds to processing the branch thus depends on the secret.

Prerequisites. Once such a vulnerability is found, the next step for the adversary is to find methods to exploit latency differences caused by the secret-dependent section to infer the secret bit. To successfully launch the clock-edge attack with audit logs, the attacker must satisfy certain prerequisites. First, the attacker requires a pair of processes: a *reference* process x, and a *target* process y that runs code in a secret-dependent section. The reference process has some characteristics that the attacker knows (or is able to learn offline), and serves as the reference clock edge (or source of events that are generated at secret-independent times). Second, the attacker should be able to actively control the concurrency of two events. Ideally, attacker would remotely interact with the victim to ensure that the two events start simultaneously. These match the prerequisites of a previous clock-edge attack [68].

The logging setting, however, introduces an additional prerequisite. We assume that each process should has a system call event that both happens after the secret-dependent section and is logged by LAS. Preferably, a system call happens right after the secret-dependent section to precisely measure the latency of the section, but any system call that

deterministically occurs after the secret-dependent region will suffice.

2.4.2 Clock-Edge Attacks against Logs

Our clock-edge attack aims to learn the system secret based on variations in system log event orderings. There are two ways to infer ordering of system call events based on LAS logs. The first way is to use the actual line orderings of two log event entry strings from the log file to compare the ordering. The userspace daemon `auditd` simply writes each event to an entry as it receives audit entries from the Netlink on a FIFO basis. Thus, if two log entries are created at very similar times in the kernel, then whichever entry that is enqueued faster will be ordered first. From Figure 2.3, the latency associated with this procedure is from time A to time E.

However, there is an alternative way to find ordering of two events. Serial numbers for log entries are implemented as incremental monotonic counters (like TCP sequence numbers); these numbers may also provide information on ordering of two events. The latency associated with this procedure is from time A to time D in Figure 2.3; we can eliminate uncertainties of further log processing in kernel after time D. Nonetheless, note that this procedure occurs after the end of a system call execution at time C. Thus, our measurements are inevitably affected by the behaviors of the system call being logged.

Prior work [68] performed a detailed analysis on effects of the jitter associated with concurrency and processing time of requests. They concluded that this jitter can be reduced with repeated measurements, assuming the measurements are normally distributed. To supplement their analysis for our setting, we characterize the jitter associated with logging, and measured the difference between the execution of a Linux system call and LAS receiving the system call. From Figure 2.3, this is the time from time A to time D. We extended LAS to include a cycle count (`rdtsc`) when an system call entry is processed. Then, we executed system calls in userspace, logging cycle counts before their execution, and noted the difference between userspace calls and kernelspace log entries. We specifically selected system call parameters with error conditions, such as incorrect file descriptors, to limit noise due to processing in the kernel. For example, we measured the `close()` system call on an invalid file descriptor, and found a cycle count difference on average of only $\bar{x} = 6420$ cycles ($s = 728$).

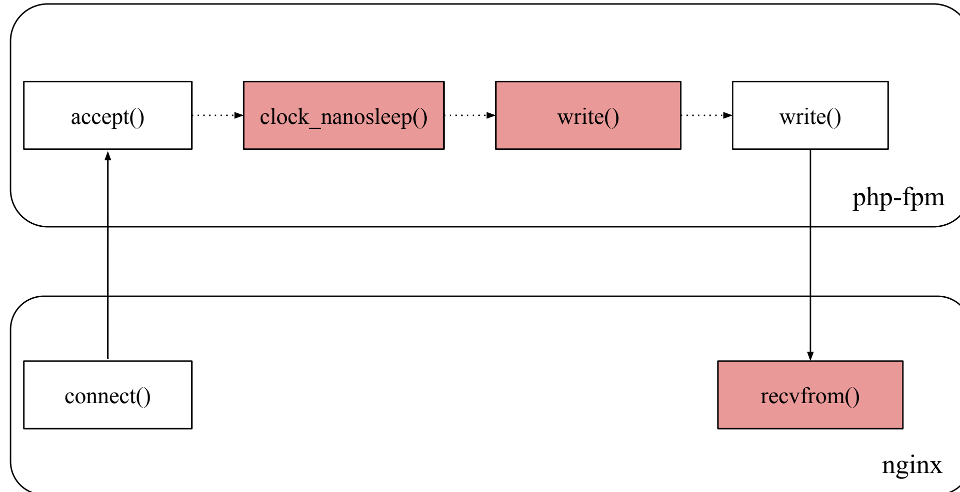


Figure 2.6: A workflow of system calls that are called during a request to the PHP endpoint. System calls in red are those used for our attack. Solid arrows indicate two system calls are directly related, while dotted arrows indicate they are not related but do happen in sequential order.

2.4.3 Implementation

As described in Section 2.2.3, the victim runs a LEMP webserver, where Nginx uses PHP-FPM for handling requests to PHP endpoints. Adversaries can remotely interact with the webserver over these endpoints. Also, major browsers only allow HTTP/2 protocols over TLS [68], so our victim webserver accepts secure connections via HTTPS.

The webserver is composed of two PHP endpoints where one endpoint sleeps for 100,000 nanoseconds (using PHP's `time_nanosleep()`) where as another endpoint sleeps for 100,000 + Δ nanoseconds. The different duration of the sleep simulates the different computation time of the aforementioned secret-dependent sections. The adversary's goal is to send a pair of concurrent requests to victim's two endpoints and make a conclusion on whether the Δ is equal to zero. The smaller the Δ, the finer-grain the side channel that we create. For each request pair, one request is sent to the 100,000 nanoseconds endpoint, and another is sent to the remaining endpoint. In particular, our implementation is analogous to that of the prior work [68] against HTTP/2.

Our evaluation machine runs a Intel Core i7-10750H CPU at 2.60GHz and has 16 GB of RAM. Note that this processor has 6 cores and 12 threads; this multi-core setup works to our advantage, as we are able to leverage concurrency [68] to execute our attack.

Offline learning. Before launching the attack, the adversary must investigate which system calls could fulfill the prerequisites that we previously mentioned. A simplified workflow of system calls that happen during request handling by our webserver is described in Figure 2.6. After Nginx receives a request and processes its header, Nginx establishes a connection with PHP-FPM for further request processing to a PHP endpoint. Then, PHP-FPM runs a `clock_nanosleep()` system call to sleep the execution with nanosecond granularity. This is the first system call event that is closest to the secret-dependent section as the system call itself is secret-dependent with secret being the duration of sleep. While `clock_nanosleep()` is not part of forensically-relevant system calls that we defined in Section 2.2, measurements from this function can provide a baseline of our timing side channel since the log entry would be created right after the secret-dependent section. In other words, the latency between actual event and serial number generation is minimal. This latency is expressed as time between C to D in Figure 2.3.

A real-world system is unlikely to be recording `clock_nanosleep()` events, though. When auditing the forensically-relevant system calls that occur after the secret-dependent section, there are several other system call events that attacker can exploit, but we primarily focus on two system calls closest to the secret-dependent section. The first event is `write()`. After PHP-FPM finishes sleeping, it performs a `write()` to a `stderr` stream.

Then, PHP-FPM returns a response back to Nginx via writing to a file descriptor socket. Nginx receives the data through socket with a `recvfrom()` call. This is the second forensic event that we are interested in. When assessing distance from the secret-dependent section, `clock_nanosleep()` is closest, `write()` is next and `recvfrom()` is furthest among the three.

The attacker must be able to adjust the delay between requests d_y such that two requests are processed concurrently. Prior work [68] has shown that an effective way of doing this against PHP endpoints is to vary the number of request parameters: the more request parameters, the longer processing takes. We sent two requests at our PHP endpoints, setting $\Delta = 0$, and modified the request parameters until both requests concurrently executed. When learning this parameter for each of three system call cases, we discovered that each case required a different number of parameters. For our machine, attacking against `clock_nanosleep()` at PHP required 105 parameters, `write()` at PHP required 55 parameters, and `recvfrom()` at Nginx required 65 parameters.

Remote interaction. Our remote interaction mimics our offline learning setup. As mentioned previously, there are two endpoints in our webserver and for each request pair, we send each request to each endpoint. The attacker varies request parameters to ensure concurrent processing of the requests.

We forked the Python client of [68] that sends concurrent request pairs, but changed its socket options. Instead of using `TCP_CORK` to ensure two requests being sent concurrently with Nagle's Algorithm, we chose to utilize `TCP_QUICKACK` for performance, following a suggestion from John Nagle [164]. [68] also showed how jitter associated with network communication does not affect measurement, so we conducted our experimentation in a `localhost` setting for simplicity.

Log disclosure and analysis. We now evaluate this timing side channel created by logging of concurrent system events in two aspects. First, we evaluate how fine-grained the channel can be to reliably distinguish timing differences between two processes. The finer the timing difference, the higher the probability existing remote timing attacks will be successful with our channel. Second, we evaluate the effects of aforementioned sources of latency/noise in our attack by quantifying the number of measurements required to reliably transmit secret bits. The practicality of our attack is proportional to the number of log entries (the number of requests made by adversary) it requires: the fewer log entries required, the better.

The resultant logs can be parsed into a series of binary Boolean vectors with size of n. Each vector represents a binomial distribution of whether the request of longer sleep is logged later than that of shorter sleep. If two requests sleep the same amount of time (i.e. $\Delta = 0$), the distribution will be uniform whereas if one request is supposed to take longer than the other (i.e. $\Delta \neq 0$), then the distribution will not be uniform. We consider the uniform distribution to be the null hypothesis; we distinguish the timing difference Δ with 95% confidence based on whether the distribution is uniform or not.

Table 2.1 shows the result of this experimentation for various different timing differences that we are able to distinguish with 95% confidence. Note that all numbers represent the number of request *pairs* – the actual number of requests are two times what the number suggests. When the timing difference required more than 50,000 requests pairs to be distinguished, we concluded the timing difference to be effectively indistinguishable. Sending 50,000 request pairs and receiving responses took approximately 51 seconds when each request pair had 0.1 ms of delay. We observe that as the system call event moves further from the secret-dependent section, the required number of request pairs increases. This is expected as the jitter increases when system call events move further from the secret-dependent section. More measurements are required to normalize the distribution of jitter.

Overall, when considering an ideal case with `clock_nanosleep()`, the timing channel we create can distinguish timing differences in the 100s of nanoseconds.

Table 2.1: The number of minimum request pairs required in order to distinguish the timing difference between two requests with 95% confidence. If the number of request-pairs is higher than 50,000, then we mark it as -. We conduct measurements against three system calls where `clock_nanosleep(2)` simulates secret dependent section, then `write(2)` to `stderr` stream, and Nginx receives computation results from PHP with `recvfrom(2)`.

Audited System Call Events	100ns	200ns	500ns	1μs	2μs	5μs	10μs
`clock_nanosleep(2)` at PHP-FPM	-	50,000	30,000	10,000	3,500	750	100
`write(2)` at PHP-FPM	-	-	-	-	12,000	4,000	200
`recvfrom(2)` at Nginx	-	-	-	-	50,000	25,000	200

2.4.4 Case Study: NetSpectre

The channel generated by the remaining (forensically-relevant) system calls, while more coarse-grained, is still fine enough to mount a μArch side-channel attack. We demonstrate this with our proof-of-concept (PoC) NetSpectre attack with multiple loads [144]. We implement our NetSpectre PoC on the same setup as Section 2.4.3, except we use FastCGI endpoints written in C++ instead of PHP endpoints. The server is consisted of two endpoints, one reference endpoint that runs 16,500 iterations of empty loops and another endpoint with leak gadget that is followed with transmit gadget. We use these empty loops as the reference endpoint for simplicity, but, as mentioned in Section 2.4.3, the attacker can vary the number of request parameters to the PHP endpoint to manipulate the endpoint execution time.

Each endpoint makes a system call `close(2)` where reference endpoint makes it after the loop and endpoint with gadgets makes the call after the transmit gadget. Note that this system call is one of aforementioned forensically-relevant system calls from Section 1.2.2. We choose cache-induced gadgets for this PoC as we leverage a recent Spectre attack with multiple loads [144]. Thus, our gadgets are consisted of multiple memory block accesses, and we show how the number of accesses impact error rates of NetSpectre attack by testing against gadgets with the number of memory blocks (N) = {1, 4, 8, 12}.

Each experiment is consisted of 20,000 batches where each batch begins with five training requests-pairs to mistrain branch predictor associated with the leak gadget, and one test request pair to measure latency due to cache hits or misses after running the transmit gadget. Thus each experiment is consisted of 120,000 request pairs, resulting in 240,000 log entries associated with the experiment. Sending these request pairs and receiving responses took approximately 470 seconds when each request pair had 0.5 ms of delay, and each log file after the experiment had size of around 110 MB.

We ran 20 experiments for each N with 10 rounds against cases when secret bit is 0 and

remaining rounds against cases when the bit is 1. Each N is associated with twenty log files as a result. We used first four log files (two when secret bit is 0, the remaining when the bit is 1) as our offline learning to find the threshold of distinguishing whether the ordering of log entries follows the uniform distribution or not. Then this threshold was used against remaining 16 log files to infer whether the secret bit was 0 or 1.

Under this setup, our PoC with $N = 12$ had error rate of 18.75% (3/16), and for $N = 8$ we had error rate of 25% (4/16). For $N = 4$ and $N = 1$, the error rate was around 50% such that our attack was not able to distinguish whether secret bit is 0 or 1. Nonetheless, we show that with our ostensibly coarse-grained log-based channel, we are able to extract fine-grained information from μArch state. To our knowledge, this is the first attack that successfully implements a μArch side-channel attack through system audit logs.

As an aside, we believe that our PoC shows that our timing side channel is fine enough to conduct other attacks, such as the recent memory deduplication attack [143], where the Δ of interest is in range of twenty microseconds. Furthermore, the authors of [143] conduct an attack like that of Timeless Timing attack [68] in their work, so we believe reproducing their work in our setting is highly plausible.

2.4.5 Limitations

Generally speaking, our threat model focuses on non-adaptive attacks, so a large number of remote timing attacks cannot be trivially ported to the audit logs setting. Additionally, our attack places an emphasis on attacker interaction by its nature: the attacker generates logs through server interaction, and extracts data from the generated logs. A more sophisticated analysis could elide this requirement, and consistently work against on passively-collected logs. Building such an analysis is an interesting direction for future work.

We leverage concurrency to potentially extract μArch-level information from system logs. While the attack introduces a novel threat vector, it has certain limitations derived from its properties. As a concurrency-based attack, the attack requires training to understand the baseline features of the system. However, as a log attack, we are in the non-adaptive setting, and therefore this training must be performed on an attacker-controlled system, instead of on the victim server. Thus, if an adversary were unable to perform baseline training offline, this attack would be infeasible as presented.

2.5 CONCLUSION

We present Termite attacks, a new class of attack that reconstruct secret information from implicit log information. Termites are non-adaptive, remote attacks that rely on system loggers such as LAS to uncover secrets through data dependencies. We show how different log content, timestamps, and ordering can be exploited to construct different types of attacks. These attack parameters are general, and thus future research must be done to identify barriers to prevent logs from being attacked by Termites. In following chapters, we propose two countermeasure schemes to address aforementioned threats against logging.

CHAPTER 3: DOVE: DATA-OBLIVIOUS VIRTUAL ENVIRONMENT

3.1 INTRODUCTION

In this chapter, we demonstrate how to systematically decouple sensitive information from logs. This work [17] is originally designed for 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. Trusted Execution Environments (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.

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., [52, 53, 79, 81]). Yet, existing software-based techniques to block these channels—coming from a rich line of research in data-oblivious/constant-time programming [76, 89, 91, 99]—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 [89, 90], require the use of custom domain-specific languages [93, 101], or only apply to close-to-metal compiled languages [76, 91]. 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 [69]. Subtle issues in any of this code create security holes.

In this chapter, we introduce a project that extends 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 design "Data-Oblivious Transcript (DOT)", a log that serves as an intermediate representation of original high-level language script. This DOT is transcribed by DOVE *frontend*, which runs the target computation on pseudonyms (NOT on secret data) in the chosen high-level language, like R or Python. Note that this log 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. Thus, DOT is only composed

of data-oblivious primitives and pseudonyms such that it contains no sensitive information as a result. Then, the DOT is evaluated by the *backend* data-obliviously, on sensitive data that backend received via a secure channel.

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 [165], 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 [136].

3.2 THREAT MODEL

In this chapter, 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 1.2.5), 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 1.2.3), given the powerful SGX adversary described in Section 1.2.5. This is formalized in our security analysis (Section 3.6).

```r
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)
```

Figure 3.1: R code snippet. `geno` is a sensitive diploid dataset.

Security non-goals. We do not defend against hardware attacks such as power analysis [166], EM emissions [167], compromised manufacturing (*e.g.*, hardware trojans [168]), or denial of service attacks. Also, our current implementation does not have mechanisms to mitigate speculative execution attacks [169] beyond default SGX protections (e.g., flushing branch predictor state on context switches [170]). If additional protection is needed, our backend (an SGX enclave) can be re-compiled with a software-level protection such as speculative load hardening [171].

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., [102]) 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 3.6 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.

3.3 ATTACK EXAMPLES

A major problem this chapter 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 [82, 83]). We demonstrate subtle $\mu Arch$ side-channel attack vectors that come up in this approach, using the code snippet in Figure 3.1 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 3.7.2. 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.

3.3.1 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 1.2.4). 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 3.7.2 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 3.1 follows guidelines for achieving data-obliviousness* (Section 1.2.6), 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.

3.3.2 Logical Operators

We start with Line 1 in Figure 3.1, specifically the logical `&` operations performed between the masks. At the level of R code, these look like safe data-oblivious operations (Section 1.2.6). 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 3.2a. When R interprets `&`, it invokes the C routine given in Figure 3.2a. 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 3.2b, which is the assembly for Lines 1 to 2 in Figure 3.2a, 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.

Table 3.1 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 Table 3.1. 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) [172]. On average, it took $\mu_{00} = 73.9$ million cycles ($\sigma_{00} = 441\text{K}$) for `(0 & 0)`, but it took $\mu_{10} = 75.2$ million cycles ($\sigma_{10} = 416\text{K}$) 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 `|` operators. Even binary comparison operators such as `==` and `!=` have similar issues. For example, R's implementation of both these operators uses branches at

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

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

```
; 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: ...
```

(b) The Intel-syntax x86-64 assembly for Lines 1 and 2 of the C code in Figure 3.2a, lightly edited for clarity.

Figure 3.2: The R interpreter implementation of the & operator.

Table 3.1: 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

the R level to first check if either operand is `NA`.

3.3.3 Functions

Aside from R-level primitive operators, R also has a large library of functions written in either R or C. In Figure 3.1, 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 [173]. 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`.

3.3.4 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 [46, 47]. 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.

3.3.5 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 [174], and is composed of hundreds of API functions and other features, implemented in a combination of R, C and Fortran [69].[4] 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., [48, 49, 74, 99]).

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 [93].

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

[4] 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 chapter.

with which they are familiar, and have strong confidence there are no latent side channels.

3.4 DESIGN

We now describe the Data-Oblivious Virtual Environment (DOVE). This begins with a design overview and summary of design benefits (Sections 3.4.1, 3.4.2). Section 3.4.3 discusses the Data-Oblivious Transcript (DOT), which serves as the link between high-level programming and data-oblivious execution. Section 3.4.4 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 3.4.5 describes the DOVE backend, an SGX enclave that converts the DOT operations on pseudonyms to data-oblivious computation on the actual sensitive data.

3.4.1 Design Overview

As stated in the threat model (Section 3.2), 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 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 3.3. 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 3.4.4 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 section, 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 *µArch* 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 3.4.4, certain constructs such as loops with sensitive loop bounds are disallowed.

3.4.2 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

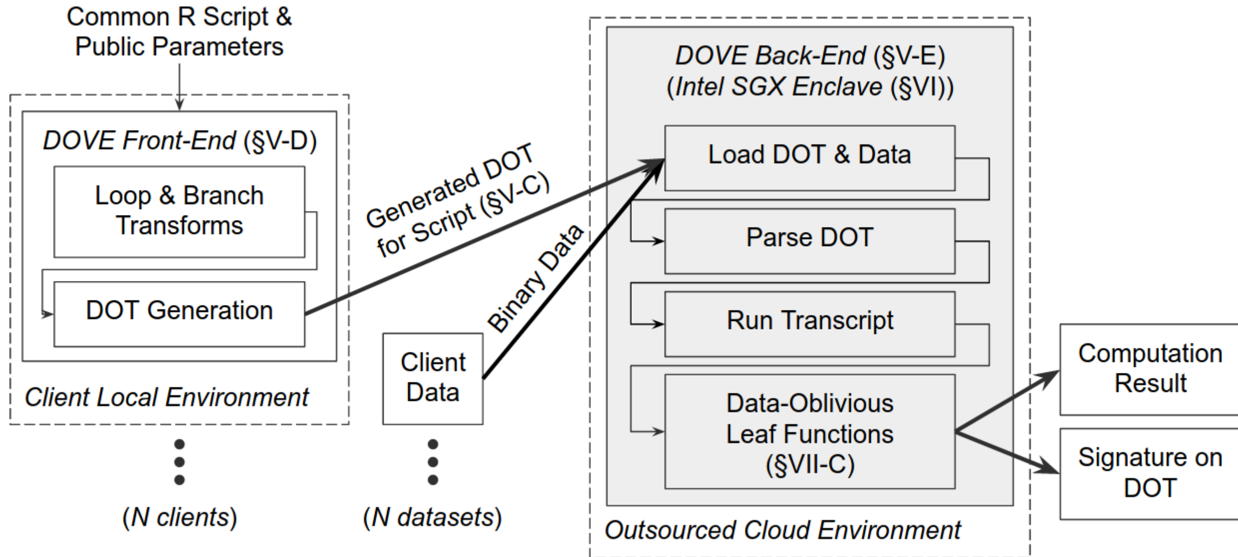


Figure 3.3: High-level overview of DOVE. Bold-face arrows between nodes represent communication over (mutually-authenticated) TLS, while thinner ones are intra-process communication within a component. Shading indicates the location of our trusted computing base (TCB).

C/C++ codebase featuring 7,001 lines of code, 4,295 of which consist of a previously-vetted, external data-oblivious fixed point library [48]. Importantly, the R stack (with its almost 1 million lines of code [69]) 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 [99, 102]) 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

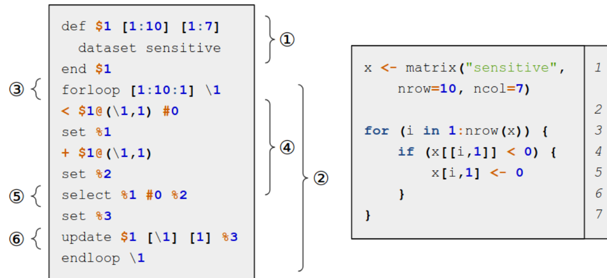


Figure 3.4: A DOT (left) and its associated R program (right). The matrix `x` corresponds to the pseudonym $1 in the DOT, and the loop index `i` with `\1`. ① corresponds to line 1 of the program, ② the `for` loop on line 3, ③ the `if` statement on line 4, and ⑥ the assignment in line 5. Intermediate values are stored in variables marked with `%`, and constants are declared using `#`.

backend security, the backend can be patched locally without necessarily making a change to the frontend.

3.4.3 Data-Oblivious Transcript (DOT)

The Data-Oblivious Transcript, or DOT, forms the core of the DOVE architecture, bridging an input program written in a 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 3.4.5).

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 [91], 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.[5] For example, to convert an if-else style conditional to the PC model, a compiler (or similar) needs to convert

[5]We note that our current implementation implements data memory-trace obliviousness [175] 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 [73, 76].

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 3.5). 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 3.4.4.

We now discuss DOT semantics in more detail, using Figure 3.4 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 DOVE paper's extended version [176].

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 3.4, 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 3.4 shows the definition of such a matrix, with $m = 10, n = 7$. Matrices are indicated with `$`, scalars with `%`.

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 Table 3.2. 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 3.2 and 3.6). 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 [177, 178] 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 [74].
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 3.4.5, 3.5), 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 [91]).

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 3.4 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 3.4. This predicated execution model, similar to that used in prior work [76, 99], 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 1.2.6).

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.

3.4.4 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

Table 3.2: 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 "Sup." (Sup. stands for Supplemental) are implemented using operations in "DOT/-Core" and exposed to the user as library functions. Safe functions require a data-oblivious implementation in the backend as they may receive pseudonyms as operands. Unsafe functions do not require a data-oblivious implementation, but can only take non-pseudonyms (non-sensitive) data as operands.

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

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 1.2.4 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.

Table 3.2 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 "Sup." 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 "Sup." 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 3.4.3.

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 complexity of the loop. Instead, the frontend automatically transforms such bounded loops to use the `forloop` DOT construct. As discussed in Section 3.4.3, 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 [101, 102].

Predicated conditionals. The frontend must translate conventional if-then-else structures into the predicated execution model supported by the DOT (Section 3.4.3). For this, we implement an if-conversion transformation that is similar to prior works [76, 99]: 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 3.4.3) in the DOT. The branch in Figure 3.4 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 3.7.2 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 3.5).

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.

3.4.5 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 3.4.3), and includes implementations of all operations in the DOT semantics, i.e, those listed under Safe and Unsafe "DOT/Core" in Table 3.2. 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 3.3.

The scope of DOVE is to block all non-speculative $\mu Arch$ side channels (Section 3.2). For this purpose, the backend provides a data-oblivious implementation for operations in Safe "DOT/Core" of Table 3.2. To implement these operations, we rely on a subset of the x86-64 ISA and well-established coding practices [99] for implementing constant-time/data-oblivious functions (see Section 3.6 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 [48].

Importantly, what hardware operations (e.g., machine instructions) open $\mu Arch$ side channels depends on the $\mu Arch$. For example, two x86-64 processors can implement `cmov` differently: one in a safe way, one in an unsafe way (e.g., by microcoding the `cmov` into a branch plus a move [74]). 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 vul-

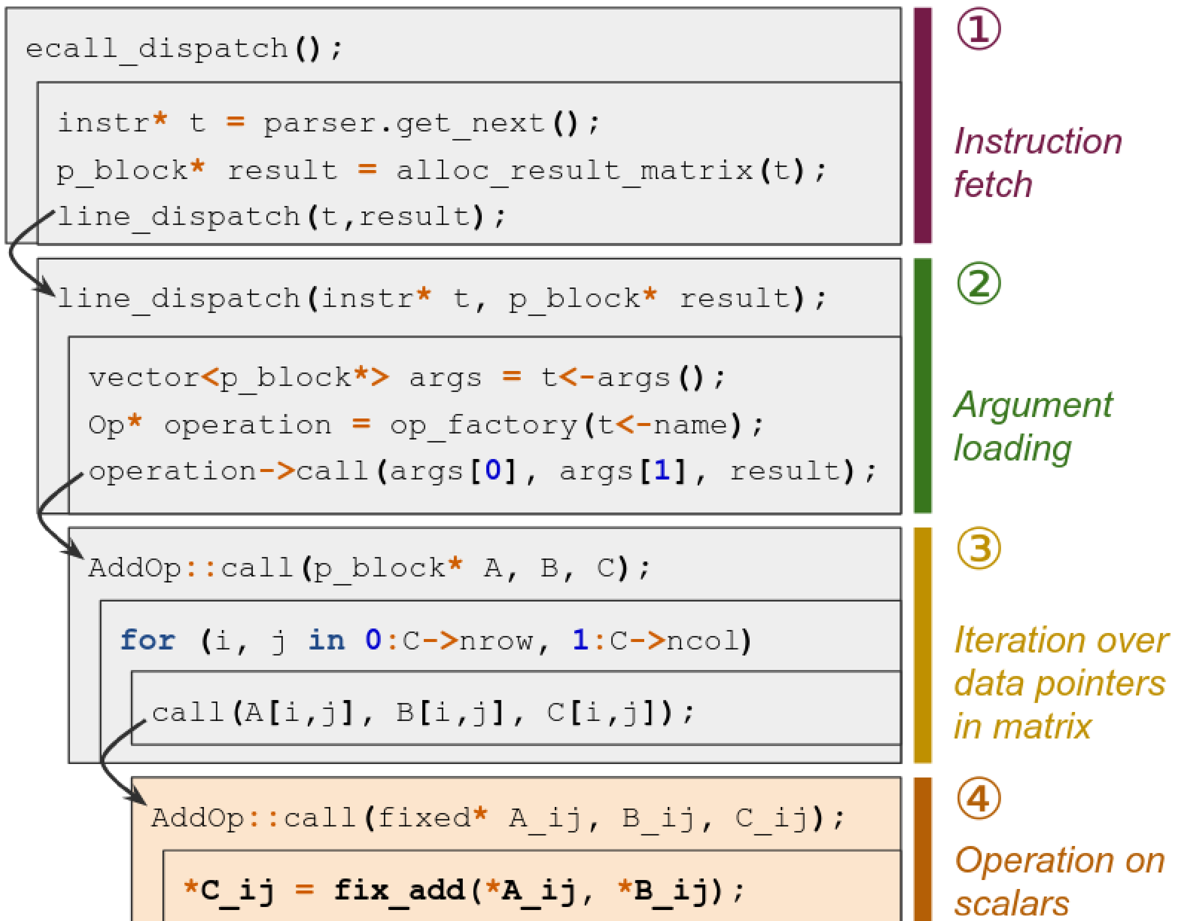


Figure 3.5: A simplified graph representing the flow of `ecall_dispatch()` for the addition instruction +. The final, bold-face portion is the only block that dereferences sensitive data.

nerability is found in `cmov`, the backend can opt to implement the DOT `select` operation using a CSWAP (bitwise operations) or other constructs.

3.5 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 3.6). 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 1.2.5). 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 3.4) 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 3.4.4) and that each DOT operation type checks (Section 3.4.3). The latter ensures that pseudonyms cannot be downgraded to non-pseudonyms (Rule 1, Section 3.4.3) and that pseudonyms are not passed as operands to Unsafe operations (Rule 3, Section 3.4.3).

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 3.5 depicts a simplified call graph for the addition instruction (+). Running different instructions entails a similar call graph.

During instruction fetch (① in Figure 3.5), 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 3.5, 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 3.5. 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.

3.6 SECURITY EVALUATION

We first present our formal definition of security under the SGX adversary first discussed in Section 3.2. We then argue that this security definition holds given our DOVE backend implementation, for the setup and instruction execution phases.

3.6.1 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 3.2, 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 3.3, 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.

Log 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.

$$\mu Arch[V(T_S, D)] = \mu Arch[V(T_S, D')] \tag{3.1}$$

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.

3.6.2 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 3.7.

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 3.7.

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 [48]. 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 3.5). Finally, Section 3.6.3 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 [179]. 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 3.4.4).

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 3.5, 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 [48]). 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 3.4.3, 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 3.5), which ensures that Rules 1 and 3 are enforced (Section 3.4.3). 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 3.5: 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.

3.6.3 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 3.4.3. 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 1.2.6).

We first analyze the leaf function instructions that take sensitive data as operands. These instructions are shown in Table 3.3. We determined this set by inspecting instruction depen-

dencies in the `objdump` disassembly. All but one of the opcodes in Table 3.3 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 [48]. 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 [76]. 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).[6] 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 Table 3.3, 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 [180] 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 [91]. 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).

3.7 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 3.6 for experimental evaluation.

[6] 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.

Table 3.3: 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`		

The first echoes prior work [93] by coding and analyzing applications of the PageRank algorithm [181]. The second examines a suite of programs [165] for genomic analysis and a case study using it for the analysis of honeybee genomes [136]. 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 3.6.1: $Q[V(T_S, D)] = Q[R(S, D)]$ 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 [182]. The libFTFP instructions' relative performance overhead is measured against its Streaming SIMD Extensions (SSE) counterpart; the overhead varies depending on the instruction, ranging from 1.2× for `neg`

Table 3.4: 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

(operand negation) to 208× for `exp` (exponential function evaluation) [48].

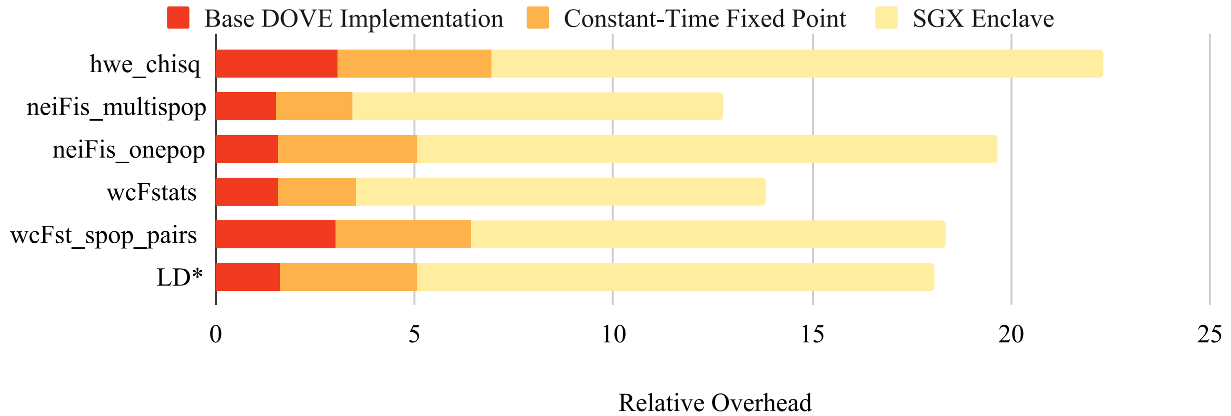
3.7.1 PageRank

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 [93]. A large proportion of this algorithm is composed of matrix multiplications, which other works choose as primary performance benchmarks [76, 183].

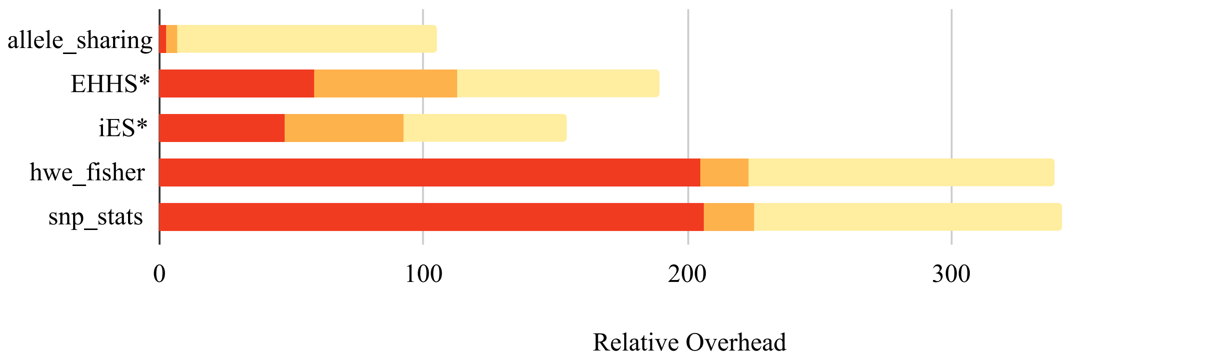
We use the Wikipedia vote network (WikiVote) [184] and Astro-Physics collaboration network (ca-AstroPh) [185] datasets from the Stanford Network Analysis Project [186] for this case study. Both datasets are converted to adjacency matrices, where WikiVote has 7,115 nodes (≈ 405 MB) and ca-AstroPh has 18,772 nodes (≈ 2.8 GB). Table 3.4 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 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 3.3).

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.



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



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

Figure 3.6: Performance evaluation results for the evaluation programs. Each stacked bar represents a measurement for each program. Each stack represents relative overhead of DOVE against vanilla R caused by generic data-oblivious computation, libFTFP and SGX from left to right. Programs marked with * run on reduced dataset due to machine limitations.

3.7.2 Genomic Analysis

To further validate DOVE, we work with an application that performs a controlled study on honeybee genomic data [136]. The study relies on R code drawn from a set of 13 genetics research programs [165] that implement important statistical measurements found in the literature [187, 188, 189, 190], totaling 478 lines of R code [165]. These programs, in addition to the coding of PageRank, constitute a practical illustration of the expressiveness of DOVE. The paper's extended version [176] 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 front-end 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., [73], to avoid such worst-case work.

We utilize the dataset from the honeybee study [136] to perform performance benchmarking. We run the full 2,808,570 $\times$ 60 ($\approx$ 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 [138, 139, 140].

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 3.6.[7] 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

[7] Numbers for each program's absolute runtime are given in the paper's extended version [176].

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 (Table 3.2), to provide a fair comparison and to reduce TCB size. When called, the frontend will convert the call into a series of equivalent DOT operations.

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

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

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× relative to vanilla R for these programs. One program, `allele_sharing` (in Figure 3.6b), 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.

3.8 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 a log called 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.

CHAPTER 4: TIMELESS LOGS

4.1 INTRODUCTION

In the previous chapter, we have investigated how data-oblivious computation and TEE can be applied to provide security against microarchitectural side-channel attacks. However, such countermeasures may not be fully applicable for logging systems. In particular, what if one cannot create a DOT without sensitive data? We assume in Section 3.2 that several attributes of DOT to be public including ordering of operations. However, we showed in Section 2.4 that an adversary can potentially exploit ordering of logs to launch timing side-channel attacks. Furthermore, DOVE is designed against hostile operating systems such that large performance loss due to deployment of TEE and of data-oblivious computation are justified, whereas most log-confidentiality problems worry about an adversary who temporarily breaks into the honest system. In such cases, an alternative solution to close timing side channels from logs is to post-process the logs.

In this chapter, we provide a study of log timestamps in terms of security and utility. If we take the same amount of security guarantee DOVE provides (*i.e.* offering completely data-oblivious solution for logs against timing side channels), then we must remove any timing information from logs. These not only include timestamps (Section 2.3.2) but also ordering of log entries (Section 2.4). Unfortunately, removal of such information would result in a huge utility loss. But how big is it? We investigate and quantify this loss based on a utility function called *reachability* (also called connectivity in other works) that is introduced with several log reduction techniques. Their objective is to remove semantically redundant log entries. When viewing a log as a table, each log entry composes a row of the table whereas each timestamp comprises a column of the table. Therefore, we view these works as works that remove rows from a table whereas our works try to remove values of a column (or a whole column in an extreme case).

We first show that there is a large increase in false positives in terms of reachability when we remove all timing information for logs. Then, we assess security-utility tradeoffs associated with the amount of timing information in logs by proposing batched logs as an alternative countermeasure. Our case study on a public dataset shows that when batches are sufficiently small (batch size of around 1,000 entries or smaller), there is only 0.05 percent increase in the number of false positive reachable nodes.

4.2 THREAT MODEL

This chapter mainly focuses on proposing countermeasures against timing side-channel attacks introduced in Section 2.4. The adversary that we model in this chapter is mostly consistent with that of Chapter 2. However, because our work focuses on mitigating timing side channels, side-channel attacks against log contents mentioned in Section 2.3.1 is out of scope in this chapter. We assume that adversary only uses i) timestamps and/or ii) ordering of log entries to learn about secrets in this chapter. It might be possible that an adversary may learn about timing information of log entries based on their content. For example, an application can make a system call where either its return value or its parameter leaks information about timing of such a call. We do not consider such threats in this work, but detection and mitigation of such leaks would be interesting future work.

This work proposes a post-processing mechanism as a solution. Thus, we assume that the adversary can only access post-processed logs when he or she breaks into the system to steal the logs. Preventing adversary from accessing raw log files can be achieved by deploying orthogonal solutions like TEEs [5, 116]. Finding and mitigating possible side-channel attacks against post-processing computation is also a future work.

4.3 REACHABILITY

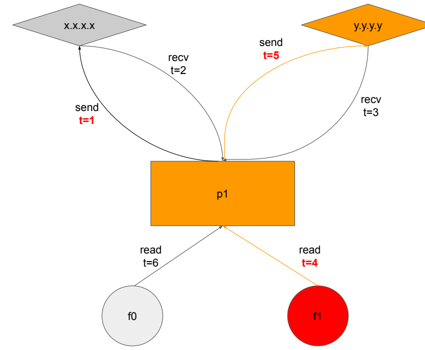
As mentioned in Section 1.2.2, modern log analysis focuses on converting logs into provenance graphs and performing graph traversal based algorithms such as forward and backward tracking from chosen edge/node. Thus, for this chapter we choose *reachability* as a key utility function to examine the utility of our countermeasure schemes.

We define a node and/or an edge of a provenance graph to be *forward-reachable* from a designated node (*e.g.,* hostile remote address or compromised process) or edge (*e.g.,* alerted event) if: 1) there is a path from the chosen node/edge to destination node/edge and 2) timestamps for each edge along the path is sorted in increasing order so that the search algorithm do not travel back in time as it traverses the graph. In other words, the traversal can only occur if prior edge's timestamp value is before than that of next edge. For example, Figure 4.1a is a short log segment where a process `p1` communicates with two remote hosts `x.x.x.x` and `y.y.y.y`. The process also reads two files `f0` and `f1` where `f1` contains sensitive information.

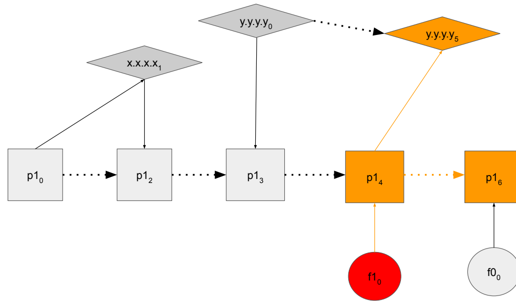
This segment can be represented as a provenance graph shown in Figure 4.1b. System admin might be interested in finding information flow from `f1` to find potential information leakage, so they may run a forward-reachability function from `f1` on the graph. This function

```
t=1, send() to x.x.x.x by pid=1
t=2, recv() from x.x.x.x by pid=1
t=3, recv() from y.y.y.y by pid=1
t=4, read() from f1 by pid=1
t=5, send() to y.y.y.y by pid=1
t=6, read() from f0 by pid=1
```

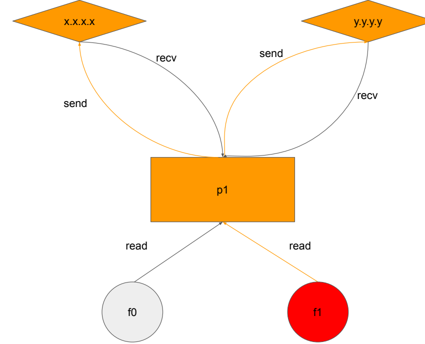
(a) A simplified, timestamped log segment that captures two remote connections (from address `x.x.x.x` and `y.y.y.y`) made with Process 1 (`p1`). `p1` also reads file 0 (`f0`) and file 1 (`f1`).



(b) Provenance graph representation of Figure 4.1a with timestamps.



(c) Versioned counterpart of Figure 4.1b. Dotted arrows represent version changes for each node and solid arrows represent events. On this graph, edge timestamps are dropped whereas node versions (in subscripts) provide timestamp information.



(d) Timeless counterpart of Figure 4.1b. Node `x.x.x.x` and edge from `p1` to the remote host are now reachable. These are false positive reachable node and edge.

Figure 4.1: A simplified timestamped log segment and its provenance graph representations. Orange nodes and edges are forward-reachable nodes/edges from `f1`.

would return edges and nodes colored orange as reachable nodes and edges. Note that an edge from `p1` to `x.x.x.x` and a node `x.x.x.x` are NOT reachable from `f1` since the read from `p1` to `x.x.x.x` happened before the read from `p1` to `f1`.

We can also define *backward-reachable* in the same manner by traversing from destination node/edge to starting node/edge while requiring timestamps of edges on the path to be decreasing order. For remainder of the chapter, we mainly focus on forward-reachability as our main utility function and we simply call it reachability.

4.3.1 Log Reduction Techniques and Reachability

Log reduction techniques attempt to reduce volume of logs by finding and pruning "redundant" events. Whether events are redundant largely depends on the type of utility functions that examine log semantics. Log reduction techniques are interested in removing events such that timing information of removed events are also reduced from logs. Thus we strictly benefit from log reduction methods since reduction of a log entry means reduction of its timing information as well. Reduced logs, unlike approximated logs, mentioned in Section 1.3.4 retain full utility. Our attack prototypes demonstrated in Section 2.4 requires a large number of measurements to infer a single bit. These repeated attack requests from an adversary will be semantically "redundant" for auditing point of view so that log reduction works can greatly deter such attacks by removing entries that logged such requests.

As mentioned in Section 1.3.4, there are multiple prior works that focus on reachability for as their main utility function, but we focus on Dependency Preserving Reduction (DPR) [127] in this chapter as this work aims to preserve global reachability of all nodes while constructing a provenance graph. DPR is motivated by a prior work Causality Preserving Reduction [126] (CPR), but reduces edges more aggressively than CPR does. CPR only examines reachability based on each node's perspective by counting incoming and outcoming edges. This will ensure preservation of each information flow, but there can exist multiple flows between source and destination nodes. DPR, on the other hand, tries to make a global assessment of the whole provenance graph to reduce aforementioned redundant flows. While this may impact other utility functions that require preservation of every information flow, in terms of reachability, there is no effect on reducing such redundant edges.

However, making such global analysis can be very computationally expensive as the analysis may require a complete graph traversal. Thus, DPR leverages versioning to solve this problem where the algorithm converts a standard provenance graph into a versioned graph. Figure 4.1c is versioned graph representation of Figure 4.1b where dotted arrows indicate version changes caused by an influx of information from another node. While in this example their is no log reduction benefit by converting the simple graph from Figure 4.1b to Figure 4.1c as there is no redundant edge to remove, authors of DPR demonstrated effectiveness of log reduction for DPR against large log datasets.

In this DPR representation, edge timestamps are not necessarily needed, since each node version retains information about time written in subscripts. Furthermore, when solely considering reachability, the versioned graph does not need to retain any timestamps.[8] The

[8] Under original paper's definition of forward reachability, removing all timestamps from the graph may create false positive reachable nodes from the source since their reachability function also requires starting time as one of inputs, but we do not consider such requirement for our utility analysis.

only timing information needed by this graph to run reachability analysis is the direction of dotted arrows that indicate version changes of nodes. Applying DPR and storing logs as forms of versioned graphs without timestamps can greatly reduce timing information by not only removing redundant edges, but also potentially remove all timestamps from logs.

However, naively applying DPR or other log reduction techniques and expecting to close all timing side channels unfortunately is not plausible. Binary information about direction of dotted edges reveals the relative ordering of solid edges (*i.e.*, log events) that cause versioning changes so that such timing information can be exploited against clock-edge attacks mentioned in Section 2.4. These reductions are not fundamentally designed for protecting against side channel attacks, so an adversary can easily evade these countermeasures if he or she knows about them ahead. Even if we can remove all timestamps with DPR, we still need to retain information about version changes for each node, and an adversary can exploit such timing information to launch clock-edge attacks. In addition, some of log reduction techniques [128, 191] are solely interested in reducing file I/O related syscalls, because those comprise the majority of logged events. Unfortunately, our clock-edge attack prototypes demonstrated in Section 2.4 exploit network-related events by making concurrent request to an Nginx server. Removing these events are not considered in most log reduction works.

Nevertheless, reachability as utility function can help assess the utility of post-processed logs for any countermeasure. In next section, we explain how redaction of all timestamps and randomization of log entry ordering results in utility loss.

4.4 TIMELESS LOGS

In Chapter 3, we introduce data-obliviousness as key concept to close side-channel attacks. In this section, we examine post-processed logs where we remove all timestamps and shuffle log entries such that we remove any timing information. The result is "time-oblivious"; data-oblivious in terms of timing information. We call these logs Timeless Logs. We assess the utility loss associated with such post-processing with the reachability utility function that we introduced in Section 4.3.

4.4.1 Reachability and Timestamps

Figure 4.1d is a time-oblivious provenance graph constructed from the log segment shown in Figure 4.1a, illustrating the effect of removing timing information from logs. When forward tracking from `f1` to find reachable nodes and edges, there is a false positive reachable node `x.x.x.x` and a false positive edge `send()` from `p1` to `x.x.x.x`. As shown in Figure 4.1b,

this node and edge are not supposed to be reachable because the `send()` event happened at $t = 1$ whereas a previous event `read()` happened at $t = 4$. Without timestamps, however, such restriction no longer exists. This simple example shows what happens to logs when timing information is removed. While removal of timestamps do not cause an increase in false negative reachable nodes/edges, but it can cause an increase in false positive reachable nodes/edges. From a system admin's perspective, the increase in the false positives would make forward tracking tedious as he or she must examine and filter extra information which are not related with Point of Interest (POI) events. Thus, it is crucial that our countermeasure does not produce a large number of false positive reachable nodes.

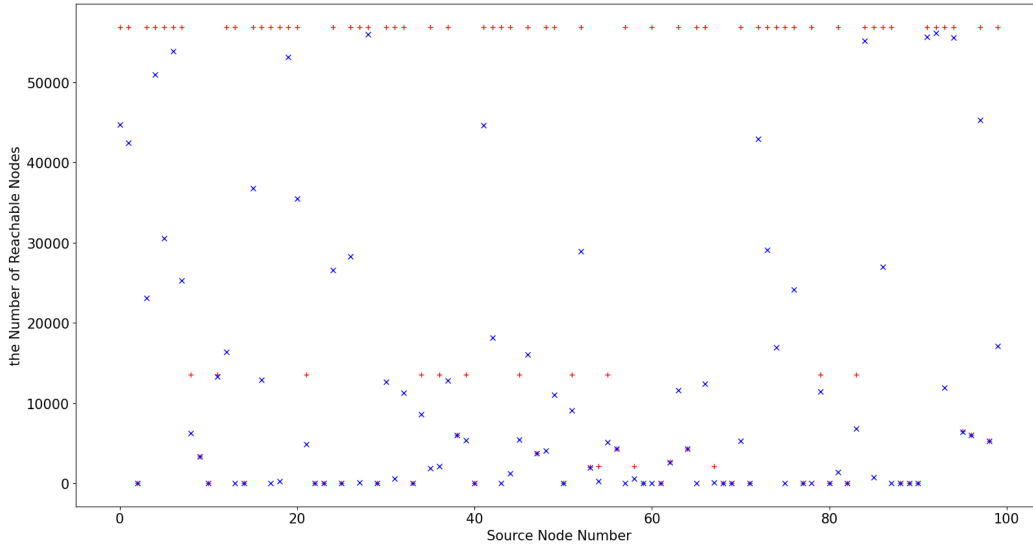
4.4.2 Implementation

For this analysis, we use DARPA Transparent Computing (TC) Program datasets for our evaluation. This DARPA Program intended to provide a transparent view of complex systems and provide rich provenance data for researchers. This data has been used in prior audit log research [128, 192, 193]. In particular, we choose a subset of datasets from host Theia and Cadet where we use first three binary files of each dataset. This is approximately 2.98 million events collected over 4.5 hours for Theia and 5.15 million events collected over 48.7 hours for Cadet. Our experimentation machine configurations are consistent with that of Section 2.4.3. We first the parse original DARPA Engagement data with a parser written in Java. This parser has been used in prior works [192] for converting datasets stored in binaries into LAS format. Then our analysis tool written in Python builds a provenance graph from the datasets in memory using the `NetworkX` graph library [194]. Like previous works, this graph is a `MultiDiGraph`, a directed graph with self loops and parallel edges where nodes correspond to: 1) processes, 2) files or 3) network sockets, and edges correspond to log entries audited by a system logger.

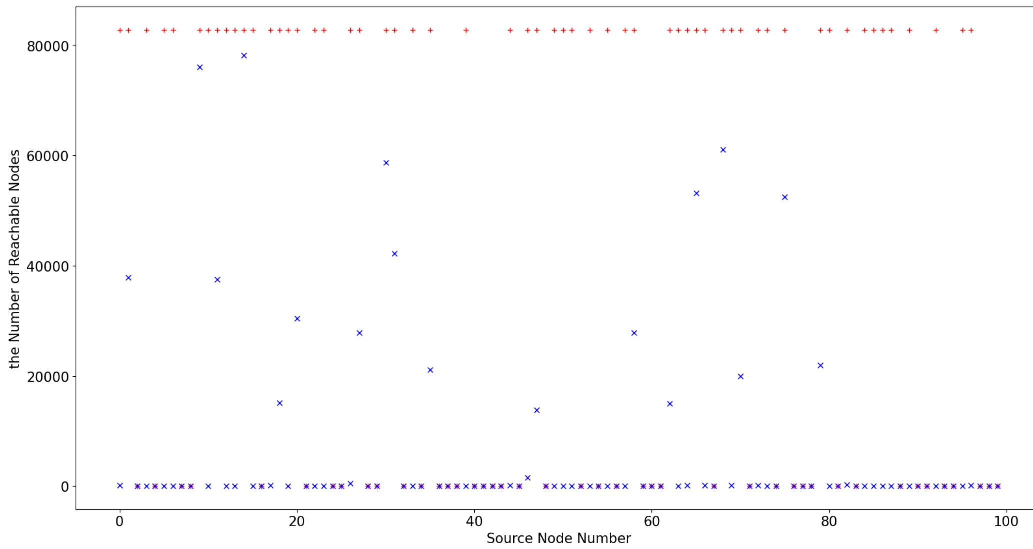
4.4.3 Utility Evaluation

To compare results of Timeless Logs against timestamped logs, our tool performs reachability analysis on timestamped logs by first converting the logs into a versioned graph like DPR [127]. For reachability analysis, we choose 100 random source nodes for the analysis from each dataset.

Figure 4.2a is a scatterplot that shows the number of reachable nodes for the 100 source nodes from dataset Theia. The red dots are measurements for source nodes from Timeless Logs and blue dots are ones from timestamped logs. For this scatterplot, We identify 100



(a) The number of reachable nodes from 100 random source nodes for timestamped logs and Timeless Logs for Theia.



(b) The number of reachable nodes from 100 random source nodes for timestamped logs and Timeless Logs for Cadet.

Figure 4.2: Scatterplots that show the number of reachable nodes from 100 random source nodes for timestamped logs and Timeless Logs. Red dots are measurements for Timeless Logs and blue dots are those of timestamped logs. The distance between red and blue dot for each source node illustrates the number of false positive reachable nodes created by removing timing information from logs.

source nodes with a number from 1 to 100 and x-axis show such representation. The y-axis shows the number of reachable nodes from each source node. The vertical distance between two dots for the same source is the number of false positive reachable nodes for the source node. As we previously explained, the removal of timing information fortunately does not create any false negatives, but does create false positives. For Timeless Logs of Theia, on average there is an increase of 36,600 percent false positive reachable nodes over true positive nodes from these 100 source nodes. Furthermore, note that for Timeless Logs, a lot of source nodes share the same number of reachable nodes. This phenomenon shows how removal of timing information degraded the utility as most randomly-picked source nodes gained no reachability information. Figure 4.2b is a scatterplot for dataset Trace and we observe the same trends where on average there is an increase of 283,000 percent of false positive reachable nodes over true positive nodes from these hundred source nodes. For dataset Cadet, the result can be interpreted worse than that of Theia where for a majority of source nodes, the number of true positive reachable nodes is only itself whereas the numbers of reachable nodes for those of Timeless Logs are more than 80,000.

Thus, complete time-obliviousness does not provide plausible utility as shown with our reachability analysis. In next section, we show an alternative approach we call Batched Timeless Logs, that does not achieve complete time-obliviousness, but provides improved security guarantees compared to that of timestamped logs while maintaining good utility compared to that of Timeless Logs.

4.5 BATCHED TIMELESS LOGS

In last section we showed that a complete Timeless log does not retain much utility, but if we cannot assure complete time-obliviousness against current threat model, then cannot completely mitigate threats from the adversary. Thus, we instead propose a countermeasure that does not completely eradicate all risks, but is able to: 1) minimize and quantify risks associated with timing information and 2) retain good utility with respect to the reachability function. Our solution is motivated by Trace Wringing [125], a work that proposes 'wringing" program traces by compressing traces and formalizes the bounds of leaked information from such traces. For our case, our solution is akin to one of rudimentary aggregation algorithms that a prior work suggested. Xu et al. [126] compared their reduction algorithm CPR and approximation algorithm PCAR against a naive aggregation method where events within 10-seconds time window are aggregated. While this study is very similar to what we have designed, there are several major limitations that we must address. First, their focus is mainly on log reduction so that their study did not test against various different

time windows. Second, their methods aggregated semantically repeated events within the time window. In other words, only edges sharing the same source and destination nodes were merged. In our case, we do not limit ourselves to unifying timestamps only for these redundant edges. Instead, we aggregate the timestamps where events in the same time window (*i.e.*, a batch) share one timestamp and ordering of events is randomized within each batch. Finally, their methods simply attempt to coarsen the granularity of timestamps, a method which is fundamentally vulnerable to clock-edge attacks. Recall that our clock-edge attack from Section 2.4 created a timing side channel that exploits latency differences of hundreds of nanoseconds even though timestamps written on logs were with a granularity of milliseconds.

To address aforementioned limitations, we propose a post-processing mechanism that processes logs into *Batched Timeless Logs*. This method divides a log file L into multiple smaller log files $\{l_0, ..., l_n\}$ (we call each log file as a batched log file) where each smaller log file l_i shares one timestamp t_i. Ordering of log events within each l_i is also randomized. As a result, each l_i becomes a Timeless log, but there exists timing information between batched log files since one timestamp shared within each batch reveal ordering of batches. This implies that unfortunately adversary can potentially launch timing edge attacks mentioned in Section 2.4 across batches by having one event on one batch and another event on another batch. To minimize an adversary exploiting such vulnerabilities, we choose batch size x to be the number of events instead of a time window interval. This change will greatly increase uncertainty of each clock edge as we later show in Section 4.5.2. Furthermore, we choose a random timestamp among x events to represent timing information for each batch.

Additionally, one can further increase protection against timing side-channel attacks by injecting noise to the value of the timestamps for each batch. We choose not to implement such a noise-injection method, because we primarily desiged this countermeasure against clock-edge attacks. Randomizing batch size x for each batch can also benefit security, but for consistency of evaluation (Section 4.5.1), we did not implement this feature. We discuss potential threats associated with vulnerabilities of randomized batches in section 4.5.2. For utility and security evaluation of Batched Timeless Logs, we use the same dataset and machine configurations explained in Section 4.4.

4.5.1 Utility Evaluation

We now discuss utility impacts of batching. Because we use batches, we use two different reachability functions to count false positive ratios. The first function is local reachability that measures reachability within each batch. We construct a dependency graph for each

Table 4.1: Statistics on the ratio of the number of false positive reachable nodes over true positive reachable nodes (false positive ratio) for Theia and Cadet. Q1 stands for 1st Quertile and Q3 stands for 3rd Quartile.

(a) False positive ratio (%) statistics for local reachability for Theia

Batch Size	Q1	Median	Q3	Max	Mean
100	0	0	14.3	1.85K	19.9
1K	0	10.81	62.3	3.65K	77.6
10K	3.42	39.1	110	15.5K	340
100K	12.4	50.0	243	168K	1.09K
1M	4.18	74.6	2.45K	1.22M	56K
$L \approx 2.98M$	0.72	90.7	364	1.14M	36.6K

(b) False positive ratio (%) statistics for global reachability for Theia

Batch Size	Q1	Median	Q3	Max	Mean
100	0	0.06	0.62	40.0	1.47
1K	0	0.76	2.26	4,730	61.7
10K	0.07	1.19	10.6	6,340	165
100K	0.45	4.42	49.4	156K	3.12K
1M	0.72	30.6	217	190K	8.72K
$L \approx 2.98M$	0.72	90.7	364	1.14M	36.6K

(c) False positive ratio (%) statistics for local reachability for Cadet

Batch Size	Q1	Median	Q3	Max	Mean
100	0	0	20	1K	22.8
1K	0	11.1	62.5	4.3K	79.4
10K	0	63.5	448	22.3K	492
100K	13.8	205	2.92K	79.5K	4.16K
1M	197	850	20.2K	571K	38.6K
$L \approx 5.15M$	593	4.25K	96.2K	2.76M	283K

(d) False positive ratio (%) statistics for global reachability for Cadet

Batch Size	Q1	Median	Q3	Max	Mean
100	0	3.32	8.33	13.2K	213
1K	0.0	6.35	27.3	857K	19.7K
10K	0.0	21.3	297K	2.10M	77.3K
100K	5.97	81.8	15.5K	2.11M	110K
1M	77.9	3.15K	59.7K	2.76M	177K
$L \approx 5.15M$	593	4.25K	96.2K	2.76M	283K

batch, and we run reachability analysis on each batch. Because the size of provenance graph is much smaller for each batch compared to a global provenance graph we used in Section 4.4, we may have less than 100 nodes to choose from for each provenance depending on size of the batch. In this case, we simply run reachability analysis on all nodes. Table 4.1a

displays experimentation results for various batch sizes for dataset Theia and Table 4.1c displays those results for dataset Cadet. Note that when batch size is equal to that of log size, then we have a complete Timeless Log from Section 4.4, so the experiment results are consistent with that of the previous section. The false positive ratio is calculated as the number of reachable nodes from a timeless batch over the number of reachable nodes from a timestamped batch. To compare results across different batch sizes, we display results in ratio of false positive reachable nodes. As mentioned in previously, we choose not to randomize batch size for consistency across different batches. For Theia with the smallest batch size of 100, more than half of batches have no false positive reachable nodes. However, in the worst case a batch can have increase in 1850% of false positive reachable nodes over true positive reachable nodes. After a batch size of 1,000, medians of false positive ratio becomes too large, suggesting that even if graph is small, removing all timing information results in a big utility loss.

The second function is global reachability that measures reachability across the whole dataset. Provenance graph analysis mainly aims to track dependency of events for a long period of time. However, experiment results from local reachability only considers reachability within a batch which has very small time window. (We go over relationship between batch size and time window in Section 4.5.2) Thus, this global reachability function aims to measure the effects of batch size against increase in false positive reachable nodes across the entire dataset. This is done by first creating a provenance graph for each batch and connecting each graph where duplicate nodes across batches are connected with edges that are equivalent to dotted arrows for versioned graphs in Figure 4.1c. For this case, we do not need to worry about problems associated with choosing 100 source nodes, so all batches use the same 100 sources for consistency.

Table 4.1b and Table 4.1d shows that false positive ratio across different batch sizes in terms of global reachability. For Theia with batch size of 100, more than 75 percent of source nodes had less than 1 percent of increase in the number of false positive reachable nodes. However, one thing that is not shown from this statistic is that the global provenance graph is much larger than that of local provenance graph so that even if percentages increase are small for some batch sizes, the actual increase in the number of false positive reachable nodes can be very large. For Theia with a batch size of 100, the number of false positive nodes on average is 7.6 for local reachability but that of global reachability is 172.3. In the next subsection, we examine differences in time window intervals across different batch sizes for security.

4.5.2 Security Evaluation

For Timeless Logs in Section 4.4, we do not require security evaluation since the logs are time-oblivious. However, for Batched Timeless Logs, we must examine vulnerabilities associated with clock-edge attacks across batches, so this section examines tradeoffs between batch size and time windows. As shown previously, the smaller the batch size, the better the utility . In the extreme case, one can set batch size to one and retain perfect utility no timing information is lost. However, security risks against batched logs increases as batch size decreases. Even if we randomize either representative timestamp or batch size as we designed, if batch size is too small, then the timing window between batches can be smaller than the timing latency of side-channel attack which makes clock-edge attacks across batches significantly easier to perform.

Table 4.2: Statistics for an interval of time windows against different batch sizes.

(a) Results for Theia dataset.

Batch Size	Num Batches	Median	Mean	STD
100	29.3K	0.016 sec	0.548 sec	1.41 sec
1K	2.93K	1.46 sec	5.53 sec	9.67 sec
10K	293	44.5 sec	55.4 sec	44.5 sec
100K	30	524 sec	541 sec	231 sec
1M	3	79.9 min	90.1 min	15.5 min
L	1	4.51 hrs	4.51 hrs	-

(b) Results for Cadet dataset.

Batch Size	Num Batches	Median	Mean	STD
100	51.5K	0.070 sec	3.369 sec	10.87 sec
1K	5.15K	29.5 sec	33.99 sec	30.9 sec
10K	516	370 sec	340 sec	216 sec
100K	52	56.3 min	56.2 min	25.5 min
1M	6	8.64 sec	8.11 hrs	195 min
L	1	48.7 hrs	48.7 hrs	-

Table 4.2a shows the tradeoffs between batch size and interval of time window for each batch for the Theia dataset and Table 4.2b shows results for the Cadet dataset. In addition to multiple randomization schemes that we implemented, we choose batch sizes to be a fixed number of events, not a time interval. As a result, we observe relatively large standard deviation for time windows across batches. In case of Theia dataset, the median time window interval across batches is 0.016 sec but the mean is 0.548 sec with a standard deviation of 1.41 sec for a batch size of 100. For Cadet dataset, the median, the mean and a standard deviation are 0.070 sec, 3.369 sec and 10.87 sec for the same batch size. This

will greatly affect an adversary trying to infer timing information, but this will not affect an adversary if the timing window of timing side-channel attacks is much larger than the intervals. For example, if adversary's side-channel attack relies on timing window between two events to be larger than one second, then a batch size of 100 would be too small even with randomization. For such an adversary, the number of batches dictates the number of bits leaked by the logs. If we decrease the number of batches by increasing batch sizes, then the number of bits leaked decreases, but as we showed previously, the utility of resulting logs degrades as a result. Therefore, system admins who apply Batched Timeless log as a main countermeasure against clock-edge attacks must make tradeoffs between utility and security. Fortunately, our attack prototypes demonstrated in Section 2.4 exploit very fine-grained timing side channels in hundreds of nanoseconds such that even smallest batch size of 100 would effectively mitigate all of our attack examples.

4.6 CONCLUSION

We introduce Timeless logs, an approach to mitigate clock-edge Termite attacks and we study on utility-security tradeoffs associated with reachability functions against provenance graphs. While a complete time-oblivious log provides very little utility value based on our reachability analysis, Batched Timeless logs do exhibit reasonable tradeoffs when batch sizes are sufficiently small. Batched Timeless logs also provide a way of quantifying risks against Termite attacks where small batches provide good utility, but are potentially vulnerable against possible coarse-grained side channel attacks.

CHAPTER 5: CONCLUSIONS

In this work, we have demonstrated and explored side-channel attacks against logs and their countermeasures. We highlighted and discussed previously overlooked topics of exploiting implicit information against logs to extract secrets. We then presented countermeasures to mitigate against such threats under two different scenarios.

In Chapter 2, we demonstrate possibility of launching fine-grained side channel attacks against seemly coarse-grained channel. Our first two attacks, membership inference attacks against log contents and keystroke timing attacks against SSH, display how to extract coarse-grained secrets through implicit information. Furthermore, we share surprising results where it is possible to mount a fine-grained timing channel attack against a system logger. Our experimentation showed that despite use of coarse-grained time sources, the logger nonetheless still retained accurate timing information that can be leveraged by remote clock-edge attacks. Our PoC NetSpectre attacks show that these threats against system loggers cannot be underestimated as the attack can steal any secret information stored in the system.

We show how to address threats associated with side-channel attacks against logs in Chapter 3 and Chapter 4. In Chapter 3 we introduce a high-level programming stack that leverages TEE and data-oblivious computation. This work is originally intended to provide a secure programming environment against untrusted software including OS and hypervisor. However, our analysis shows how our DOVE's architecture can provide an interface for side-channel resistant logging by introducing the concept of pseudonyms. Our design of pseudonyms and DOT ensures confidentiality of secret information in original computation by completely decoupling all secrets from logs that record operations for high-level language interpreters.

Finally, in Chapter 4, we provide a detailed study that examines utility-security tradeoffs with respect to timing information stored in logs. While there exits a prior work that limitedly conducted such analysis, our work is the first to make a systematic analysis of the impact of removal of timestamps. Then we designed Timeless logs where its batched version provided good security guarantee with reasonable false positive ratios with respect to reachability, a widely used utility function in the audit log research community.

5.1 DISCUSSION AND FUTURE WORKS

We conclude this thesis with a discussion and potential research opportunities associated with side channels and logs.

5.1.1 Current DOVE 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.

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 3.4.4 to mask out architectural state updates when the actual input requires fewer loop iterations.

5.1.2 Additional Countermeasure Suggestions

We now outline potential countermeasures for Termite attacks which we did not cover in Chapter 4. Much like their namesake, however, Termite attacks, once found, are hard to remove. Overall, we believe that there is no one-size-fits-all solution against Termite attacks. A proper countermeasure will likely involve techniques from each of the areas described above. Developing defenses against Termite attacks will be a productive avenue for future work.

Reducing log content. A common security maxim is to not log secret data. As we have seen, what constitutes secret data is quite complex. Thus, by paring down the logs' content, we would ostensibly be shrinking the attack surface; the attacker now has less data over which to build an exploit. We have seen, however, that this is not sufficient. With the log metadata of a single process, we can extract secret information, as seen in Section 2.3.1. Moreover, the question of how much content to eliminate remains. Relatively coarse-grained timestamps were enough to recover keystrokes in Section 2.3.2. The mere presence of logging between two processes allowed us to build our timing side channel in Section 2.4.

Limiting log access. Our threat model (Section 4.2) expects log disclosure to occur, but logs could potentially be encrypted [2, 115]. However, encryption could still leak information

about the underlying content. Logs of differing length could be encrypted to the same or similar lengths, allowing for similar analysis. This type of length-based side channel has been employed successfully against TLS-encrypted network traffic [195]. Encrypting the logs also has questions of key management, as key principals still need access to the log. Too many privileged users could be a security flaw, but not enough could slow incident recovery.

In the extreme case, a concerned administrator could disable LAS and logging entirely. While this would close the side channel (vacuously), it would also make system administration problematic. Without logs, debugging would be difficult and, potentially, security events would remain undetected. This is akin to disabling processor speculation to close speculative execution side channels [65]: secure against the attack, but impractical in the real world.

5.1.3 Limitations and Future Works for Timeless Logs

Unfortunately, our current study was conducted in relatively small dataset and we have not had a chance to run experiments on DARPA dataset collected by hosts other than Theia and Cadet. We are interested in conducting future experiments on different datasets with larger size.

While our work discusses potential of prior works (especially log reduction algorithms) against Termite attacks as countermeasures, we choose not to implement any of these methods. In particular, DPR proposes storing logs as versioned graphs which can potentially wash out a lot of timing information as we discussed in Section 4.3. We have not found a systematic way to apply their works as countermeasures, but we believe there are potential opportunities associated with versioned graphs as countermeasures against timing side-channel attacks. Furthermore, there has been a prior work that uses application logs [36] to assist log analysis. This work aims to reduce size of provenance graph when making reachability analysis by getting semantic information of processes that generate application logs. This work if used in conjunction with our Batched Timeless logs may be able to reduce false positives, but in order to do so, we need to justify our threat model that an adversary does not learn any timing information from such application logs as well. In summary, it would be interesting if prior works can be applied either solely or composed with Batched Timeless logs to achieve better utility and stronger security guarantees.

As mentioned in threat model, this study does not consider threats associated with adversary learning about timing information through means other than timestamps and relative ordering of events. It would be interesting future work to find implicit timing information in logs and assess if countermeasures we suggested are plausible.

REFERENCES

[1] Wikipedia contributors, "Logging (computing) — Wikipedia, the free encyclopedia," 2023, [Online; accessed 15-June-2023]. [Online]. Available: https://en.wikipedia.org/wiki/Logging_(computing)

[2] I. Ray, K. Belyaev, M. Strizhov, D. Mulamba, and M. Rajaram, "Secure logging as a service—delegating log management to the cloud," *IEEE Systems Journal'13*.

[3] B. Böck, D. Huemer, and A. M. Tjoa, "Towards more trustable log files for digital forensics by means of "trusted computing"," in *IEEE AINA'10*.

[4] SQLite, "The error and warning log," https://www.sqlite.org/errlog.html.

[5] R. Paccagnella, P. Datta, W. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian, "Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution," in *NDSS'20*.

[6] J. Biskup and U. Flegel, "Transaction-based pseudonyms in audit data for privacy respecting intrusion detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2000, pp. 28–48.

[7] J. Biskup and U. Flegel, "On pseudonymization of audit data for intrusion detection," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 161–180.

[8] F. Brunton and H. Nissenbaum, "Vernacular resistance to data collection and analysis: A political theory of obfuscation," *First Monday*, 2011.

[9] R. Shokri, "Privacy games: Optimal user-centric data obfuscation," *arXiv preprint arXiv:1402.3426*, 2014.

[10] Y. Long, L. Xu, and C. A. Gunter, "A hypothesis testing approach to sharing logs with confidence," in *CODASPY'20*.

[11] K. Berlin, D. Slater, and J. Saxe, "Malicious behavior detection using windows audit logs," in *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, 2015, pp. 35–44.

[12] W. U. Hassan, L. Aguse, N. Aguse, A. Bates, and T. Moyer, "Towards scalable cluster auditing through grammatical inference over provenance graphs," in *NDSS'18*.

[13] M. Dymshits, B. Myara, and D. Tolpin, "Process monitoring on sequences of system call count vectors," in *ICCST'17*.

[14] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *(USENIX Security'17)*.

[15] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie et al., "MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation," in *NDSS'18*.

[16] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-Supported Cost-Effective audit logging for causality tracking," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/ma-shiqing pp. 241–254.

[17] H. B. Lee, T. Jois, C. W. Fletcher, and C. A. Gunter, "DOVE: A Data-Oblivious Virtual Environment," in *NDSS'21*.

[18] Apache Software Foundation, "Log files - Apache HTTP server version 2.4," https://httpd.apache.org/docs/2.4/logs.html.

[19] The PostgreSQL Global Development Group, "Error Reporting and Logging," https://www.postgresql.org/docs/current/runtime-config-logging.html.

[20] Red Hat, Inc., "Auditing the system Red Hat Enterprise Linux 8," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/security_hardening/auditing-the-system_security-hardening.

[21] FreeBSD, "DTrace," https://wiki.freebsd.org/DTrace.

[22] Microsoft, "About Event Tracing," https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[23] P. Fei, Z. Li, Z. Wang, X. Yu, D. Li, and K. Jee, "{SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2987–3004.

[24] Microsoft, "Microsoft Sysmon," https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon, accessed: 2022-01-28.

[25] S. Grubb, *auditctl(8) Linux manual page*, July 2021.

[26] S. Grubb, *audit.rules(7) Linux manual page*, January 2019.

[27] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 307–325.

[28] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *NDSS'16*.

[29] National Security Agency, "Controlled Access Protection Profile, Version 1.d," https://www.niap-ccevs.org/Profile/Info.cfm?PPID=14&id=14, 1999, accessed: 2022-08-18.

[30] R. Paccagnella, K. Liao, D. Tian, and A. Bates, "Logging to the danger zone: Race condition attacks and defenses on system audit frameworks," in *CCS'20*.

[31] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics." in *NDSS*, 2021.

[32] M. A. Inam, W. U. Hassan, A. Ahad, A. Bates, R. Tahir, T. Xu, and F. Zaffar, "Forensic analysis of configuration-based attacks," in *NDSS'22*.

[33] J. Zengy, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "Shadewatcher: Recommendation-guided cyber threat analysis using system audit records," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 489–506.

[34] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4363–4376, 2021.

[35] S. Yoo, J. Jo, B. Kim, and J. Seo, "Longline: Visual analytics system for large-scale audit logs," *Visual Informatics*, vol. 2, no. 1, pp. 82–97, 2018.

[36] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis," in *Network and distributed system security symposium*, 2020.

[37] C. Yagemann, M. A. Noureddine, W. U. Hassan, S. Chung, A. Bates, and W. Lee, "Validating the integrity of audit logs against execution repartitioning attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3337–3351.

[38] A. Gehani and D. Tariq, "SPADE: Support for Provenance Auditing in Distributed Environments," *Middleware'12*.

[39] N. Husted, S. Quresi, and A. Gehani, "Android provenance: diagnosing device disorders," in *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*, 2013.

[40] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1554–1554.

[41] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," IACR'16.

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *CT-RSA'06*.

[43] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*.

[44] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *IACR'16*.

[45] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE S&P*, 2019.

[46] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting Secret Keys via Branch Prediction," *IACR'06*.

[47] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS'18*.

[48] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P'15*.

[49] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*.

[50] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," IACR'18.

[51] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *CoRR'17*.

[52] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*.

[53] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS'17*.

[54] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security'18*.

[55] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security'16*.

[56] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS '16*.

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

[58] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1109. Springer, 1996, pp. 104–113.

[59] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference.* Springer, 2006, pp. 1–20.

[60] Y. Yarom and K. Falkner, "{FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack," in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 719–732.

[61] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[62] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *ESORICS'19.*

[63] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," Cryptology ePrint Archive, Report 2018/1060, 2018, https://ia.cr/2018/1060.

[64] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin et al., "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[65] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher et al., "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 2019, pp. 1–19.

[66] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *USENIX Security'16.*

[67] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *International Conference on Financial Cryptography and Data Security.* Springer, 2017, pp. 247–267.

[68] T. Van Goethem, C. Pöpper, W. Joosen, and M. Vanhoef, "Timeless timing attacks: Exploiting concurrency to leak secrets over remote connections," in *USENIX Security'20.*

[69] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2019. [Online]. Available: https://www.R-project.org/

[70] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," 1959.

[71] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *CCS'17*.

[72] Intel®, "Intel® software guard extensions programming reference," 2014.

[73] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel sgx," in *NDSS'18*.

[74] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS'19*, https://eprint.iacr.org/2018/808.

[75] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *USENIX Security'16*.

[76] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Security'15*.

[77] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *EuroSec'17*.

[78] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX ATC'17*.

[79] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," *CoRR'17*.

[80] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," *CoRR'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] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM TOCS'15*.

[83] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX ATC'17*.

[84] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB linux applications with sgx enclaves," in *NDSS'17*.

[85] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." *HASP'13*.

[86] Intel®, "Intel® software guard extensions sdk for linux os developer reference," 2020.

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

[88] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX'17*.

[89] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *PKC'06*.

[90] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *FSE'05*.

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

[92] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional encryption using Intel SGX," in *CCS'17*.

[93] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors," in *CCS'17*.

[94] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *NSDI'17*.

[95] S. Eskandarian and M. Zaharia, "An oblivious general-purpose SQL database for the cloud," *CoRR'17*.

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

[97] S. Tople and P. Saxena, "On the trade-offs in oblivious execution techniques," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Springer'17.

[98] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A Data Oblivious Filesystem for Intel SGX," in *NDSS'18*.

[99] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P'09*.

[100] D. Darais, C. Liu, I. Sweet, and M. Hicks, "A language for probabilistically oblivious computation," *CoRR'17*.

[101] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," *SecDev'17*.

[102] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *S&P'15*.

[103] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," *IACR'15*.

[104] E. Stefanov, M. van Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," *CCS'13*.

[105] M. Blanton, A. Steele, and M. Alisagari, "Data-oblivious graph algorithms for secure computation and outsourcing," in *ASIA CCS'13*.

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

[107] J. Doerner, D. Evans, and abhi shelat, "Secure stable matching at scale," *IACR'16*.

[108] T.-H. H. Chan, Y. Guo, W.-K. Lin, and E. Shi, "Cache-oblivious and data-oblivious sorting and applications," *IACR'17*.

[109] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar, "Tiny-garble: Highly compressed and scalable sequential garbled circuits," in *S&P'15*.

[110] S. Zahur and D. Evans, "Circuit structures for improving efficiency of security and privacy tools," in *S&P'13*.

[111] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, "Oblivious data structures," *IACR'14*.

[112] M. Ring, D. Schlör, S. Wunderlich, D. Landes, and A. Hotho, "Malware detection on windows audit logs using lstms," *Computers & Security*, vol. 109, p. 102389, 2021.

[113] H. D. Kuna, R. García-Martinez, and F. R. Villatoro, "Outlier detection in audit logs for application systems," *Information Systems*, vol. 44, pp. 22–33, 2014.

[114] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos, "Pillarbox: Combating next-generation malware with fast forward-secure logging," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 46–67.

[115] S. Zawoad, A. K. Dutta, and R. Hasan, "Towards building forensics enabled cloud through secure logging-as-a-service," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 2, pp. 148–162, 2015.

[116] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *ASIACCS'17*.

[117] E. Lundin and E. Jonsson, "Privacy vs. intrusion detection analysis." in *Recent Advances in Intrusion Detection*, 1999.

[118] C. Rath, "Usable privacy-aware logging for unstructured log entries," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016, pp. 272–277.

[119] J. Xu, J. Fan, M. Ammar, and S. B. Moon, "On the design and performance of prefix-preserving ip traffic trace anonymization," in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001, pp. 263–266.

[120] J. Xu, J. Fan, M. H. Ammar, and S. B. Moon, "Prefix-preserving ip address anonymization: Measurement-based security evaluation and a new cryptography-based scheme," in *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* IEEE, 2002, pp. 280–289.

[121] J. Wilkes, "More Google Cluster Data," https://ai.googleblog.com/2011/11/more-google-cluster-data.html, accessed: 2022-03-29.

[122] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[123] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Obfuscatory obscanturism: making workload traces of commercially-sensitive systems safe to release," in *IEEE Network Operations and Management Symposium'12*.

[124] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," *SIAM Journal on Computing*, vol. 45, no. 3, pp. 882–929, 2016.

[125] D. Dangwal, W. Cui, J. McMahan, and T. Sherwood, "Trace wringing for program trace privacy," *ASPLOS'19*.

[126] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High Fidelity Data Reduction for Big Data Security Dependency Analyses," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2976749.2978378 p. 504–516.

[127] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller, "Dependence-Preserving Data Compaction for Scalable Forensic Analysis," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/hossain pp. 1723–1740.

[128] N. Michael, J. Mink, J. Liu, S. Gaur, W. U. Hassan, and A. Bates, "On the Forensic Validity of Approximated Audit Logs," in *Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3427228.3427272 p. 189–202.

[129] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure linux containers with intel SGX," in *OSDI'16*.

[130] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *USENIX ATC 17*.

[131] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "TrustJS: Trusted client-side execution of javascript," in *EuroSec'17*.

[132] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang, "Running Language Interpreters Inside SGX: A Lightweight,Legacy-Compatible Script Code Hardening Approach," in *Asia CCS'19*.

[133] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *OSDI'16*.

[134] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert, "Secure and private function evaluation with Intel SGX," in *SIGSAC'19*.

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

[136] A. Avalos, H. Pan, C. Li, J. P. Acevedo-Gonzalez, G. Rendon, C. J. Fields, P. J. Brown, T. Giray, G. E. Robinson, M. E. Hudson et al., "A soft selective sweep during rapid evolution of gentle behaviour in an Africanized honeybee," *Nature communications*, vol. 8, no. 1, p. 1550, 2017.

[137] F. Chen, M. Dow, S. Ding, Y. Lu, X. Jiang, H. Tang, and S. Wang, "PREMIX: Privacy-preserving estimation of individual admixture," in *AMIA ASP'16*.

[138] M. N. Sadat, M. M. A. Aziz, N. Mohammed, F. Chen, S. Wang, and X. Jiang, "SAFETY: Secure GWAS in federated environment through a hybrid solution with Intel SGX and homomorphic encryption," *arXiv preprint arXiv:1703.02577*, 2017.

[139] F. Chen, S. Wang, X. Jiang, S. Ding, Y. Lu, J. Kim, S. C. Sahinalp, C. Shimizu, J. C. Burns, V. J. Wright et al., "Princess: Privacy-protecting rare disease international network collaboration via encryption through software guard extensions," *Bioinformatics*, vol. 33, no. 6, pp. 871–878, 2016.

[140] F. Chen, C. Wang, W. Dai, X. Jiang, N. Mohammed, M. M. Al Aziz, M. N. Sadat, C. Sahinalp, K. Lauter, and S. Wang, "PRESAGE: privacy-preserving genetic testing via software guard extension," *BMC medical genomics*, vol. 10, no. 2, p. 48, 2017.

[141] M. M. A. Aziz, M. N. Sadat, D. Alhadidi, S. Wang, X. Jiang, C. L. Brown, and N. Mohammed, "Privacy-preserving techniques of genomic data—a survey," *Briefings in Bioinformatics*, vol. 20, no. 3, pp. 887–895, 2017.

[142] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on {SSH}," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[143] M. Schwarzl, E. Kraft, M. Lipp, and D. Gruss, "Remote memory-deduplication attacks," in *NDSS'22*.

[144] J. Wikner, C. Giuffrida, H. Bos, and K. Razavi, "Spring: Spectre Returning in the Browser with Speculative Load Queuing and Deep Stacks," in *WOOT'22*, 2022.

[145] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 286–296.

[146] A. Slagell and W. Yurcik, "Sharing computer network logs for security and privacy: a motivation for new methodologies of anonymization," in *Workshop of the 1st International Conference on Security and Privacy for Emerging Areas in Communication Networks, 2005.*, 2005, pp. 80–89.

[147] R. Pang and V. Paxson, "A high-level programming environment for packet trace anonymization and transformation," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 339–351.

[148] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 645–662.

[149] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[150] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[151] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "Bigger, longer, fewer: what do cluster jobs look like outside google," 2017.

[152] M. Naor and M. Yung, "Public-key cryptosystems provably secure against chosen ciphertext attacks," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 427–437.

[153] R. A. Clothey, E. F. Koku, E. Erkin, and H. Emat, "A voice for the voiceless: online social activism in uyghur language blogs and state control of the internet in china," *Information, Communication & Society*, vol. 19, no. 6, pp. 858–874, 2016.

[154] B. Krebs, "Online cheating site ashleymadison hacked," Krebs on security, 2015.

[155] S. Vinberg and J. Overson, "2021 credential stuffing report," https://www.f5.com/labs/articles/threat-intelligence/2021-credential-stuffing-report.

[156] The PHP Group, "password_hash," https://www.php.net/manual/en/function.password-hash.php.

[157] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, “Netcat: Practical cache attacks from the network,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 20–38.

[158] K. Zhang and X. Wang, “Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems.” in *USENIX Security Symposium*, vol. 20, 2009, p. 23.

[159] Sysdig, “Sysdig – Open Source System Capturing.” https://sysdig.com/opensource/inspect/, accessed: 2022-03-29.

[160] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, “Trustworthy Whole-System provenance for the Linux kernel,” in *USENIX Security’15*.

[161] Microsoft, “Process Monitor v3.89.” https://docs.microsoft.com/en-us/sysinternals/downloads/procmon, accessed: 2022-03-29.

[162] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, “Practical keystroke timing attacks in sandboxed javascript,” in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 191–209.

[163] S. A. Crosby, D. S. Wallach, and R. H. Riedi, “Opportunities and limits of remote timing attacks,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, pp. 1–29, 2009.

[164] Nagle, John, “The trouble with the Nagle algorithm,” https://developers.slashdot.org/comments.pl?sid=174457&threshold=1&commentsort=0&mode=thread&cid=14515105, accessed: 2022-05-08.

[165] E. K. Chan, “Handy R functions for genetics research,” https://github.com/ekfchan/evachan.org-Rscripts, 2019.

[166] P. C. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *CRYPTO’99*.

[167] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, “EDDIE: EM-Based Detection of Deviations in Program Execution,” in *ISCA’17*.

[168] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, “A2: Analog Malicious Hardware,” in *S&P’16*.

[169] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *S&P’19*.

[170] 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)*.

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

[172] I. Corporation, "Processor Counter Monitor," https://github.com/opcm/pcm.

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

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

[175] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *CSF'13*.

[176] H. B. Lee, T. Jois, C. W. Fletcher, and C. A. Gunter, "DOVE: A Data-Oblivious Virtual Environment," in *Arxiv eprint*.

[177] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," in *ASPLOS'04*.

[178] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan 2003.

[179] P.-L. Aublin, F. Kelbert, D. O'keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "TaLoS: Secure and transparent TLS termination inside SGX enclaves," *Imperial College London, Tech. Rep*, vol. 5, p. 2017, 2017.

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

[181] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[182] V. Costan and S. Devadas, "Intel SGX explained," IACR'16.

[183] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *SIGPLAN Not.*, vol. 50, no. 4, pp. 87–101, Mar. 2015.

[184] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *SIGCHI'10*.

[185] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM TKDD'07*.

[186] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[187] K. Tang, K. R. Thornton, and M. Stoneking, "A new approach for using genome scans to detect recent positive selection in the human genome," *PLoS biology*, vol. 5, no. 7, p. e171, 2007.

[188] X. Gao and J. Starmer, "Human population structure detection via multilocus genotype clustering," *BMC genetics*, vol. 8, no. 1, p. 34, 2007.

[189] M. Nei, "F-statistics and analysis of gene diversity in subdivided populations," *Annals of human genetics*, vol. 41, no. 2, pp. 225–233, 1977.

[190] B. S. Weir and C. C. Cockerham, "Estimating F-statistics for the analysis of population structure," *evolution*, vol. 38, no. 6, pp. 1358–1370, 1984.

[191] K. H. Lee, X. Zhang, and D. Xu, "LogGC: garbage collecting audit log," in *CCS'13*.

[192] M. A. Inam, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan, "FAuST: Striking a Bargain between Forensic Auditing's Security and Throughput," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3564625.3567990 p. 813–826.

[193] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hossain pp. 487–504.

[194] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[195] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 191–206.