Pave: Information Flow Control for Privacy-preserving Online Data Processing Services

Minkyung Park[*]
University of Texas at Dallas
Richardson, Texas, USA
minkyung.park@utdallas.edu

Jaeseung Choi
Sogang University
Seoul, Republic of Korea
jschoi22@sogang.ac.kr

Hyeonmin Lee
University of Virginia
Charlottesville, Virginia, USA
frv9vh@virginia.edu

Taekyoung Kwon[†]
Seoul National University
Seoul, Republic of Korea
tkkwon@snu.ac.kr

Abstract

In online data-processing services, a user typically hands over personal data to a remote server beyond the user's control. In such environments, the user cannot be assured that the data is protected from potential leaks. We introduce Pave, a new framework to guarantee data privacy while being processed remotely. Pave provides an arbitrary data-processing program with a sandboxed execution environment. The runtime monitor, PaveBox, intercepts all data flows into and out of the sandbox, allowing them only if they do not compromise user data. At the same time, it guarantees that the benign flows will not be hampered to preserve the program's functionality. As the PaveBox is built on top of Intel SGX, a user can verify the integrity and confidentiality of the PaveBox by remote attestation. We provide a formal model of Pave and prove its security and carry out the quantitative analysis with prototype-based experiments.

CCS Concepts: • **Security and privacy → Pseudonymity, anonymity and untraceability**.

Keywords: Privacy-preserving Computation, Intel SGX, Secure Multi-Execution, Software Fault Isolation

ACM Reference Format:
Minkyung Park, Jaeseung Choi, Hyeonmin Lee, and Taekyoung Kwon. 2025. Pave: Information Flow Control for Privacy-preserving Online Data Processing Services. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3716266

[*]This work was conducted while the first author was at Seoul National University.
[†]Corresponding author.

1 Introduction

Online data-processing services such as Pixlr (photo editing), VEED (video editing), and ezyZip (file compression) are becoming more prevalent. To use such an online service, a user typically uploads the user's personal data, selects how to process it, and receives the result through a browser.

One of their critical drawbacks is the loss of control over data. After the data is sent to a remote server, the user cannot figure out how it is processed, and whether the service provider stores it for other purposes. Also, a data-processing server may be vulnerable to attacks for data leakage. For instance, a server hosting file converter services was hacked and the attacker obtained control over the server [96]. A series of reported data breaches [9, 34, 47] show potential risks of using such services. Also, since an online service provider typically deploys its service on a cloud, the cloud provider that has access to user data can also be compromised [29].

Trusted hardware, such as Intel SGX, has emerged as one of the solutions to protect user data even when processed in a cloud environment [10, 40, 54, 73]. A data-processing program is enforced not to leak user data by a controller, which is in turn protected by Intel SGX. Intel SGX provides an isolated container, called an enclave, which the OS cannot access. However, as Intel SGX protects only a user-level application (ring 3), using OS-provided services such as standard I/O, network, and persistent storage cannot be protected, and hence can be exposed to the service provider. Specifically, when the program requests a syscall for the OS-provided services, it exits the enclave via the pre-defined *software interfaces* (between the enclave and the OS). Thus, any malicious service provider may try to analyze the user data by conducting a covert channel attack, intentionally encoding the data into syscall parameters or the timing between syscall invocations. For instance, the elapsed time between two syscalls can be decided by the value of private data.

To the best of our knowledge, Ryoan [40] is the only approach to thwart this covert channel attack via the software interfaces. Ryoan targets a request-oriented data-processing program, which refers to a program that accepts only one request and generates one response over the network. Specifically, it eliminates this covert channel of the software interfaces by blocking all the syscalls after receiving the request, except for the one response transmission. Furthermore, to block the timing attack, Ryoan sets a fixed time interval between receiving the request and sending the response.

In this paper, we present Pave, an information flow control (IFC) framework with an SGX-based sandbox, capable of supporting general programs (i.e., not limited to a request-oriented program) and preserving timing-sensitive privacy. Similar to Ryoan, Pave provides an SGX-based sandbox, dubbed a PaveBox, which means that Intel SGX protects the PaveBox, which controls an untrusted data-processing program. Compared to Ryoan, Pave can preserve the functionality of a wider range of data-processing programs. While Ryoan unconditionally forbids syscalls after receiving the user request, Pave does not pose a constraint on the syscall invocation as long as it does not violate the user privacy. Therefore, Pave can accommodate general programs such as interactive applications that require multiple data exchanges and long-running applications that continuously invoke syscalls to keep their states[1]. Further, Pave does not enforce the execution time to be constant, as long as it does not leak the user data. To sum up, Pave enforces every program to protect the user privacy. At the same time, if a program is already secure without Pave, then Pave does not restrict its functionality.

To preserve the privacy and the functionality, PaveBox needs a criterion for determining whether any flow is benign or malicious. For this, we formally model these requirements as timing-sensitive non-interference and functionality, which will be detailed in §3. Regardless of private user data, a benign program given the same public data should generate a consistent syscall pattern (e.g., the timing of invocations). By contrast, malicious programs would show different patterns of syscall invocations depending on the private data.

Specifically, Pave needs to identify privacy-independent syscall patterns. We simulate a data-processing program with dummy data, which serves on behalf of private data. Then, the simulation results in a sequence of syscall invocations. When the program is executed with the real user data, the PaveBox enforces it to generate the same syscall pattern as that of the simulation. In addition, since the benign program always generates a consistent pattern, the generated pattern by the simulation would match that of the real execution, which preserves the functionality.

[1]Although an interactive application program can be rewritten into a request-oriented program, which takes previous states as a new input, it is not scalable as the number of states increases.

To formally prove the properties of Pave, we define the behavior of a data-processing program under our language model with and without a PaveBox. Then, we prove that for any program, its behaviors are consistent under the PaveBox, regardless of sensitive user data. Also, we prove that if the program is already non-interferent without Pave, the program behavior would not be modified by the PaveBox.

Lastly, we implement the prototype of Pave, which is tested for six use cases, to demonstrate its feasibility and applicability. We evaluate its computation overheads compared to Linux applications without Pave or Intel SGX.

We outline the contributions of this paper as follows.

- We design an IFC framework Pave to protect user data against an adversary who is a potentially malicious service provider. It is the first approach that models the requirements as timing-sensitive non-interference and functionality preservation.
- We formally model a data-processing program with and without a PaveBox and define the notion of the privacy and the functionality under our system settings. In addition, we formally prove that Pave preserves both.
- We implement a proof-of-concept of PaveBox and evaluate its computation overhead to demonstrate its feasibility.
- For further research, we make our code publicly available in https://github.com/alsroad/pave.

2 Background and Threat Model

2.1 Intel SGX

Intel SGX is a set of instructions to protect a user-level process. It enables an application to build an enclave, a memory area protected from disclosure or modification by any privileged software. The enclave memory is encrypted by the Memory Encryption Engine and decrypted when it is loaded to CPU. Intel SGX provides remote attestation that allows a remote entity to verify the integrity of the code and initial data of an enclave by checking its signed hash.

Inside the enclave, operations that need the intervention of an OS are prohibited. Intel SGX blocks an enclave application from directly invoking syscalls to use the OS-provided services (e.g., storage, network, or time). Instead, an enclave developer defines software interfaces. Through this software interface, a thread can temporarily exit the enclave, invoke a syscall from the host function, and re-enter it.

2.2 Threat Model

Pave assumes three types of participants similar to [29]. **A service provider** provides functions that process user data and develops a software *module* to embody those functions. The service provider should run its module using Pave to assert and attest that its service protects the user privacy. A data-processing service may be provided by combining one or more such modules, each of which might be developed by a different service provider. **A user**, or a user agent such

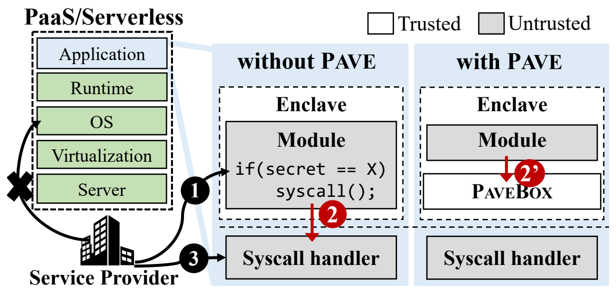


Figure 1. A module is deployed without and with PAVE.

as a browser or an application, wishes to protect user data while being processed. The user initiates a request to the module, transmits the data, selects how to process it (maybe repeatedly), and receives the final result. As the user trusts only PAVE, it uses the data-processing service only if the software module is protected by PAVE. **A cloud provider** provides a computing platform equipped with Intel SGX as PaaS (Platform-as-a-Service) or serverless models. In PaaS and serverless models, as illustrated in Figure 1, the cloud provider manages not only network, storage, server, and virtualization layers (as IaaS model), but also the OS and runtime layers. Since these models reduce the burden for a cloud user to manage the underlying layers, current cloud providers support these models (e.g., AWS ECS, Azure Functions, and Google Functions). On the other hand, the application layer consists of a PAVEBOX, an external syscall handler, and a data-processing module. Here, the external syscall handler refers to the user-level code that redirects a syscall request from the enclave to the OS and its response from the OS to the enclave. While the service provider can observe the software interfaces [2], the service provider cannot have root privileges to access the lower layers (e.g., OS). Also, the user can employ Intel SGX DCAP [71] to ensure that the module is running on the cloud provider, which in turn guarantees that the service provider does not have the root privilege.

We assume that the service provider and cloud provider cannot be trusted, whereas the user is benign. Since Intel SGX protects the enclave from any direct access, we focus on the scenario in which the adversary uses the software interface as a covert- or side-channel to leak the user data. While some syscalls can be implemented without escaping the enclave [85], the syscalls that use network device or storage must be implemented with the software interface. The service provider and cloud provider can observe such syscalls invoked by the module code and attempt to infer private user data. From now on, a *syscall* refers to a syscall implemented with the software interface. Note that because the service provider can deploy the module, it is capable of launching both covert and side-channel attacks *via the syscall interface*. Therefore, in terms of the attacks via software interface, the service provider is a more powerful adversary than the cloud provider, who is limited to side-channel attacks. In the later sections, we assume that the adversary is a service provider who targets the software interface as an attack surface.

We assume that the cloud provider does not collude with the service provider. Therefore, the service provider cannot launch side-channel attacks requiring the root privileges, such as exploitation on Branch Prediction Units (BPU) [31, 50], L1/L2 cache [55], Translation Lookaside Buffer (TLB) [37], last-level cache (LLC) [76], DRAM [63], page table [56, 98], or microarchitecture [20, 46, 86, 87, 91]. We discuss the feasibility of such attacks by the cloud provider in §7. Note that fully preventing Iago-type attacks [19] is challenging [88]. Although these are out of our design scope, PAVE should be promptly updated when new vulnerabilities are disclosed.

3 Challenges

Figure 1 shows that the service provider ❶ deploys a module and ❸ monitors the syscall handler. To prevent the adversary from learning whether the user secret is `X` ❷, PAVEBOX controls the syscall invocations ❷'. In this section, we model our problem as timing-sensitive non-interference and functionality preservation and identify challenges to achieve them.

Challenge 1: Modeling and enforcing non-interference. Consider a program that takes (and processes) both sensitive data and non-sensitive data as input. In the traditional setting where the program runs in a local machine, there exist both *trusted* and *untrusted* output channels. A trusted channel (e.g., local display) is securely protected from the adversary, while an untrusted channel (e.g., network session with the adversary) is observable by the adversary. A program is said to be *non-interferent* if its output to the untrusted channel is not influenced by sensitive inputs (say, private user data).

The health insurance premium calculator below takes in (i) personal information `pinfo` and (ii) desired coverage type `cov_type`. Then, it calculates the premiums of insurance plans. We assume that `pinfo` is sensitive user data here.

```
premiums = calculate(pinfo, cov_type);
send_to_user(premiums);
if (contains(pinfo, "diabetes=1"))
  for (int i=0; i<100000; i++) ;
send_to_user("Done\n");
```

Suppose that this example program runs on a user's local desktop. In this setting, the program satisfies non-interference since `send_to_user()` is a trusted output channel that is invisible to an adversary.

In our threat model for remote service, however, no data channels can be trusted since `send_to_user()` must transfer the data over the network via syscalls. Even if we conceal its payload with encryption, the occurrence of syscall remains observable. As a result, an adversary can still infer whether

[2]Although the cloud provider offers PAVE as a service, the software interfaces can be observed (e.g., audit logs such as AWS CloudWatch or network). For generality, we assume that the software interfaces are directly visible.

the condition in Line 3 is satisfied by measuring the elapsed time between the two `send_to_user()`. Therefore, we model every syscall invocation as an untrusted channel.

To ensure the absence of such implicit leakages, a module must satisfy *timing-sensitive* non-interference. A module is timing-sensitive non-interferent if it exposes a consistent sequence of syscalls (passed parameters, orders, and timings), regardless of the sensitive input values. For network syscalls, we can encrypt the content of messages (e.g., argument of `send_to_user()` in this case), so we only have to consider the order and timing of syscalls invocations.

For the timing-sensitive non-interference, we propose *shadow execution*. PaveBox runs a duplicate of the module (say, *low execution*). The low execution running in an isolated execution environment is not allowed to access any sensitive data and is given dummy values instead. It means that the syscall invocations from the low execution are not affected by the private user data. The PaveBox can control that the module outputs the same sequence of syscall invocations as the low execution. Instead of directly invoking the syscalls, the original execution of the module (say, *high execution*) delegates it to the low execution. The low execution is responsible for the syscall invocation, and the high execution reuses its corresponding return.

To enforce the shadow execution, each execution should be isolated from the others. The executions and the trusted monitor operate at the same user-level privilege within the enclave. It allows a compromised execution to bypass the monitor. The PaveBox ensures that the low and high executions cannot access each other's data and the monitor by using multi-domain Software Fault Isolation (SFI).

Timing-sensitive non-interference is one of the most restrictive privacy concepts. There are less restrictive concepts called termination-sensitive non-interference and progress-sensitive non-interference [7]. Since the external syscall handler is controlled by the service provider, it can observe the application layer interactions at any time. Thus, Pave is designed to consider timing-sensitive non-interference. Henceforth, 'non-interference' refers to timing-sensitive non-interference.

Challenge 2: Exchanging sensitive data without trusted channels. A subsequent challenge is how to enable the secure exchange of sensitive data while our shadow execution technique separates the low execution and high execution. The high execution should process the real user data and return an output to the user. However, since the high execution is restricted from directly invoking syscalls, including those for network operations, it needs a mechanism to securely acquire the user data from a network message that it cannot access directly. Although the low execution invokes syscalls to receive the network message that contains user data, note that Pave must prevent the low execution from accessing the sensitive data in the message.

For this, we introduce a *shared buffer* that is accessed in a controlled fashion by the low and high executions. Before the low execution receives an encrypted network message from the user, the PaveBox decrypts and writes it into the shared buffer. Each entry in the shared buffer is filled with both a dummy value and the real user data. The PaveBox ensures that the low execution can access only the dummy values, whereas the high execution can access the real user data. The output message to return is also produced with the same principle. The high execution can update an entry in the shared buffer with the output data derived from the user data while the low execution is blocked from accessing this output. Eventually, the response to the user is constructed with the content in the shared buffer and then transmitted through the low execution's syscall invocation.

Challenge 3: Ensuring the functionality of benign modules. If the low execution runs too early or too late, the high execution may fail to read the user data or write its output data. Therefore, a proper scheduling discipline is needed to preserve the functionality of a *benign* module. To achieve it, the low and high executions are scheduled to keep pace with each other in Pave.

Difference from Secure Multi-Execution (SME). Our shadow execution shares the intuition with SME [27] in that a given program is executed multiple times, once for each security level. One of its key assumptions is the existence of dedicated input and output channels for each security level. In contrast, our threat model assumes that the service provider can control the syscall interface, leaving *no secure channel* for the high execution.

4 Design

4.1 Shadow Execution

Given a module, the PaveBox executes its two copies running in parallel as threads. We classify syscalls into two groups: (1) those for handling user requests through the network, and (2) those for accessing other kinds of system resources. We discuss the first group of syscalls in §4.2. For the second group, we simply reuse the results of the low execution's syscalls to handle those of the high execution.

Since an enclave operates as a single process domain, we cannot rely on the OS process isolation to separate two distinct executions within it. Therefore, we use Software-Fault Isolation (SFI) by extending Native Client (NaCl) [78], a well-known SFI system based on address masking. NaCl ensures that every memory access within a module is restricted to a designated memory region.

In Pave, we further divide the module region into separate areas for high and low executions [2, 14, 79]. Each execution is assigned a designated region for data, with all data memory access adjusted to point to that region. The base address for the data region is initialized in `r14`, ensuring that each execution accesses its own data region [`r14`, `r14+4GB`),

Table 1. An execution is provided with SP APIs for secure data exchange, which is mediated by the PaveBox.

API Name	Description
`session_accept(port)`	It accepts an incoming connection. It returns a session identifier `sid`.
`session_connect(IP,port)`	It connects to a remote module using its IP and port. It returns a session identifier `sid`.
`create_msg()`	It creates an empty message and returns its identifier `mfd`.
`send_msg(sid,mfd)`	It sends the message `mfd` through the session `sid`.
`receive_msg(sid)`	It receives a message from the session `sid` and returns an identifier `mfd`.
`add_msg_entry(mfd,key,value,sec)`	It adds an entry to `mfd` with the key `key`, the sensitiveness `sec`, and the value `value`.
`get_msg_entry(mfd,key,sec)`	It returns the value of the matched entry in the `mfd` with `key` and `sec`.

through address masking. The example below shows that `rbx` is masked before it is accessed.

```
mov %ebx, %ebx # upper 32-bits are cleared
add %r14, %rbx
mov %rax, (%rbx)
```

Specifically, the following rules `E1`–`E6` are applied to ensure secure data access isolated for each execution.

- `E1`: The memory space for each execution consists of a *shared* code region and a *private* data region.
- `E2`: The base addresses for the code and data regions are initialized in `r15` and `r14`, respectively. These registers must remain unmodifiable.
- `E3`: All indirect data access instructions should use `r14` for its address to be in the range [`r14`, `r14+4GB`).
- `E4`: `rsp` and `rbp` can be modified by copying each other without masking. Otherwise, they should be masked.
- `E5`: `rip`-relative addressing is disallowed.
- `E6`: Trampoline code securely transfers control between an execution and a PaveBox. The code is attested by SGX.

Additionally, Pave adheres to the control transfer policies from NaCl as follows.

- `N1`: The code section is divided into 32-byte aligned code bundles. Neither individual instructions nor a series of instructions modified for address masking cannot cross a bundle boundary.
- `N2`: All indirect control flow instructions are masked to ensure that the target address is aligned with a 32-byte bundle boundary. (e.g., `and %eax, 0xffffffe0; jmp *%eax;`)
- `N3`: `ret`, `syscall`, and `int` instructions are forbidden.

Forbidden `syscall` instructions (Rule `N3`) are replaced with the trampoline code (Rule `E6`). The trampoline mechanism invokes a context switch function to the PaveBox, which saves the current execution context and loads the context of PaveBox. The PaveBox then handles the syscall based on the shadow execution. Afterwards, the context is restored, switching back to the original execution. We provide a toolchain to (1) generate a PaveBox-compliant binary and (2) verify such compliance before running it (detailed in §4.4).

4.2 Secure Data Exchange

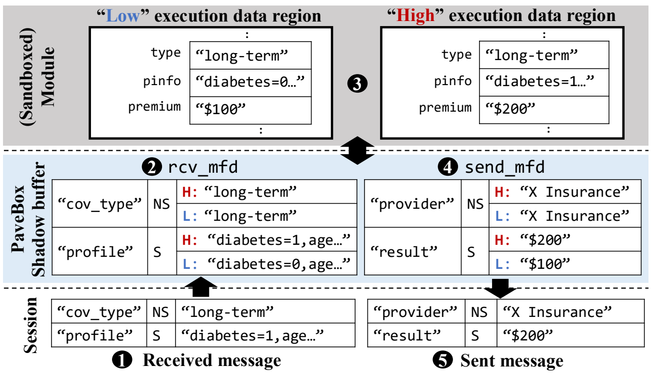
Shadow proxy (SP) is responsible for sensitive data exchange via the shared buffer and secure sessions. It offers the module with SP APIs listed in Table 1. Note that both low and high executions are prevented from directly accessing the secure session or shared buffer (Rule `E3`). Instead, the module must invoke SP APIs, which is implemented through the secure control transfer (Rule `E6`). Then, the SP will intercept the calls and handle the requests securely. Suppose a scenario where a single module processes a user request. When `session_accept()` is called, it is the shadow proxy that establishes a session with the user. Since the user verifies the integrity of the PaveBox through the remote attestation, the module cannot impersonate the shadow proxy and establish the session. We employ TLS with SGX extension [45] for this.

Distributed Modules. When multiple modules collaborate, user data should not be forwarded to any insecure module (i.e., not controlled by Pave). To achieve this, a client-side PaveBox, upon handling `session_connect`, performs the remote attestation of a server-side PaveBox. They cannot mutually authenticate since the user who is not equipped with Intel SGX cannot be verified by remote attestation. Instead, a pair of high and low executions is allowed to accept only one incoming session from either a user or another execution pair. Meanwhile, there is no restriction on the number of outgoing sessions. That is, the user first verifies a PaveBox through remote attestation when sending a request. Subsequently, the verified PaveBox iteratively verifies other PaveBoxes. Consequently, this chained remote attestation can guarantee that the user data is only sent and received to the modules controlled by Pave. Note that the PaveBox can still support multi-threading and run multiple pairs of executions, each of which handles a single request at a time.

Network Message Format. Pave defines a formatted network message that can be interpreted by the PaveBox. The message consists of multiple entries, and each of the entries has three components: key, value, and security level. The *key* is an index of the entry and the *value* contains the data of the entry. The *security level* indicates whether the entry's data is sensitive (`S`) or non-sensitive (`NS`).

```
1  int main() {
2    ...
3    sid = session_accept(PORT_NUM);
4    rcv_mfd = receive_msg(sid);                                  ❶→❷
5    type = get_msg_entry(rcv_mfd, "cov_type", NS);                 ❸
6    pinfo = get_msg_entry(rcv_mfd, "profile", S);                  ❸
7    premium = calculate(pinfo, type);                              ❸
8    send_mfd = create_msg();
9    // Line 10-11 breaks non-interference
10   if (contains(pinfo, "diabetes=1"))
11      for (int i=0; i<10000; i++) ;
12   add_msg_entry(send_mfd, "provider", "X Insurance", NS);        ❹
13   add_msg_entry(send_mfd, "result", premium, S);                 ❹
14   send_msg(sid, send_mfd);                                     ❹→❺
15   return 0;
16 }
```

(a) Example code.


"Low" execution data region
"High" execution data region
(Sandboxed) Module
type
pinfo
premium
"long-term"
"diabetes=0…"
"$100"
"diabetes=1…"
"$200"
❸
PaveBox Shadow buffer
❷ rcv_mfd
❹ send_mfd
"cov_type" NS H: "long-term" L: "long-term"
"profile" S H: "diabetes=1,age…" L: "diabetes=0,age…"
"provider" NS H: "X Insurance" L: "X Insurance"
"result" S H: "$200" L: "$100"
Session
"cov_type" NS "long-term"
"profile" S "diabetes=1,age…"
"provider" NS "X Insurance"
"result" S "$200"
❶ Received message
❺ Sent message

(b) State changes in different areas.

Figure 2. The example program and corresponding state changes of the shared buffer and the module data regions are shown.

The SP APIs allow an execution to send or receive messages and to read or write entries within these messages (under the control of the PaveBox). In low execution, send_msg and receive_msg internally trigger network syscalls, whereas high execution simply ignores these API calls. Meanwhile, add_msg_entry and get_msg_entry enable high and low executions to access the data authorized for each level. Figure 2 illustrates an example where the module processes a user request for the insurance premium calculation service. When a message is received (Line 4), the SP replaces the value of each entry with a pair of fields (L and H) and stores it in a shared buffer (from ❶ to ❷)[3]. For S entries, the SP populates the L field with a predefined dummy value and the H field with the actual user data. For NS entries, both fields are filled with the actual user data. Note that there is no strict requirement on how these dummy values are set, as long as they are determined independently of any sensitive values.

The message is identified by an abstract descriptor (rcv_mfd in Figure 2). Using this descriptor, the low execution is allowed to access only the L field, while the high execution can access only the H field (Line 5–6 and ❸ in Figure 2). In Line 7, the calculated premium can vary depending on type and pinfo (say 100 and 200 for example, as shown in ❸ of Figure 2).

To send a response, the executions create a new message (Line 8) and populate the corresponding H or L fields (Line 12–13; ❹ in Figure 2b). When the message is sent, the SP extracts an appropriate field for each entry (from ❹ to ❺ in Figure 2b.) For S entries, H fields are used, while L fields are used for NS entries. Extracting H fields for S entries is necessary since these fields contain the actual outputs computed from the user data. For NS entries, PAVE must avoid extracting the H values computed from the high execution. Otherwise, if the populated message is passed to another module (not the user), the receiving module can leak the private data produced in the high execution. Note that when sending a message, the SP ensures that its length is set to a pre-determined constant value to prevent traffic analysis.

As shown in §3, this program violates non-interference because the timing of syscall is affected by sensitive data pinfo. However, in Pave, send_msg only triggers syscall in low execution, and the branch condition in Line 10 is always unsatisfied by the dummy value used in the low execution. Thus, the timing of syscall invocation remains consistent. Additionally, when the *high* execution finishes in Line 15, PaveBox intercepts its exit syscall and delays its termination. Only when the low execution finishes, PaveBox intercepts its exit syscall and terminates the both executions. Note that there is a chance that the output of high execution is not properly sent back to the user. As explained earlier, Pave does not guarantee the functionality of insecure programs.

4.3 Scheduling

To enforce non-interference in the execution under Pave, the scheduler for low and high execution must be *input-agnostic*. We can model the result of scheduling as a sequence of L and H (e.g., [L, H, H, ...]). Input-agnostic scheduling means that such result is not affected by the sensitive user data. For the example in Figure 2, let us assume a scheduler that makes the low execution wait at send_msg in Line 14 until the high execution reaches the same API call. This scheduler is *not* input-agnostic, as the resulting scheduling sequence will depend on the profile field of the input. Not surprisingly, the execution under such scheduler will leak the private data through the timing of syscall. One safe scheduling that Pave can adopt is simply letting the OS scheduler to choose between the two threads for low and high execution. Since the OS cannot access the SGX enclave, it cannot inspect memory data or program counter of these executions. Therefore, this scheduler can be considered input-agnostic.

Meanwhile, to preserve the functionality of secure programs, using input-agnostic scheduler is not enough. For

[3] As an alternative design, the user agent can populate these pairs instead of the SP when it is sent. Hence, dummy values are not required in PaveBox.

example, let us assume that Line 10 in Figure 2 is changed to `if (type == "long_term")` and uses non-sensitive data instead. Now the program is non-interferent and Pave must preserve its functionality. For this, the two executions must keep pace with each other. Intuitively, both executions must reach Line 14 at the same time to ensure that the response message is correctly filled in by the high execution before the low execution sends it.

PaveBox achieves this by synchronizing two executions in terms of the number of executed basic blocks. For this, our toolchain instruments the module during compilation by inserting a synchronization instruction in every basic block. Note that each syscall, implemented as a wrapper function, is also instrumented with a synchronization instruction. When this synchronization instruction is executed, the control of the running thread is transferred to the PaveBox. The PaveBox will check how many basic blocks have been executed by low and high execution respectively, and chooses the next thread to execute in a similar manner to HyperRace [21]. Intuitively, the result of our synchronized scheduler is *fixed* as an alternate sequence of L and H ([`L`, `H`, `L`, `H`, ...]), where the execution unit is a basic block. Therefore, this scheduler is also input-agnostic. In §5, we formally model and prove the non-interference and functionality preservation under appropriate schedulers.

4.4 PaveBox Initialization

To load a module, the service provider sends its encrypted binary to the PaveBox via TLS [45]. The PaveBox loader decrypts it inside the enclave, keeping it confidential even from the cloud provider and preventing analysis of its behavior.

Before loading the binary, we need to ensure compliance with SFI (§4.1) and synchronization (§4.3). First, as in NaCl, the PaveBox validator statically analyzes every instruction to confirm that only legal instructions are present. The PaveBox will load the binary only if it passes the SFI validation. Next, the validator checks whether each basic block in the binary is instrumented with a synchronization instruction. If the verification fails, PaveBox will ignore synchronization instructions during the execution, to prevent the scheduling of PaveBox from behaving in an unexpected way. When the synchronization is disabled, the execution under PaveBox falls back on the OS scheduler discussed in §4.3. Although sound and complete identification of basic blocks is impossible in general setting, Pave only requires a sound validator that conservatively rejects uncertain binaries. The completeness of the validator can be enhanced in the future by employing state-of-the-art binary analysis techniques [28, 44, 59, 94].

5 Security Analysis

Figure 3 presents the syntax of our model language. For brevity, we assume that a module has only one network session that is already established, and manipulates only one network message at a time. As a result, SP APIs in Table 1 are simplified as follow. First, `recv` (which models `receive_msg`) reads a message from the session, while `send` (for `send_msg`) transmits a message to the user. In this section, we will refer to `recv` and `send` as syscalls, as the SP will internally invoke syscalls for these statements. Also, `get` (for `get_msg_entry`) reads the value for key k from the message and `add` (for `add_msg_entry`) updates the value for key k in the message. We assume that the set of keys for sensitive and non-sensitive entry is predefined as K_S and K_{NS} respectively.

$$
\begin{array}{lcl}
\texttt{Command}\ c & ::= & \texttt{recv} \\
 & | & \texttt{send} \\
 & | & \texttt{get}\ k\ \texttt{to}\ x \\
 & | & \texttt{add}\ x\ \texttt{to}\ k \\
 & | & \texttt{skip} \\
 & | & x := e \\
 & | & c; c \\
 & | & \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \\
 & | & \texttt{while}\ e\ \texttt{do}\ c
\end{array}
$$

Figure 3. The syntax of our model language is presented.

$$(S1)\quad \frac{I = [a_1, a_2, ..., a_n] \quad I' = [a_2, ..., a_n]}{\langle \texttt{recv}, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m, a_1, I', O\rangle}$$

$$(S2)\quad \frac{O = [a_1, ..., a_n] \quad O' = [a_1, ..., a_n, a]}{\langle \texttt{send}, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m, a, I, O'\rangle}$$

$$(S3)\quad \frac{a(k) = v \quad m' = m[x \mapsto v]}{\langle \texttt{get}\ k\ \texttt{to}\ x, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m', a, I, O\rangle}$$

$$(S4)\quad \frac{m(x) = v \quad a' = a[k \mapsto v]}{\langle \texttt{add}\ x\ \texttt{to}\ k, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m, a', I, O\rangle}$$

$$(S5)\quad \frac{eval(e, m) = v \quad m' = m[x \mapsto v]}{\langle x := e, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m', a, I, O\rangle}$$

$$(S6)\quad \frac{}{\langle \texttt{skip}; c_2, m, a, I, O\rangle \rightarrow \langle c_2, m, a, I, O\rangle}$$

$$(S7)\quad \frac{\langle c_1, m, a, I, O\rangle \rightarrow \langle c_1', m', a', I', O'\rangle}{\langle c_1; c_2, m, a, I, O\rangle \rightarrow \langle c_1'; c_2, m', a', I', O'\rangle}$$

$$(S8)\quad \frac{eval(e, m) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, a, I, O\rangle \rightarrow \langle c_1, m, a, I, O\rangle}$$

$$(S9)\quad \frac{eval(e, m) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, a, I, O\rangle \rightarrow \langle c_2, m, a, I, O\rangle}$$

$$(S10)\quad \frac{eval(e, m) \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, a, I, O\rangle \rightarrow \langle c; \texttt{while}\ e\ \texttt{do}\ c, m, a, I, O\rangle}$$

$$(S11)\quad \frac{eval(e, m) = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m, a, I, O\rangle \rightarrow \langle \texttt{skip}, m, a, I, O\rangle}$$

Figure 4. Standard execution semantics without PaveBox.

5.1 Standard Semantics

We define the standard semantics as transition rules of an execution state $\langle c, m, a, I, O\rangle$. Here, c represents the commands to be executed and memory m is a mapping from variable to value. Key-value map a represents a network message stored in the shared buffer. Input I denotes the list of messages that will be provided to the module and output O denotes the accumulated list of the transmitted messages.

Figure 4 presents relation $\rightarrow$ for the transition rules of an execution state. This standard semantics describes the original behavior of the module without PaveBox. Function $eval(e, m)$ evaluates the given expression e under the memory state m, and $m[k \mapsto v]$ denotes a map obtained by updating m to have value v for key k. We consider $m(k)$ to return $\perp$ if key k is not found in map m. Rule (S1) models the receiving of input message into the shared buffer, while (S2) models the transmission of the current message stored in the shared buffer. Although our simplified model processes only one network message at a time, it is straightforward to extend this and support multiple messages distinguished by descriptors, as in §4.2.

5.2 PaveBox Semantics

Figure 5 defines relation $\Rightarrow$ for the *local* semantics for the low or high execution. Local state $\langle c, m\rangle_l$ denotes remaining commands c and memory m for the execution level l (i.e., L or H). Other components such as a, I, and O are shared by the two executions. The syscall operations are delegated to the low execution, as described in (L1) to (L4). To define the low execution semantics for `recv`, we introduce a pre-processing function that generates a pair of values for each message entry. For the key k and value v of an entry, we define $pair(k, v)$ as $\langle v, v\rangle$ if $k \in K_{NS}$ and $\langle v_{def}(k), v\rangle$ if $k \in K_S$, where $v_{def}(k)$ is the pre-determined dummy value for k. We can also extend this for a network message a as $\{(k, v') \mid (k, v) \in a,\ v' = pair(k, v)\}$. The pre-processed message is then stored in the shared buffer. For `send`, the low execution should construct an output message by extracting proper fields from the message in the shared buffer. We define a function $ext(k, \langle v_L, v_H\rangle)$ as v_L if $k \in K_{NS}$ and v_H if $k \in K_S$ and extend it over a message a, by defining $ext(a)$ as $\{(k, v') \mid (k, v) \in a,\ v' = ext(k, v),\ v' \neq \perp\}$. (L5) to (L7) describe controlled access to the message in the shared buffer based on the execution level.

Figure 6 defines the global execution semantics over both executions. (G1a) and (G1b) define the global execution semantics when the scheduling result is provided as s. Here, counter n is initialized to 0 and incremented per every local transition. As discussed in §4.3, the scheduling result s is a sequence of `L` and `H`, and $s[n]$ tells which execution level will be taken. Note that the global executions will stop when the low execution terminates, since (L11) delays the termination of high execution. Next, (G2) defines relation $\rightsquigarrow$, which the execution under the synchronized scheduling of PaveBox. In (G2), low and high executions are executed alternately, so this can be thought as a special case of (G1a) and (G1b) where $s = [L, H, L, H, \ldots]$. Note that our formal

$$(L1)\qquad \frac{I = [a_1, a_2, \ldots, a_n] \quad I' = [a_2, \ldots, a_n]}{\langle\langle \mathsf{recv}, m\rangle_L, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_L, pair(a_1), I', O\rangle}$$

$$(L2)\qquad \frac{}{\langle\langle \mathsf{recv}, m\rangle_H, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_H, a, I, O\rangle}$$

$$(L3)\qquad \frac{O = [a_1, \ldots, a_n] \quad O' = [a_1, \ldots, a_n, ext(a)]}{\langle\langle \mathsf{send}, m\rangle_L, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_L, a, I, O'\rangle}$$

$$(L4)\qquad \frac{}{\langle\langle \mathsf{send}, m\rangle_H, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_H, a, I, O\rangle}$$

$$(L5)\qquad \frac{a(k) = \langle v_L, v_H\rangle \quad m' = m[x \mapsto v_l]}{\langle\langle \mathtt{get}\ k\ \mathtt{to}\ x, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m'\rangle_l, a, I, O\rangle}$$

$$(L6)\qquad \frac{m(x) = v \quad a(k) = \langle v_L, v_H\rangle \quad a' = a[k \mapsto \langle v, v_H\rangle]}{\langle\langle \mathtt{add}\ x\ \mathtt{to}\ k, m\rangle_L, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_L, a', I, O\rangle}$$

$$(L7)\qquad \frac{m(x) = v \quad a(k) = \langle v_L, v_H\rangle \quad a' = a[k \mapsto \langle v_L, v\rangle]}{\langle\langle \mathtt{add}\ x\ \mathtt{to}\ k, m\rangle_H, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_H, a', I, O\rangle}$$

$$(L8)\qquad \frac{eval(e, m) = v \quad m' = m[x \mapsto v]}{\langle\langle x := e, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m'\rangle_l, a, I, O\rangle}$$

$$(L9)\qquad \frac{}{\langle\langle \mathsf{skip}; c_2, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle c_2, m\rangle_l, a, I, O\rangle}$$

$$(L10)\qquad \frac{c_1 \neq \mathsf{skip} \quad \langle\langle c_1, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle c_1', m'\rangle_l, a', I', O'\rangle}{\langle\langle c_1; c_2, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle c_1'; c_2, m'\rangle_l, a', I', O'\rangle}$$

$$(L11)\qquad \frac{}{\langle\langle \mathsf{skip}, m\rangle_H, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_H, a, I, O\rangle}$$

$$(L12)\qquad \frac{eval(e, m) \neq 0}{\begin{gathered}\langle\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m\rangle_l, a, I, O\rangle \Rightarrow \\ \langle\langle c_1, m\rangle_l, a, I, O\rangle\end{gathered}}$$

$$(L13)\qquad \frac{eval(e, m) = 0}{\begin{gathered}\langle\langle \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, m\rangle_l, a, I, O\rangle \Rightarrow \\ \langle\langle c_2, m\rangle_l, a, I, O\rangle\end{gathered}}$$

$$(L14)\qquad \frac{eval(e, m) \neq 0}{\begin{gathered}\langle\langle \mathtt{while}\ e\ \mathtt{do}\ c, m\rangle_l, a, I, O\rangle \Rightarrow \\ \langle\langle c; \mathtt{while}\ e\ \mathtt{do}\ c, m\rangle_l, a, I, O\rangle\end{gathered}}$$

$$(L15)\qquad \frac{eval(e, m) = 0}{\langle\langle \mathtt{while}\ e\ \mathtt{do}\ c, m\rangle_l, a, I, O\rangle \Rightarrow \langle\langle \mathsf{skip}, m\rangle_l, a, I, O\rangle}$$

Figure 5. Local semantics under PaveBox.

$$(G1a)\qquad \frac{s[n] = L \quad \langle\langle c_1, m_1\rangle_L, a, I, O\rangle \Rightarrow \langle\langle c_1', m_1'\rangle_L, a', I', O'\rangle}{\begin{gathered}\langle\langle c_1, m_1\rangle, \langle c_2, m_2\rangle, a, I, O, n\rangle \overset{s}{\dashrightarrow} \\ \langle\langle c_1', m_1'\rangle, \langle c_2, m_2\rangle, a', I', O', n+1\rangle\end{gathered}}$$

$$(G1b)\qquad \frac{s[n] = H \quad \langle\langle c_2, m_2\rangle_H, a, I, O\rangle \Rightarrow \langle\langle c_2', m_2'\rangle_H, a', I', O'\rangle}{\begin{gathered}\langle\langle c_1, m_1\rangle, \langle c_2, m_2\rangle, a, I, O, n\rangle \overset{s}{\dashrightarrow} \\ \langle\langle c_1, m_1\rangle, \langle c_2', m_2'\rangle, a', I', O', n+1\rangle\end{gathered}}$$

$$(G2)\qquad \frac{\begin{gathered}\langle\langle c_1, m_1\rangle_L, a, I, O\rangle \Rightarrow \langle\langle c_1', m_1'\rangle_L, a', I', O'\rangle \\ \langle\langle c_2, m_2\rangle_H, a', I', O'\rangle \Rightarrow \langle\langle c_2', m_2'\rangle_H, a'', I'', O''\rangle\end{gathered}}{\langle\langle c_1, m_1\rangle, \langle c_2, m_2\rangle, a, I, O\rangle \rightsquigarrow \langle\langle c_1', m_1'\rangle, \langle c_2', m_2'\rangle, a'', I'', O''\rangle}$$

Figure 6. Global semantics under PaveBox.

model synchronizes the execution with a granularity of one local transition step. In §7, we discuss the difference between our model and implementation that comes from this point.

5.3 Security Properties

For any transition $\hookrightarrow$, we define $\hookrightarrow^n$ to mean a transition by applying $\hookrightarrow$ for n times. We abuse $\hookrightarrow^n$ by defining it over the essential input and output components. With standard semantics on program P and input I, if $\langle P, m_0, a_0, I, O_0\rangle \rightarrow^n \langle c, m, a, I', O'\rangle$ holds where m_0 and a_0 are empty maps and O_0 is an empty list, we also say that $(P, I) \rightarrow^n (I', O')$ holds. Next, we introduce the equivalence between two inputs (or two outputs). For messages a and a', we say $a{=_{NS}}a'$ iff $dom(a) = dom(a')$ and $\forall k \in dom(a) \cap K_{NS}$, $a(k) = a'(k)$. Similarly, we say $a{=_S}a'$ iff $dom(a) = dom(a')$ and $\forall k \in dom(a) \cap K_S$, $a(k) = a'(k)$. Now, we extend $=_{NS}$ on message lists $A = [a_1, ..., a_n]$ and $A' = [a'_1, ..., a'_m]$. We say $A{=_{NS}}A'$ iff $len(A) = len(A')$ and $\forall i. 1 \le i \le n$, $a_i{=_{NS}}a'_i$. We can extend $=_S$ in the same way.

Definition 1 (Non-interference). Assume a program P, semantics $\hookrightarrow$, and two inputs I_1, I_2 such that $I_1{=_{NS}}I_2$. Non-interference of P under $\hookrightarrow$ means that $\forall n \ge 0$, if $(P, I_1) \hookrightarrow^n (I'_1, O'_1)$, then $(P, I_2) \hookrightarrow^n (I'_2, O'_2)$, $I'_1{=_{NS}}I'_2$ and $O'_1{=_{NS}}O'_2$ hold.

This definition implies $len(I'_1) = len(I'_2)$ and $len(O'_1) = len(O'_2)$. This captures the consistent syscall pattern of the two executions. For example, if there exists n that makes $len(O'_1) \ne len(O'_2)$, it means that the number of send syscalls observed in two executions do not match at this time point.

Now we define the non-interference property under an input-agnostic scheduling (§4.3). Input-agnostic scheduler must output the same sequence for I_1 and I_2 in Definition 1. We let this arbitrary sequence as s in the following theorem.

Theorem 1 (Non-interference of PAVEBOX). For any program P and sequence s, P is non-interferent under $\overset{s}{\dashrightarrow}$.

Next, we define the functionality preservation of PAVE. PAVEBOX should not amend the execution output of a module that is non-inteferent under the standard semantics.

Theorem 2 (Transparency). Assume a program P that is non-interferent under $\rightarrow$. Now, $\forall n \ge 0$, if $(P, I) \rightarrow^n (I_1, O_1)$, then $(P, I) \rightsquigarrow^n (I_2, O_2)$ holds where $O_1 = O_2$.

5.4 Proof of Non-Interference

First, we introduce equivalence over the L fields (or H fields). For two tuples $v = \langle v_L, v_H\rangle$ and $v' = \langle v'_L, v'_H\rangle$, we say $v{=_L}v'$ iff $v_L = v'_L$. Also, we say $v{=_H}v'$ iff $v_H = v'_H$. We extend this over two messages a and a' in the shared buffer. We say $a{=_L}a'$ iff $\forall k \in dom(a) \cup dom(a')$, $a(k){=_L}a'(k)$. We also extend $=_H$ over messages in the same way.

From the definition of $=_{NS}$, $=_L$, and $pair()$, it is trivial to show that for two input messages a and a', if $a{=_{NS}}a'$, then $pair(a){=_L}\ pair(a')$ holds. Also, we can show that for two messages a and a' stored in the shared buffer, if $a{=_L}a'$, then $ext(a){=_{NS}}ext(a')$ holds.

Now we introduce two lemmas to prove Theorem 1.

Lemma 1 (Equivalence in Low Execution). Assume two transitions $\langle\langle c, m\rangle_L, a_1, I_1, O_1\rangle \Rightarrow \langle\langle c'_1, m'_1\rangle_L, a'_1, I'_1, O'_1\rangle$ and $\langle\langle c, m\rangle_L, a_2, I_2, O_2\rangle \Rightarrow \langle\langle c'_2, m'_2\rangle_L, a'_2, I'_2, O'_2\rangle$, where $a_1{=_L}a_2$, $I_1{=_{NS}}I_2$, and $O_1{=_{NS}}O_2$ hold. Then, $c'_1 = c'_2$, $m'_1 = m'_2$, $a'_1{=_L}a'_2$, $I'_1{=_{NS}}I'_2$, and $O'_1{=_{NS}}O'_2$ hold.

Lemma 2 (Confinement of High Execution). Assume a transition $\langle\langle c, m\rangle_H, a, I, O\rangle \Rightarrow \langle\langle c', m'\rangle_H, a', I', O'\rangle$. Then, $a{=_L}a'$, $I = I'$ and $O = O'$ hold.

Intuitively, Lemma 1 means that if two low execution states have the same L fields in shared buffer and NS entries in messages, then this equivalence is maintained after one step of low execution. Lemma 2 states that a high execution step cannot break this equivalence as well.

Proof of Lemma 1. We can prove Lemma 1 and Lemma 2 by examining the semantic rules in Figure 5. First, we can see that transition of a local state from $\langle c, m\rangle_L$ to $\langle c'_i, m'_i\rangle_L$ in the lemma is always decided by c, m, and the L fields of a_i. Since $a_1{=_L}a_2$, we can prove $c'_1 = c'_2$ and $m'_1 = m'_2$.

Now we consider the semantic rules that can affect the global state a_i, I_i, and O_i. To start with, when c is add x to k, it can update the L fields in a_i. Still, we know that a_1 and a_2 will always be updated with the same value, $m(x)$. Therefore, $a'_1{=_L}a'_2$ holds when c is add x to k. Next, recv command can also update the global state. Recall that if $a{=_{NS}}a'$, then $pair(a){=_L}pair(a')$ holds. Using this property and $I_1{=_{NS}}I_2$ in assumption, we can show $a'_1{=_L}a'_2$ and $I'_1{=_{NS}}I'_2$ hold by examining (L1). Similarly, we have shown that if $a{=_L}a'$, then $ext(a){=_{NS}}ext(a')$ holds. Thus, when c is send, we can use this property and $a_1{=_L}a_2$ to show that $O'_1{=_{NS}}O'_2$ by examining (L3).

Proof of Lemma 2. We can also prove this lemma by examining the high execution's semantic rules in Figure 5. In the high execution, the semantic rules prevent any update on I and O. Also, rule (L7) only allows updates to the H fields of a. Therefore, $a{=_L}a'$ holds. Consequently, Lemma 2 holds.

Proof of Theorem 1. Suppose two inputs I_1 and I_2 such that $I_1{=_{NS}}I_2$ holds. We will prove the following property that subsumes Theorem 1: $\forall n \ge 0$, if $\langle S_0, S_0, a_0, I_1, O_0, 0\rangle \overset{s}{\dashrightarrow}^n \langle L'_1, H'_1, a'_1, I'_1, O'_1, n\rangle$ then $\langle S_0, S_0, a_0, I_2, O_0, 0\rangle \overset{s}{\dashrightarrow}^n \langle L'_2, H'_2, a'_2, I'_2, O'_2, n\rangle$, where $L'_1 = L'_2$, $a'_1{=_L}a'_2$, $I'_1{=_{NS}}I'_2$, and $O'_1{=_{NS}}O'_2$. S_0 is the initial local state $\langle P, m_0\rangle$, where P in an input program.

First, this property trivially holds when $n = 0$. Next, we can prove that if this property holds for $n = k$, it also holds for $n = k + 1$ by Lemma 1 and Lemma 2. When $s[k] = \mathtt{L}$, Lemma 1 directly proves that $L'_1 = L'_2$, $a'_1{=_L}a'_2$, $I'_1{=_{NS}}I'_2$, and $O'_1{=_{NS}}O'_2$ hold for $n = k + 1$. Meanwhile when $s[k] = \mathtt{H}$, we can first see $L'_1 = L'_2$ holds for $n = k + 1$ from rule (G1b) in Figure 6. Also, Lemma 2 states that I and O do not change in the high execution step, so $I'_1{=_{NS}}I'_2$ and $O'_1{=_{NS}}O'_2$ also hold for $n = k + 1$. In addition, Lemma 2 also shows that $=_L$ holds between the two a'_1 for $n = k$ and $n = k + 1$. The same holds for a'_2 as well. Thus, $a'_1{=_L}a'_2$ holds for $n = k + 1$, from the induction hypothesis and the transitivity of $=_L$. Therefore, the property above is proved by induction on n.

5.5 Proof of Transparency

First, we define correspondence between an input message and a message in the shared buffer. Assume a raw value v and a pre-processed pair $v' = \langle v_L, v_H \rangle$. We say $v \simeq_L v'$ iff $v = v_L$ and $v \simeq_H v'$ iff $v = v_H$. Next, we define $\simeq_L$ between an input message a and a message stored in the shared buffer, a'. We say $a \simeq_L a'$ iff $\forall k \in dom(a) \cup dom(a'), a(k) \simeq_L a'(k)$ holds. We can also extend $\simeq_H$ in the same way.

Next, we define a replacement function for sensitive entries of an input message. We first define $rep(k, v)$ as $v_{def}(k)$ if $k \in K_S$ and v if $k \in K_{NS}$. Then, we can extend this for an input network message or message list, as we did for $pair()$.

From the definition of $\simeq_L$, $rep()$, and $pair()$, we can easily show that $rep(a) \simeq_L pair(a)$ holds. Likewise, from the definition of $\simeq_H$ and $rep()$, we can show that $a \simeq_H pair(a)$ holds. Additionally, by combining the definition of $\simeq_L$, $=_{NS}$, and $ext()$, we can show that if $a \simeq_L a'$ then $a =_{NS} ext(a')$ holds. Similarly, we can show that if $a \simeq_H a'$ then $a =_S ext(a')$ holds.

Lastly, we define $=_{sys}$ for c_1 and c_2. First, we say $c_1 =_{rcv} c_2$ iff (i) $c_1 \neq$ `recv` and $c_2 \neq$ `recv` or (ii) $c_1 = c_2 =$ `recv`. Similarly, we will say $c_1 =_{snd} c_2$ iff (i) $c_1 \neq$ `send` and $c_2 \neq$ `send` or (ii) $c_1 = c_2 =$ `send`. Finally, we say $c_1 =_{sys} c_2$ iff $c_1 =_{rcv} c_2$ and $c_1 =_{snd} c_2$.

Now we introduce two lemmas to prove Theorem 2.

Lemma 3 (Correspondence to Low Execution). Assume a transition step with the standard semantics $\langle c, m, a_1, I_1, O_1 \rangle \rightarrow \langle c_1', m_1', a_1', I_1', O_1' \rangle$ and a global transition step with the PaveBox semantics $\langle \langle c, m \rangle, \langle c_H, m_H \rangle, a_2, I_2, O_2 \rangle \rightsquigarrow \langle \langle c_2', m_2' \rangle, \langle c_H', m_H' \rangle, a_2', I_2', O_2' \rangle$. Now, if $a_1 \simeq_L a_2$, $I_1 = rep(I_2)$, and $O_1 =_{NS} O_2$, then $c_1' = c_2'$, $m_1' = m_2'$, $a_1' \simeq_L a_2'$, $I_1' = rep(I_2')$, and $O_1' =_{NS} O_2'$.

Lemma 4 (Correspondence to High Execution). Assume a transition with the standard semantics $\langle c, m, a_1, I_1, O_1 \rangle \rightarrow \langle c_1', m_1', a_1', I_1', O_1' \rangle$ and a global transition with the PaveBox semantics $\langle \langle c_L, m_L \rangle, \langle c, m \rangle, a_2, I_2, O_2 \rangle \rightsquigarrow \langle \langle c_L', m_L' \rangle, \langle c_2', m_2' \rangle, a_2', I_2', O_2' \rangle$. Now, if $a_1 \simeq_H a_2$, $I_1 =_S I_2$, $O_1 =_S O_2$, and $c =_{sys} c_L$, then $c_1' = c_2'$, $m_1' = m_2'$, $a_1' \simeq_H a_2'$, $I_1' =_S I_2'$, and $O_1' =_S O_2'$.

Intuitively, Lemma 3 means that if there is a correspondence between a standard execution and a low execution, it is maintained after a single step. Lemma 4 describes the preservation of the correspondence between a standard execution and a high execution.

Proof of Lemma 3. Based on (G2) of Figure 6, we first split the global transition $\rightsquigarrow$ in Lemma 3 into two local transitions: (i) $\langle \langle c, m \rangle_L, a_2, I_2, O_2 \rangle \Rightarrow \langle \langle c_2', m_2' \rangle_L, \tilde{a}_2, I_2', O_2' \rangle$ and (ii) $\langle \langle c_H, m_H \rangle_H, \tilde{a}_2, I_2', O_2' \rangle \Rightarrow \langle \langle c_H', m_H' \rangle_H, a_2', I_2', O_2' \rangle$. Note that the second transition is not making any change to I_2' and O_2', as proved in previous Lemma 2.

First, we consider the low execution step (i). We know that the transition of c and m in the standard execution is decided by c, m, and a_1. Meanwhile, in the low execution, the transition of $\langle c, m \rangle$ is decided by c, m, and the `L` fields of a_2. From $a_1 \simeq_L a_2$, we can see that c, m in the standard execution and c, m in the low execution always change in the same way. Thus, $c_1' = c_2'$ and $m_1' = m_2'$ hold.

Next, we will prove that $a_1' \simeq_L \tilde{a}_2$, $I_1' = rep(I_2')$, and $O_1' =_{NS} O_2'$. If c does not affect shared buffer or network messages (i.e., a_1, a_2, I_1, I_2, O_1, O_2), then these properties are directly satisfied. Thus, we will examine commands that can modify these message-related components. First, when c is `add` x `to` k, the standard execution updates a_1 with $m(x)$ while the low execution updates `L` field of a_2 with $m(x)$. Therefore, $a_1' \simeq_L a_2'$ holds. Next, let us assume c is `recv`. Recall that we have shown $rep(a) \simeq_L pair(a)$. With this property and $I_1 = rep(I_2)$, we can prove that $a_1' \simeq_L \tilde{a}_2$ from rule (S1) and (L1). Also, these rules remove the first element in I_1 and I_2 respectively, so $I_1' = rep(I_2')$ holds, too. Lastly, when c is `send`, we will use the fact that if $a \simeq_L a'$ then $a =_{NS} ext(a')$ holds. With this property and $a_1 \simeq_L a_2$, we can show that $O_1' =_{NS} O_2'$ holds from rule (S2) and (L3).

Now, we move on to the high execution step (ii). From Lemma 2, $\tilde{a}_2 =_L a_2'$ holds. Since we have proven $a_1' \simeq_L \tilde{a}_2 =_L a_2'$, we can see that $a_1' \simeq_L a_2'$.

Proof of Lemma 4. As we did in the proof of Lemma 3, we split the transition with global semantics into two local transitions: (i) $\langle \langle c_L, m_L \rangle_L, a_2, I_2, O_2 \rangle \Rightarrow \langle \langle c_L', m_L' \rangle_L, \tilde{a}_2, I_2', O_2' \rangle$ and (ii) $\langle \langle c, m \rangle_H, \tilde{a}_2, I_2', O_2' \rangle \Rightarrow \langle \langle c_2', m_2' \rangle_H, a_2', I_2', O_2' \rangle$.

We start by proving $I_1' =_S I_2'$ and $O_1 =_S O_2'$. First, we will prove $I_1' =_S I_2'$. When c is not `recv`, neither is c_L, from $c =_{sys} c_L$. Then, $I_1 = I_1'$ and $I_2 = I_2'$, so $I_1' =_S I_2'$. When c is `recv`, so is c_L, and the first element is removed respectively from I_1 and I_2. Since $I_1 =_S I_2$, we can see $I_1' =_S I_2'$. Next, we will prove $O_1 =_S O_2'$. When c is not `send`, neither is c_L, from $c =_{sys} c_L$. Then, $O_1 = O_1'$ and $O_2 = O_2'$, so $O_1' =_S O_2'$. When c is `send`, so is c_L, and we will use the fact that if $a \simeq_H a'$ then $a =_S ext(a')$. With this property and $a_1 \simeq_H a_2$, we can see $O_1' =_S O_2'$.

Now, we will prove $c_1' = c_2'$, $m_1' = m_2'$ and $a_1' \simeq_H a_2'$. For this, we should consider $c =$ `recv` case separately. When $c =$ `recv`, so is c_L, from $c =_{sys} c_L$. Recall that we have shown $a \simeq_H pair(a)$. Using this property, we can examine rule (S1) and (L1) to conclude that $a_1' \simeq_H \tilde{a}_2$ holds. Besides, from rule (L2), $\tilde{a}_2 = a_2'$. From $a_1' \simeq_H \tilde{a}_2 = a_2'$, we can see $a_1' \simeq_H a_2'$. Also, $c_1' = c_2'$ and $m_1' = m_2'$ trivially hold from rule (S1) and (L2).

Next, we consider the case where c is not `recv`. Then, c_L is not `recv` as well, so the low execution can only update the `L` fields of a_2, and $a_2 =_H \tilde{a}_2$ holds. From $a_1 \simeq_H a_2 =_H \tilde{a}_2$, we can see $a_1 \simeq_H \tilde{a}_2$. Now we can prove $c_1' = c_2'$ and $m_1' = m_2'$. This time, the transition of $\langle c, m \rangle$ is decided by c, m, and the `H` fields of $\tilde{a}_2$. From $a_1 \simeq_H \tilde{a}_2$, we can see that c, m in the standard execution and c, m in the high execution always change in the same way. Therefore, $c_1' = c_2'$ and $m_1' = m_2'$ hold. Lastly, we prove $a_1' \simeq_H a_2'$. When c is `add` x `to` k, the standard execution updates a_1 with $m(x)$ while the high execution updates the `H` field of $\tilde{a}_2$ with m(x). Thus, $a_1' \simeq_H a_2'$ holds. If

c is not `add x to k`, $a_1 = a'_1$ and $\tilde{a}_2 = a'_2$, so $a'_1 \simeq_H a'_2$ directly holds from $a_1 \simeq_H \tilde{a}_2$.

Proof of Theorem 2. Assume input I, non-interferent program P, and the following three executions. First, we assume an execution with input I under the standard semantics $\langle P, m_0, a_0, I, O_0\rangle \rightarrow^n \langle c_1, m_1, a_1, I_1, O_1\rangle$, Second, we assume another execution under the standard semantics, using $rep(I)$ as input: $\langle P, m_0, a_0, rep(I), O_0\rangle \rightarrow^n \langle c_2, m_2, a_2, I_2, O_2\rangle$. Third, we assume a PAVEBOX execution with I: $\langle S_0, S_0, a_0, I, O_0\rangle \rightsquigarrow^n \langle\langle c_L, m_L\rangle, \langle c_H, m_H\rangle, a_3, I_3, O_3\rangle$.

To prove Theorem 2, we should prove that $\forall n \geq 0$, $O_1 = O_3$. We can prove this by showing that (i) $\forall n \geq 0$, $O_1 =_{NS} O_3$, and (ii) $\forall n \geq 0$, $O_1 =_S O_3$.

First, we prove (i) $\forall n \geq 0$, $O_1 =_{NS} O_3$. We first use the non-interference of P. Since P is non-interferent and $rep(I) =_{NS} I$, we know that $\forall n \geq 0$, $O_1 =_{NS} O_2$ holds from Definition 1. Next, we will prove that $O_2 =_{NS} O_3$. For this, we prove that $\forall n \geq 0$, $c_2 = c_L$, $m_2 = m_L$, $a_2 \simeq_L a_3$, $I_2 = rep(I_3)$, and $O_2 =_{NS} O_3$. When $n = 0$, this property trivially holds. Also, if the property holds for $n = k$, we can prove that it also holds for $n = k + 1$, using Lemma 3. Therefore, the property holds $\forall n \geq 0$, by induction on n. At this point, we have shown $O_1 =_{NS} O_2 =_{NS} O_3$.

Next, we prove (ii) $\forall n \geq 0$, $O_1 =_S O_3$. For this, we prove that $\forall n \geq 0$, $c_1 = c_H$, $m_1 = m_H$, $a_1 \simeq_H a_3$, $I_1 =_S I_3$, and $O_1 =_S O_3$. When $n = 0$, this property trivially holds. Next, we will use Lemma 4 to show that if the property holds for $n = k$, then it also holds for $n = k + 1$. However, we must first show that $\forall n \geq 0$, $c_1 =_{sys} c_L$ holds. From the non-interference of P, we know that $\forall n \geq 0$, $c_1 =_{sys} c_2$, as the length of I_1, I_2, O_1 and O_2 can change only by `recv` and `send`. Also, during the proof of $O_2 =_{NS} O_3$, we have proved $\forall n \geq 0$, $c_2 = c_L$. We now know that $\forall n \geq 0$, $c_1 =_{sys} c_L$ holds, so we can use Lemma 4 and prove the property with induction on n.

6 Evaluation

6.1 Experiment Setup

Prototype. We implemented a proof-of-concept of PAVE. Our toolchain supports a module written in C/C++ languages and libraries for libc and the SP interfaces. We use LLVM for the instrumentation for SFI and synchronization. Shadow execution uses approximately twice as many resources as standard execution. To reduce the redundancy, the PAVEBOX delays shadow execution: initially running a single execution, it starts a new thread for high execution once a message entry is read. Also, the PAVEBOX provides a snapshot mechanism [40] that stores its state after initializing a module and restores the snapshot right after finishing a request.

We implement two different versions of PAVE; PAVE Single and PAVE Sync. In PAVE Single, there is only a low execution without a high execution. It does not implement the shadow execution and the synchronization. Therefore, we can measure the overhead caused solely by Intel SGX and our SFI mechanism. PAVE Sync fully implements the PAVEBOX including the shadow execution with the synchronization. We also evaluate a version where the synchronization is omitted, which still preserves non-interference as proved in §4.3 and §5. However, it did not show a substantial performance advantage compared to PAVE Sync.

Benchmark. We use six applications for data-processing scenarios. In (1) file compression service (FileComp) such as ezyZip and (2) video cropping service (VidEdit) such as VEED, after a server module receives a file from the user, it returns the processed file. We used gzip [65] and FFmpeg [83]. For (3) OpenSSL file encryption service (FileEnc), a user first sends a key file. The server checks its validity and returns the validity. Then, the user sends a file and receives its encrypted file. For (4) GIF frame extractor service (GIFExt) and (5) audio converter service (AudConv), the server processes the user media file and sends the intermediate result (e.g., the number of GIF frames or the new file size). Then, the user responds back about which frame number is chosen or whether to download the new file. The service returns the corresponding result. We used GIFLIB [35] and FLAC [33] libraries. The last scenario is (6) privacy-preserving ad service (RTB) [23, 39, 84]. We developed a user that sends a set of user interests (e.g., sports, cars, etc.). Based on the user interests, ad servers collaborate to deliver the best matching ad. For this, we build three ad server modules performing real-time bidding (RTB), one of the well-known ad mechanisms for programmatic ad auctions. We developed these ad server modules by extending the RTB algorithms in [99] with the IPinYou dataset [51]. When the first module receives the set of user interests, it is propagated to the others. Each of the other two modules submits a bid back to the first module, which performs the matching or arithmetic operations to choose the ad (in a URL form) with the highest bid. The first module finally delivers the chosen ad to the user.

The scenarios above illustrate the problem that PAVE aims to solve: the trade-off between utilizing online data-processing services and sacrificing user privacy. The FileComp, VidEdit, FileEnc, GIFExt, and AudConv modules each require memory-intensive operations compared to the RTB modules. Additionally, the FileEnc, GIFExt, AudConv, and RTB modules require multiple round-trip communications in a distributed network, which was not possible in previous work.

Environments. We measure (i) the computation overhead to initialize a PAVEBOX, (ii) the execution time of a module, and (iii) the user latency. For comparison purposes, baseline programs are general Linux programs without an enclave or a PAVEBOX. Every plot is averaged over ten runs. All the modules run on a Linux desktop computer equipped with an Intel Core i9-10900K CPU with 10 cores and 20 hyperthreads. The lengths of input messages for FileComp, VidEdit, FileEnc, GIFExt, AudConv, and RTB are about 50KB, 600KB, 5K, 100K, 100K, and 8KB, respectively. Since an enclave has no time source, we use the host time source to measure the

Table 2. The execution time (ET) and the page fault number (PF) of each module for our scenarios. The unit of ET is ms.

	FileComp		VidEdit		FileEnc		GIFExt	
	ET	PF	ET	PF	ET	PF	ET	PF
Baseline	65	383	239	2,906	47	312	78	1,093
Pave Single	187	36,108	286	47,939	75	37,002	90	35,364
Pave Sync	272	45,120	509	79,422	157	45,265	189	46,389

	AudConv		RTB					
			Module 1		Module 2		Module 3	
	ET	PF	ET	PF	ET	PF	ET	PF
Baseline	69	778	235	415	44	289	44	283
Pave Single	123	35,602	298	34,817	48	34,891	45	34,857
Pave Sync	191	44,076	409	43,143	106	43,292	102	43,270

performance. In contrast to the baseline, the measurement of a PaveBox includes an additional overhead.

6.2 PaveBox Initialization Overhead

To evaluate the computation overhead to initialize and load a PaveBox, we measure the time to run a simple program in C '`int main{return 0;}`.' Specifically, we measure (i) the time to create and load an enclave memory, (ii) the time to initialize the PaveBox after entering the enclave, and (iii) the time to run the test program and terminate the enclave. It takes 1,431 ms to create and load an enclave memory. Initializing the PaveBox including the loading process of the encrypted module takes 1,839 ms. This includes SFI validation time in §4.1 (30 ms), synchronization validation time (330 ms), and remote attestation overhead (1,454 ms). The time for running the test program is 2 ms. The total time from creating the enclave to exiting from the enclave is 3,272 ms. Note that, by using the snapshot technique, the initialization overhead ((i) and (ii)) incurs only once when the host starts the module. The snapshot overhead is 3.7 ms, which is more efficient than the initialization.

6.3 Execution Time

Table 2 shows the execution time to run a module. The execution time is measured from the moment the module accepts a user request until the termination of the module. Compared to the baseline, the execution time for Pave Single increased by 187.6% for FileComp, 19.6% for VidEdit, 57.9% for FileEnc, 15.1% for GIFExt, and 79.2% for AudConv. For these scenarios, the shadow execution and synchronization (i.e., in Pave Sync) incur 45.5% for FileComp, 77.6% for VidEdit, 109.5% for FileEnc, 110.8% for GIFExt, and 55.0% for AudConv. Pave Single increases the execution time mainly due to the SGX overhead to access the enclave memory and the code overhead of the SFI-compliant binary. The overhead of Pave Sync primarily stems from the shadow execution and synchronization. Since both executions share the enclave EPC memory area, this increases memory access overhead. Additionally, the number of executed basic blocks contributes to the amount of instrumented code to be executed. For the three RTB modules, the execution time increases as the Pave implementation becomes a full stack; 12.9% (i.e., Pave Single compared to the baseline) and 94.1% (i.e., Pave Sync compared to Pave Single).

We plot the number of page faults to show that the paging overhead due to the limited EPC size (128 MB) for the entire execution may influence the overhead of Pave. It also implies that VidEdit, FileEnc, GIFExt, AudConv, and FileComp perform memory-intensive operations and hence incur more overhead, compared to RTB scenarios. Even though three RTB modules share the limited ECP, the numbers are similar.

We further analyze the impact of the EPC size on memory access overhead. We allocate data of 32 MB, 48 MB, 64 MB, 80 MB, 96 MB, 112 MB, 128 MB, and sequentially access each page from start to end by selecting a random address within each page. The total time to access all pages is measured. Using the 32 MB access time as a baseline, the observed overheads are 53%, 106%, 154%, 420%, 1236%, and 2218%, respectively. These results indicate a non-linear increase in overhead when data size reaches 96 MB, likely due to the EPC limit of 128 MB EPC, with only 93 MB available to user applications. As processors with larger EPC sizes (512 MB–1 TB [42]) become more common, it can be further mitigated.

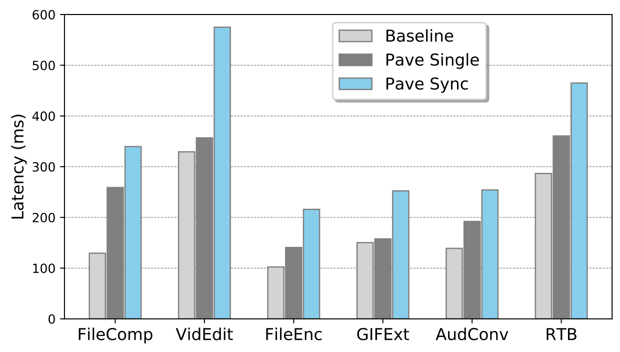


Figure 7. The user latency to receive a processed result.

6.4 User Latency

The user latency refers to the time between the moment of sending user data and that of receiving its result as shown in Figure 7. The user latency also increases from the baseline to Pave Sync. Compared to the baseline, the user latencies are increased by 100.1% for FileComp, 8.4% for VidEdit, 37.4% for FileEnc, 4.8% for GIFExt, 38.1% for AudConv, and 25.8% for RTB. Compared to Pave Single, the shadow execution and synchronization (i.e., in Pave Sync) incurs 28~61% more latencies; FileComp (31.2%), VidEdit (61.2%), FileEnc (53.4%), GIFExt (60.2%), AudConv (32.3%), and RTB (28.9%). Similar to the results of the execution time, the memory-intensive tasks affect the latency. These overheads show that the user may experience delays depending on online service scenarios. Similarly, it can be reduced with a larger EPC memory.

Table 3. Features of Pave that mitigate a side-channel attack.

	Attack Vector	MP	DH	OI	IL	CA	SFI	ETM[1]
[98]	Page table				✔	✔		✔
[90]	Page table		✔		✔			
[55]	L1 cache			✔	✔			✔
[15, 36]	L1 cache		✔	✔	✔			
[41, 50]	Branch Prediction Unit			✔	✔			✔
[31]	Branch Prediction Unit		✔	✔	✔			✔
[63, 95]	Cache, DRAM			✔	✔			
[76]	Cache, DRAM			✔	✔			✔
[49]	Memory safety			✔		✔	✔	
[52, 86][2]	Speculative Execution	✔	✔					
[20, 46, 48, 75][2]	Speculative Execution	✔						
[74, 87, 91, 92][2]	Microarchitectural buffers	✔						
[68]	Microarchitectural buffers	✔		✔	✔			
[56]	# of instructions				✔	✔		✔
[89]	Interrupt latency			✔	✔			✔
[4]	Execution port		✔	✔	✔			
[43, 57]	Voltage interface	✔			✔	✔		
[5]	Floating point operations				✔	✔	✔	
[66]	CPU Frontend			✔	✔	✔	✔	✔
[13][2]	APIC	✔						

[1] It is based on the cases when there are direct performance references in the paper or when the victim state is measured after every instruction.

[2] These studies demonstrate how to steal seal keys or attestation keys from the Intel SGX architecture. Thus, we do not evaluate the impacts of OI, IL, CA, SFI, and ETM.

7 Discussions

Side-Channel Attacks (SCAs). Table 3 shows whether Pave can mitigate SCAs by the cloud provider using the following features:

- Microcode patch (MP): A user can verify that the microcode version is kept up-to-date.
- Disabling Hyperthreading (DH): It prevents an attacker from compromising resources on the same core.
- One-time input (OI): Some attacks need to run the victim program multiple times with the same input. In Pave, the user request is processed only once and the cloud provider cannot replay a user request due to the TLS-based session.
- In-enclave loading (IL): Some attacks require analyzing a victim program (i.e., binary or source code) to exploit its control flow depending on secret data or to reveal in-flight data. However, Pave encrypts the module binary.
- Code Auditing (CA): Pave is intended to be open-sourced allowing the user to verify its security or implementation. This enables the community to manually harden the code.
- Software Fault Isolation (SFI)
- Execution Time Monitoring (ETM): Some attacks significantly increase the runtime overhead, which can be detected by the user or the service provider.

We believe the benefits of the SCAs are marginal. First, it is not in the interest of cloud providers to allow insider attacks, as such actions would undermine their reputation and credibility. Moreover, predicting which module will process a specific user request is challenging, making targeted attacks on individual users impractical. Finally, ongoing development of countermeasures diminishes the impact [1, 21, 22, 24, 38, 61, 62, 80–82].

Spectre Attacks. In §4, we demonstrate each execution is isolated by our SFI mechanism. However, it can be bypassed by Spectre attacks. Spectre attacks exploit speculative execution to manipulate the microarchitectural state by leveraging mispredictions in various prediction units (e.g., Pattern History Table; PHT). These attacks are typically categorized into four types based on their root cause [16]: Spectre-PHT, Spectre-BTB, Spectre-RSB, and Spectre-STL. Except for Spectre-STL, the other types have been shown to be feasible even when both an attacker (i.e., low execution) and a victim (i.e., high execution) are sandboxed within the same address space [16]. The attacker can influence the victim's control flow by manipulating the PHT (Spectre-PHT [46]), Branch Target Buffer (Spectre-BTB [46]), or Return Stack Buffer (Spectre-RSB [53]), redirecting execution to an intended destination.

Although Pave cannot entirely thwart Spectre attacks, our SFI rules mitigate their effects. First, because the (SFI-enforced) attacker cannot jump into the middle of a code bundle (Rules `N1` and `N2`), it also cannot train the BTB to steer the victim's execution to an arbitrary location within the bundle. Consequently, with the absence of the `ret` instruction (Rule `N3`), return-oriented programming (ROP)-based control hijacking, which is exploited in Spectre-BTB and Spectre-RSB attacks, can be mitigated. Second, as no instruction is allowed to access data outside its designated region (Rules `E1`–`E5`), leak gadgets that extract out-of-bounds data cannot exist in the module at the first place. Therefore, Spectre-PHT, which relies on such gadgets, is mitigated.

Developer Efforts. Recall that Pave preserves the functionality of a program only when it is already non-interferent without Pave. Thus, the service provider is responsible for developing software modules that satisfy the non-interference. Such burden of developers can be alleviated in several ways. First, a module can adopt well-studied algorithms that prevent data leakages through a timing channel (e.g., [77]). Next, the developer can use some tools that analyze whether a module is non-interferent [60], or other tools that automatically re-write a given module [69, 97] to satisfy non-interference. Lastly, the software interfaces of Pave can be narrowed by integrating the SGX enclave with system features such as LibOS [85] and oblivious file system [3]. Such integration reduces the number of syscalls that the developer should consider (e.g., `ioctl`).

Formal Model vs. Implementation. The key difference between the formal model (§5) and implementation (§4) is the notion of time unit. In the formal model, the minimal time unit was one local transition step defined in Figure 5. For Definition 1 to be meaningful, all the local transition steps must take the same amount of time, or their difference must be negligible. Meanwhile, in the implementation, our minimal time unit is execution of a basic block. Therefore, the non-interference of Pave relies on the assumption that the execution time of basic blocks are indistinguishable from one another. To strengthen this assumption, Pave toolchain may split a large or possibly time-consuming basic block into smaller blocks, which we leave as a future work.

Transparency under OS Scheduler. As discussed in §4.3, PaveBox relies on synchronization instructions to make the low and high executions keep pace with each other. If PaveBox fails to validate proper instrumentation of basic blocks during the initialization step (§4.4), it falls back on the OS scheduler. In this case, the transparency of the system cannot be guaranteed and the functionality of the program may not be preserved.

8 Related Work

Thoth [30] and Flowfence [32] provides user-defined policies to prevent data leaks by buggy server programs. Thoth uses a kernel monitor for I/O tracking, while FlowFence employs an Opacified Computation Model to restrict data movement from IoT devices. VC3 [73], inline monitoring [10], Riverbed [93], Mitigator [54], and HasTee [70] use trusted hardware to control user data. VC3 proposes a MapReduce framework to enhance the code and data security by using Intel SGX. Inline monitoring uses Intel SGX to check whether a binary complies with the policy described using Avenance [11]. In Mitigator, a verifier checks the compliance of the source code with a privacy policy using static analysis, and produces a signature. Riverbed uses TEE with taint tracking. HasTee [70] proposes a TEE-abstracted type system that partitions a program to isolate sensitive code and data while enforcing security policies. Ryoan [40] is the first work to address implicit IFC in the context of an untrusted server. While it supports request-oriented applications more efficiently, our system accepts wider range of applications.

SME [17, 27] was generalized as a scheduling approach and its security properties were analyzed from the perspective of timing- and termination-sensitivity. Also, declassification schemes were proposed [12, 67]. Rafnsson and Sabelfeld [67] use the concept of an observable channel to declassification. Multiple Facets (MF) [6, 8, 58, 72] simulate the result of the SME without multiple executions. While our presentation in §4.2 resembles that of MF, our shadow proxy uses it for a different purpose, which is allowing the two executions to share the message in a securely controlled fashion. Finally, SME has been adapted for various applications including Web APIs [25, 26], Unix-like systems [64], and Android [18]

9 Conclusion

This paper presents Pave, an IFC framework to accomplish privacy-preserving online data-processing services. Even if a module is developed by a potentially malicious service provider, Pave can assure a user that the user's data would be protected. The shadow execution, the shadow proxy, and the synchronization are the key design elements to achieve the timing-sensitive non-interference and the functionality preservation at the same time. Our implementation enforces the IFC rules by multi-domain address masking. We carry out prototype-based experiments to demonstrate the feasibility of Pave.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Owen Arden, for their helpful comments. This research was supported by the NRF and IITP (ITRC program and Next-generation Cloud-native Cellular Network Leadership Program) funded by the Korea government (MSIT). The projects are RS-2023-00220985, RS-2023-00210989, NRF-2022R1A2C2011221, IITP-2024-2021-0-02048, and IITP-2025-RS-2024-00418784.

References

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*, 2019.

[2] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: efficient multi-client isolation under adversarial programs. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[5] Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. Faulty point unit: Abi poisoning attacks on intel sgx. In *Annual Computer Security Applications Conference*, pages 415–427, 2020.

[6] Maximilian Algehed, Alejandro Russo, and Cormac Flanagan. Optimising faceted secure multi-execution. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 1–115. IEEE, 2019.

[7] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*, pages 333–348. Springer, 2008.

[8] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 165–178, 2012.

[9] Guest Authors. The threats you're facing when using free online converters and how to avoid them, November 2020. [Online]. Available: https://www.techwalls.com/threats-free-online-converters-how-to-avoid/ (Retrieved, Apr 16, 2024).

[10] Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B Schneider. Sgx enforcement of use-based privacy. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, pages 155–167, 2018.

[11] Eleanor Jane Birrell. *A Reactive Approach for Use-Based Privacy.* PhD thesis, Cornell University, 2018.

[12] Iulia Boloşteanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *International Conference on Principles of Security and Trust*, pages 24–45. Springer, 2016.

[13] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. {ÆPIC} leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, 2022.

[14] Nicholas Boucher. Multi-domain sfi, May 2019. [ONLINE]. Available: https://github.com/nickboucher/Multi-Domain-SFI/blob/master/paper/Multi%20Domain%20SFI.pdf (Retrieved, Apr 15, 2024).

[15] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *WOOT*, pages 11–11, 2017.

[16] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[17] Roberto Capizzi, Antonio Longo, VN Venkatakrishnan, and A Prasad Sistla. Preventing information leaks through shadow executions. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 322–331. IEEE, 2008.

[18] Dhiman Chakraborty, Christian Hammer, and Sven Bugiel. Secure multi-execution in android. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1934–1943, 2019.

[19] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[20] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[21] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 178–194. IEEE, 2018.

[22] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[23] Chromium. Google privacy sandbox. [Online]. Available: https://www.chromium.org/Home/chromium-privacy/privacy-sandbox (Retrieved, Apr 15, 2024).

[24] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[25] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 748–759, 2012.

[26] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4):469–509, 2014.

[27] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[28] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[29] Paul Ducklin. Online file conversion services – why trust them?, September 2017. [Online]. Available: https://news.sophos.com/en-us/2017/09/01/online-file-conversion-services-why-trust-them/ (Retrieved, Apr 15, 2024).

[30] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 637–654, 2016.

[31] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[32] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 531–548, 2016.

[33] Flac. [Online]. Available: https://xiph.org/flac/ (Retrieved, Apr 16, 2024).

[34] Jeremiah Fowler. Fotor photo editing app leaked 13 million users' info online, Dec 2020. [Online]. Available: https://securitydiscovery.com/fotor-photo-editing-app-leaked-13-million-users-info-online/ (Retrieved, Apr 13, 2024).

[35] Giflib. [Online]. Available: https://giflib.sourceforge.net/ (Retrieved, Apr 16, 2024).

[36] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[37] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[38] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, 2017.

[39] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical privacy in online advertising. In *USENIX conference on Networked systems design and implementation*, pages 169–182, 2011.

[40] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):13, 2018.

[41] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 321–347, 2020.

[42] Intel. Intel processors supporting intel sgx. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html (Retrieved, November 6, 2024).

[43] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1445–1461, 2020.

[44] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. Reassembly is hard: a reflection on challenges and strategies. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1469–1486, 2023.

[45] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863*, 2018.

[46] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[47] KIM KOMANDO. 1.9 million records leaked after online photo editor hacked, January 2021. [Online]. Available: https://www.komando.com/security-privacy/pixlr-data-leak/774844/ (Retrieved, Apr 14, 2024).

[48] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael B Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT@ USENIX Security Symposium*,

2018.
[49] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, August 2017. USENIX Association.
[50] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017.
[51] Hairen Liao, Lingxiao Peng, Zhenchuan Liu, and Xuehua Shen. ipinyou global rtb bidding algorithm competition dataset. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–6. ACM, 2014.
[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.
[53] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.
[54] Miti Mazmudar and Ian Goldberg. Mitigator: Privacy policy compliance using trusted hardware. *Proceedings on Privacy Enhancing Technologies*, 1:18, 2020.
[55] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.
[56] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 469–486, 2020.
[57] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.
[58] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. A better facet of dynamic information flow control. In *Companion Proceedings of the The Web Conference 2018*, pages 731–739, 2018.
[59] Huan Nguyen, Soumyakant Priyadarshan, and R Sekar. Scalable, sound, and accurate jump table analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 541–552, 2024.
[60] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187. IEEE, 2019.
[61] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 227–240, 2018.
[62] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.
[63] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, 2016.
[64] Tobias Pfeffer, Thomas Göthel, and Sabine Glesner. Efficient and precise information flow control for machine code through demand-driven secure multi-execution. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 197–208, 2019.
[65] GNU Project. Gzip. [Online]. Available: http://www.gnu.org/software/gzip/ (Retrieved, Apr 15, 2024).
[66] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in sgx via the cpu frontend. In *USENIX Security Symposium*, pages 663–680, 2021.
[67] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.
[68] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.
[69] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, 2015.
[70] Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. Hastee: Programming trusted execution environments with haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium*, pages 72–88, 2023.
[71] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for intel sgx with intel data center attestation primitives. *White paper*, 12, 2018.
[72] Thomas Schmitz, Maximilian Algehed, Cormac Flanagan, and Alejandro Russo. Faceted secure multi execution. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1617–1634, 2018.
[73] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE symposium on security and privacy*, pages 38–54. IEEE, 2015.
[74] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.
[75] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.
[76] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.
[77] Martin Schwarzl, Pietro Borrello, Gururaj Saileshwar, Hanna Müller, Michael Schwarz, and Daniel Gruss. Practical timing side channel attacks on memory compression. *arXiv preprint arXiv:2111.08404*, 2021.
[78] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010.
[79] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 955–970, 2020.
[80] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.
[81] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *arXiv preprint arXiv:1506.04832*, 2015.
[82] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks.

arXiv preprint arXiv:1712.08519, 2017.

[83] FFmpeg team. Ffmpeg. [Online]. Available: https://ffmpeg.org/ (Retrieved, Apr 15, 2024).

[84] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Proceedings Network and Distributed System Symposium*, 2010.

[85] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[86] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[87] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[88] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1741–1758, 2019.

[89] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195, 2018.

[90] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, pages 1041–1056. USENIX Association, 2017.

[91] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[92] Stephan Van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354. IEEE, 2021.

[93] Frank Wang, Ronny Ko, and James Mickens. Riverbed: enforcing user-defined privacy constraints in distributed web services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 615–630, 2019.

[94] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[95] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[96] Zack Whittaker. A server hosting dozens of popular file converter sites has been hacked, September 2017. [Online]. Available: https://www.zdnet.com/article/dozens-of-online-file-converter-sites-are-unsafe-to-use-warns-researcher/ (Retrieved, Apr 14, 2024).

[97] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[98] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[99] Weinan Zhang, Shuai Yuan, and Jun Wang. Optimal real-time bidding for display advertising. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1077–1086. ACM, 2014.