WireTap: Breaking Server SGX via DRAM Bus Interposition

Alex Seto
Purdue University
West Lafayette, IN, USA

Oytun Kuday Duran
Georgia Tech
Atlanta, GA, USA

Samy Amer
Georgia Tech
Atlanta, GA, USA

Jalen Chuang
Georgia Tech
Atlanta, GA, USA

Stephan van Schaik
van Schaik, LLC
Ann Arbor, MI, USA

Daniel Genkin
Georgia Tech
Atlanta, GA, USA

Christina Garman
Purdue University
West Lafayette, IN, USA

Abstract

Intel's Software Guard eXtension (SGX) aims to offer strong integrity and confidentiality properties, even in the presence of root-level attackers. However, while Intel clearly indicates that SGX offers no security against attackers with physical access, many current real world SGX deployments are actually done in potentially adversarial environments, where node operators have a financial incentive to subvert computations performed inside SGX enclaves. While the two threat models clearly differ, a common conception is that physical attacks on SGX require expensive laboratory equipment, thus putting them out of reach of hobbyist-level attackers.

In this work we challenge this belief, showing how simple memory bus interposition hardware can be constructed cheaply and easily in basic environments, using equipment easily purchased on the internet. We then combine our setup with SGX's recent migration from client CPUs to servers, which resulted in a weaker (and deterministic) memory encryption being used to encrypt the machine's physical memory. Applying our acquisition setup to SGX's attestation enclaves, we are able to extract an SGX attestation key from a machine in fully trusted status.

Finally, we study the real world implication of such SGX breaches, by examining how SGX-backed blockchain deployments perform in the presence of these adversaries. As many of these deployments allow any SGX machine in trusted status to perform critical network functionality, we show end-to-end attacks on both confidentiality and integrity guarantees of deployments with multi-million dollar market caps, allowing attackers to disclose confidential transactions or illegitimately obtain transaction rewards.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**; **Hardware-based security protocols**; **Hardware reverse engineering**; • **Hardware → Buses and high-speed links**.

Keywords

Side Channels, TEE, Hardware Security, Memory, DRAM, Blockchain

ACM Reference Format:
Alex Seto, Oytun Kuday Duran, Samy Amer, Jalen Chuang, Stephan van Schaik, Daniel Genkin, and Christina Garman. 2025. WireTap: Breaking Server SGX via DRAM Bus Interposition. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3719027.3765204

1 Introduction

Trusted Execution Environments (TEEs) promise to revolutionize computing, giving confidentiality and integrity guarantees even in the presence of root-level adversaries with nearly unlimited software capabilities. Using TEEs, clients can now offload their data and computation to untrusted cloud providers, while obtaining strong guarantees about the correctness of their offloaded computation and the confidentiality of their secret data. While many examples such as AMD's SEV or ARM's TrustZone exist, perhaps the most mature TEE example is Intel's Software Guard eXtensions (SGX), which has recently been migrated from low-end client CPUs to full fledged data-center grade Xeon Scalable processors.

Recognizing the potential of strong integrity and confidentiality guarantees at near native-level performance, SGX has thus far received adoption primarily in the blockchain community, where (confidential) transactions are executed by enclaves, allowing their contents to be protected by SGX's security guarantees. Starting from humble origins of initial academic prototypes [7, 33, 75], SGX has now seen usage in many real-world Web3 deployments, from simple confidential transactions [51], to confidential smart contracts [46], to even complicated schemes of arbitrary delegation of computation or storage [10, 13]. Together, these deployments service numerous users and have a combined market cap of hundreds of millions of dollars, ushering in a new age of hardware-backed privacy guarantees.

However, when comparing SGX's threat model to its actual deployments, an unsettling gap arises. More specifically, while Intel claims that SGX does not offer complete mitigation in the presence of hardware attackers with physical access [22, 23], Web3-related SGX deployments are often permissionless, meaning any node can join the network regardless of its physical location, by merely presenting a trusted attestation status. Indeed, real-world deployments often acknowledge SGX's checkered security record [58, 62, 76], often arguing that SGX attacks are only possible in carefully-controlled and unrealistic lab settings while requiring expensive hardware attack equipment [52]. Thus, given the clear gap in SGX threat modeling, in this paper we ask the following questions:

What are the true costs of mounting physical attacks on SGX hardware? In particular, can such attacks be mounted by low-budget hobbyists? And if so, how can one best defend against them?

Our Contributions

In this paper we positively answer the above questions, demonstrating how memory interposition attacks on server-grade SGX can be done cheaply and simply, by hobbyists with under $1000 budget. We empirically show how our interposition setup is able to extract secret attestation keys from machines in fully trusted status, allowing us to run enclaves outside of SGX's protections. Finally, with SGX's confidentiality and integrity guarantees successfully breached, we proceed with evaluating the implications of SGX attestation key extraction attacks on real-world SGX deployments totaling a market cap of over $135M USD. Examining three main case studies, we show how SGX breaches result in loss of both transaction privacy and integrity guarantees, in some cases even allowing attackers to undetectably reap rewards and profit without doing any actual legitimate work.

Constructing a Cheap Memory Interposition Setup. While prior work cites costs of memory interposition setups to be around $170,000 [34], we begin by showing that such numbers are vastly overestimated, allowing hobbyist-level attackers to perform DIMM interposition attacks with under $1000. We achieve this by first showing how to significantly slow down DDR4 bus transactions by modifying the DIMM's internal metadata, thus significantly simplifying our acquisition setup. We then show how DIMM interposition probes can be constructed from readily-available supplies from secondhand electronic marketplaces, using just tweezers and simple soldering irons. Finally, we show how both can be used to observe DDR4 DRAM traffic, using outdated (and thus cheap) logic analyzers not originally intended for DDR4 analysis (or even capable of operating near DDR4 clock speeds). See Figure 1.

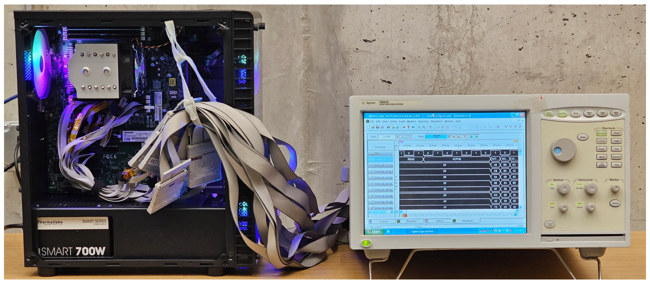


Figure 1: Our memory interposition setup, with target machine (left) and logic analyzer (right). Notice the gray wires from the DIMM interposer (middle).

Obtaining Enclave Control. With our interposition setup in place, we proceed to gain a sufficient level of control over SGX enclaves in production status. First, as our interposition setup is only capable of observing traffic to a single DDR4 DIMM out of a minimum of eight required for SGX activation, we must reverse engineer the system's mapping between physical address and DIMM locations. With the system's mappings in hand, our next step is to modify the SGX driver to place the enclave place of interest on the DIMM connected to our logic analyzer. Finally, to make the traffic observable on the DRAM bus we overcome the system's caching by flushing, using an enclave control channel to precisely synchronize cache flushing events.

Attacking SGX Attestation. Having achieved the level of enclave control required for observing enclave memory traffic, we proceed to attacking SGX's cryptographic security mechanisms. To that aim, we first verify Intel's use of deterministic XTS encryption, characterizing its dependence on the data's physical address. We then continue to attacking SGX's DCAP attestation mechanism inside Intel's quoting enclave, using a machine in a fully trusted attestation status. Here, we apply the techniques from the previous step to observe encrypted memory traffic during ECDSA signing operations, essentially obtaining a ciphertext equality oracle during ECDSA's scalar by point operation. With this method, we are able to extract the machine's attestation key within 45 minutes, with the main limitation being the limited IO bandwidth of our Pentium III based logic analyzer. Finally, to the best of our knowledge, this is the first demonstration of SGX DCAP attestation key extraction from Xeon Scalable servers in fully trusted status.

Breaking Enclave Confidentiality. Having recovered the machine's attestation key, we now proceed to investigate how real world SGX deployments fail in the presence of key compromise. We noticed that when faced with such an attack, numerous deployments have complete breakdowns of their confidentiality guarantees, resulting in passive data decryption. As such, we begin by analyzing PHALA and SECRET, two privacy-preserving smart contract networks with $79M USD and $55M USD market caps respectively. These networks utilize SGX enclaves to ensure the confidentiality of smart contract data, as well as for key derivation, distribution, and storage. We demonstrate how an attacker can extract various keys used to encrypt contract data from the enclaves by forging quotes using a custom quoting enclave, allowing us to breach confidentiality and carry out network wide decryption of smart contract state.

Breaking Enclave Integrity. We then proceed to study CRUST, a decentralized blockchain storage system that utilizes SGX enclaves to guarantee proof of storage of files. Unlike PHALA and SECRET, CRUST relies on SGX and its attestation mechanism primarily to ensure integrity of computation. The attestation process should guarantee the integrity and correctness of actions performed by a network node, that is, that the node is running a trusted, unmodified binary inside an enclave. We demonstrate how, using our extracted key and custom quoting enclave, an attacker can run a modified enclave and fake proofs of storage, thus allowing them to claim rewards without ever actually storing files.

Summary of Contributions. We contribute the following:

- We show how DDR4 bus interposition can be performed cheaply using equipment available on electronic marketplaces (Section 4).
- We demonstrate how to achieve control over SGX enclaves, forcing enclave data to be visible to our interposition setup (Section 5).
- We breach SGX's security guarantees by extracting an attestation key from a Xeon server in a fully trusted status (Section 6).
- We show how a compromised attestation key allows breaking confidentiality of execution inside SGX enclaves by extracting secret keys for PHALA and SECRET confidential smart contracts (Section 7).
- We forge proofs of storage on the CRUST network, thereby showing how enclave breaches affect the integrity guarantees of applications that use SGX for enforcing execution correctness (Section 8).

Disclosure. Following the practice of coordinated vulnerability disclosure, we have shared our findings with Intel, as well as the security teams of the affected SGX deployments. Intel and all affected deployments have acknowledged our findings and will make statements simultaneously with the public release of this paper.

Ethics. Keeping with ethical research practices, we extracted the attestation key only from our own local machine. Additionally, all blockchain case studies and attack PoCs were performed either on our own isolated local network setups or project testnets designated for such testing purposes. This allowed us to carefully ensure that no real user data, keys, or other sensitive material was ever accessed or exposed, beyond our own created to verify our proof of concepts.

2 Background and Related Work

2.1 Intel Software Guard Extensions

Intel's Software Guard eXtensions (SGX) extend the x86_64 instruction set with instructions to create trusted execution environments, called enclaves, that support secure code execution. SGX aims to prevent the inspection and modification of both code and data within enclaves, even in the presence of root-level adversaries, with full control over the systems software stack. In addition, Intel SGX provides an attestation process that allows remote parties to ensure that enclaves are running on genuine and trustworthy Intel hardware.

Enclave Identification. To identify enclaves, SGX maintains a measurement of the enclave's initial state (`mrenclave`) and the enclave developer's identity (`mrsigner`) for each enclave. The `mrsigner` value is a cryptographic hash of the public RSA key that the enclave developer used to sign the enclave. The `mrenclave` value represents the enclave's initial state and is a measurement of the enclave's contents (code and data) chosen by the developer.

Trusted Compute Base (TCB). The SGX TCB consists of the trusted components that must operate correctly, and thus may not be malicious or compromised, for SGX to operate securely. More specifically, among these components are the CPU itself and its microcode, as well as the CPU's Provisioning Certificate Key (PCK). For software, SGX relies on the correctness of Intel-provided Provisioning and Quoting enclaves, which handle the machine's attestation key. Notably, other system components, such as the machine's DRAM bus, BIOS, or even the DIMM modules themselves, remain untrusted, and thus are not part of SGX's TCB.

Enclave Page Cache. A subsection of the physical address space, named the Enclave Page Cache (EPC), is reserved for SGX enclave memory. This section can only be occupied by enclave related pages and is managed by the SGX driver in the Linux kernel.

Memory Encryption in Client SGX. Intel SGX client platforms implement a Memory Encryption Engine (MEE) to guarantee the confidentiality, integrity and freshness of SGX memory [14]. While confidentiality is guaranteed through the use of AES encryption, the MEE in Intel SGX client platforms employs Merkle trees, stored on-die, to guarantee integrity. In addition, it uses a "tweaked" AES counter mode to guarantee freshness, storing a counter for each cache line and incrementing it on every write operation. Being part of every Intel platform from Skylake to CometLake (6th to 10th generation), the now-deprecated client SGX implementation avoided the use of deterministic encryption at the cost of limiting SGX memory to 512 MB.

Memory Encryption in Scalable SGX. With SGX's migration from client to scalable server platforms, Intel has increased the size of available SGX memory up to 512GB per processor on high-end CPU models. This was done by deploying a new memory encryption engine called Intel Total Memory Encryption (TME). More specifically, TME is implemented in the DRAM controller and encrypts the entire address space using AES-XTS with a key determined at boot time [22], writing ciphertexts to memory instead of plaintext data at a granularity of 128-bit blocks. Furthermore, AES-XTS for TME includes a tweak function that incorporates the physical address, guaranteeing that data written to different physical addresses produce different ciphertexts. For the rest of this paper, we focus on Scalable SGX, which is present in all Intel Xeon Scalable processors starting from the 3rd generation (IceLake).

2.2 Remote Attestation

One of the compelling features of SGX is that an enclave can attest to a remote verifying party that it is indeed running on genuine and trustworthy Intel hardware, guaranteeing the confidentiality and integrity of data inside it. This for example ensures that the remote party can subsequently provision that enclave with secrets, with the assurance that these secrets never leave. In this paper, we specifically focus on the remote attestation process used by Intel's Xeon Scalable servers, called SGX Data Center Attestation Primitive (SGX DCAP). We note that an earlier primitive (Enhanced Privacy ID — EPID) originally meant for client platforms has been recently deprecated by Intel [25]. We now proceed to overview the DCAP attestation mechanism.

Local Attestation. When an enclave wants to prove (or attest) to a remote verifier, it first needs to prove its own identity to the Quoting Enclave (QE) through a process called *local attestation*. First, the proving enclave issues the `ereport` instruction to generate a report containing the `mrenclave` and `mrsigner` values of the proving enclave. This report is then signed using a key that is only accessible to the QE. Finally, the proving enclave passes the report to the QE, which proceeds with the remote attestation process.

Remote Attestation. Upon completing local attestation, the QE uses the data from the report to produce and sign a "Quote", which is used to authenticate the proving enclave to remote parties. More specifically, Intel's QE issues the `egetkey` instruction to request the sealing key from which it derives a repeatable signing key that is unknown to Intel. While any QE can use its own preferred methods and algorithms to generate their own attestation key with which to sign the quote, the Intel-provided QE uses IETF RFC 6090 compliant 256-bit ECDSA signatures over the NIST p-256 curve [39, 60].

Provisioning Certificates. To ensure that an arbitrary key pair cannot be used to sign and verify quotes, the QE provides the public attestation key to Intel's PCE. The PCE derives a device-specific PCK, which is also a 256-bit ECDSA key used to sign the certificate identifying the QE and its attestation key. Intel additionally publishes certificates and certificate revocation lists for the PCKs in all genuine Intel platforms, ensuring a complete signature chain from the DCAP quotes to the Intel Certificate Authority. Using these, anyone can verify that the resulting quote originates from a genuine Intel platform.

2.3 Attacks on TEEs

SGX. While attacks such as microarchitectural data sampling (MDS) [5, 61, 66] demonstrate how an attacker can exfiltrate data from enclaves, Foreshadow, CacheOut and AEPICLeak [3, 65, 67] show various ways of how attackers can extract SGX attestation keys. Crosstalk [59] demonstrates how to extract the ECDSA nonces from an SGX enclave using IPP crypto's ECDSA implementation, and how to recover the ECDSA key from these nonces. In general, SGX.Fail [68] provides an overview of SGX attacks and demonstrates how these can be used to attack applications. However, all of these works focus on (now deprecated) Intel SGX platforms implementing EPID, rather than Xeon Scalable platforms implementing DCAP.

AMD SEV. While using a similar 128-bit AES-XEX memory encryption, unlike SGX, AMD SEV does not prevent the hypervisor from issuing memory accesses to read enclave ciphertexts. Exploiting this, Cipherleaks [36] breaks AMD SEV-SNP by observing ciphertexts corresponding to the encrypted VM Save Area (VMSA), allowing them to breach the constant-time ECDSA and RSA implementations of OpenSSL. Li et al. [35] extend Cipherleaks' result that ciphertext side-channel attacks can be applied to *all* memory pages, rather than just the VMSA. Finally, CipherSteal [74] and HyperTheft [73] show that attackers can exploit ciphertext side channels to recover input data and weights from TEE-shielded neural networks respectively.

Hardware Attacks. DRAMA demonstrates that by observing the latency of DRAM accesses, an attacker can infer whether the victim has recently accessed the same row [57]. Attacking older client-oriented Core series CPUs, Membuster [34] shows how an attacker can use a professional $170,000 memory interposition setup to extract fine-grained memory access patterns of hardware enclaves. However, due to the cryptographic protections used by older SGX systems, Membuster is unable to fully breach SGX and recover the attestation key. Next, Buhren et al. [4] present a voltage glitching attack where adversaries can execute custom SEV firmware on the AMD-SP, allowing them to decrypt a VM's memory as well as extract endorsement keys while VoltPillager [6] demonstrates a similar attack where the attacker can control the CPU core's voltage to mount fault-injection attacks to breach the confidentiality and integrity of SGX enclaves. Finally, BadRAM [12] enables memory aliasing attacks by modifying the DIMM's SPD data, sidestepping SEV's aliasing checks. This results in a DIMM where two physical addresses are mapped into the same memory cells, thereby allowing [12] to break SEV attestation via replaying launch token digests.

2.4 Caches

To overcome the increasing performance gap between CPUs and memory, processors employ small buffers called *caches*. Caches exploit locality by storing recently and frequently used data closer to the CPU to hide the memory's access latency. Modern Intel processors implement a hierarchy of caches, with a shared last-level cache (LLC).

Cache Organization. In a typical Intel CPU, the caching hierarchy consists of three levels. First, each core has two L1 caches, one for data and one for instructions. Next, the core also has a unified L2 cache. Additionally, the CPU has a last level cache (LLC), which is shared between all of the cores. When accessing memory, the processor first checks if the data is in L1. If not, the search continues down the hierarchy. Caches consist of multiple cache sets that each host a number of cache lines or *ways*. To provide Quality of Service, some Intel processors feature Intel Cache Allocation Technology (CAT) [19], which can be used to allocate a different number of ways to each CPU core, and consequently to each guest virtual machine, to prevent trashing, therefore improving performance.

Cache Attacks. Timing access to memory can reveal information on the status of the cache, giving rise to side-channel attacks, which extract information by monitoring the cache state. Previous works proposed many different techniques to perform such attacks, most notably Flush+Reload attacks [15, 72] and Prime+Probe attacks [27, 32, 37, 55, 56].

2.5 Memory

Memory Organization. The memory system is organized into multiple channels, each potentially with multiple DIMMs. A single DIMM consists of a number of RAM chips, organized into ranks, banks, rows and columns, which uniquely define the address of each byte available within the DIMM. Within a chip, each die contains an array of capacitors, with each capacitor corresponding to a single bit. In the case where the system contains multiple DIMMs, DIMMs on different channels can be accessed in parallel, while modules sharing a channel must be accessed sequentially. Finally, Intel CPUs use proprietary addressing functions to map the physical address space onto DRAM locations, with prior works [28, 57] reverse engineering these functions for common processors up to 4 DIMM configurations.

DDR4 Memory Bus Overview. Each DDR4 DIMM contains 288 pins, which are used by the CPU's memory controller to issue DIMM commands as well as send and receive data read and written from the system's physical memory. From these pins, DDR4 ECC-RAM uses 24 pins for address and commands, as well as 72 bits for data transfer. This includes 64 bits of data and 8 bits of ECC encoding. Data is sent after a certain amount of clock cycles, known as the DIMM's CAS value. See [30] for additional details.

Memory Bus Speed. The operations of the DRAM bus are governed by a clock signal generated by the CPU's memory controller for all modules present in the system. DIMM commands take one clock cycle, and are only transmitted on rising clock edges. On DDR (double data rate) memory, each data burst also takes a single cycle, but with bursts being transmitted on both rising and falling clock edges. This effectively doubles the DIMM's data transferring speed, while allowing for slower (and thus more reliable) transmission of DIMM commands. Finally, the DDR4 JEDEC standardizes speeds of 1600 MT/s (mega transfers per second) til 3200 MT/s, using 800–1600 MHz clock frequencies, respectively [31].

Serial Presence Detect (SPD). The SPD is an additional chip present on DIMM modules, which contains the module configuration data written to non-volatile memory. This data includes values such as the DIMM's memory size and supported speeds (among others). Communication with the SPD chip is done via two dedicated pins, and is implemented via the I^2C protocol [29]. During boot, the SPD of all DIMMs present in the system is read by the BIOS, and is used to configure the system's memory controller. Finally, the DIMM's SPD data is programmable either via BIOS or via external SPD programming boards, and is often used by overclockers to speed up memory access times by forcing the DIMM into higher speed regimes.

3 Threat Model and Target Setup

Adversarial Capabilities. We assume a threat model where the attacker has physical and supervisor access to the target machine. While SGX was not designed to protect against physical attacks, in Sections 7 and 8 and Appendix B we instantiate this threat model by examining realistic Web3 SGX deployments.

More specifically, we assume that the adversary is capable of running a custom modified kernel, manipulating memory management in an adversarial manner, and changing page permissions. For physical access, we assume capabilities of a trained computer technician, capable of modifying and installing parts into the target machine. However, we do not allow for a more advanced adversarial capabilities such as silicon-level circuit editing (e.g., using a focused ion beam machine). Finally, we assume a modest attack budget of under $1000 USD, putting our work within reach of most hobbyists.

Target Machine Configuration. For the target machine, we use a computer equipped with an Intel Xeon Silver 4310T (Ice Lake) CPU and a Supermicro X12SPL-F motherboard. Next, as enabling SGX requires all memory controller DIMM slots to be populated, our machine was equipped with eight 16 GB sticks of DDR4 RAM. For software, our machine is using `Ubuntu 24.04.2 LTS (noble)`, with kernel version `6.10.6`. Finally, we used BIOS version `2.2` containing CPU microcode version 0xd0003f5, allowing our machine to pass DCAP attestation in a fully trusted status.

4 Building a Cheap DRAM Interposition Setup

We now describe the construction of our hardware acquisition setup, as well as its capabilities to observe traffic on the DRAM bus.

4.1 Slowing Down Bus Transactions

The cost of data acquisition equipment is often governed by three main factors: the number of input channels, the amount of memory available to store acquired data, and the maximum supported acquisition speed. While our attack does not require a large amount of memory as we are only interested in observing individual transactions at a time, slowing down the system's bus speed will enable us to use simpler and readily available acquisition hardware. To that aim, in this section we describe our setup for modifying the DIMM's SPD parameters, allowing us to produce slow operating DIMMs.

Experimental Setup. While prior work [12] demonstrates how to construct an SPD writer for as low as $10, we instead elected to use a pre-built unbranded SPD writer, available on e-commerce websites for $75. We then used the supplied software to modify the SPD of a Micron MTA18ASF2G72PZ-2G3B1QG 16 GB DIMM rated for 2400 MT/s, experimenting with different speed settings. See Figure 2. Finally, while [12] creates memory aliasing by changing the DIMM's SPD to indicate a larger capacity, our work uses the DIMM's SPD to report lower supported speeds (thus slowing down bus transactions), leaving the DIMM's reported capacity unchanged.

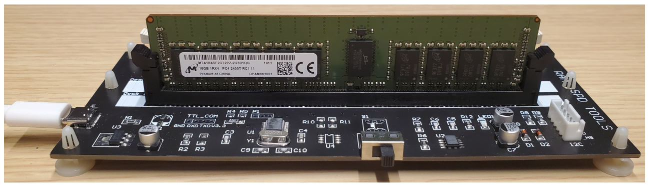


Figure 2: SPD Writer for modifying DIMM SPD values

Results. While the JEDEC standardizes DDR4 speeds in the range of 1600–3200 MT/s [31], using 800–1600 MHz clock frequencies, we were unable to find any DDR4 server ECC RDIMMs operating below 2133 MT/s (1066 MHz clock). However, by modifying our DIMM's SPD values, we have empirically observed that our system will still boot even when the DIMM's speed is set to 1333 MT/s. Moreover, we have observed that installing a single such slow DIMM has the effect of slowing down all the DIMMs in the system to 1333 MT/s, presumably as the CPU's memory controller only generates a single clock signal for all the DIMMs present in the system. Next, as the DIMM's SPD parameters are not part of SGX's TCB, using this method we were able to produce a fully trusted DDR4 system which only uses a 667 MHz clock for its memory bus, greatly simplifying our acquisition setup.

4.2 Building a DDR4 Bus Interposition Setup

Having obtained a slow running DIMM, we now build a bus interposition probe to physically connect our logic analyzer to the system's DRAM bus. While prior work [12, 34] argues that such probes are rare and expensive, requiring tens of thousands of dollars, we show how such a probe can be constructed by hobbyists, costing under $1000.

Starting Point: A DIMM Slot Protector. Our starting point is a DDR4 DIMM riser board, which is a simple pass-through PCB consisting of a female DDR4 DIMM socket and an edge connector, see Figure 3(left), which provides easy physical access to the DIMM's traces. With the DIMM's traces accessible, ideally the most basic construction of a DDR4 interposer would be to simply attach an additional wire to each such trace, connecting the other side of the wire to the logic analyzer input. However, such a design forces the system to drive signal to both the DIMM and the wires leading to the logic analyzer. This in turn overloads the driver built into the CPU, preventing the system from booting (in some cases permanently, as we empirically discovered, by damaging the CPU's ability to detect RAM).

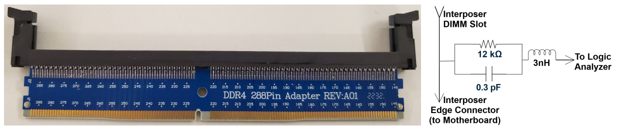

DDR4 288Pin Adapter REV:A01
Interposer DIMM Slot
12 kΩ
3nH
To Logic Analyzer
0.3 pF
Interposer Edge Connector (to Motherboard)

Figure 3: (left) DDR4 riser board. (right) Probe isolation network schematic

Step 2: Designing a Probe Isolation Network. Aiming to mitigate this issue, we observed the construction of Keysight's SoftTouch series probes, intended for high frequency buses. Here, we discover that each of the probe's channels contains 3 components: an inductor connected in series to a capacitor and a resistor, with the last two components connected in parallel, see Figure 3(right). A particularly helpful example of such construction was a Keysight (formally known as Agilent) N4252A Transition Probe Adapter, which contains 102 channels, all outfitted with such a design, and available on a second hand marketplace for around $40 at the time of purchase. While documentation about the N4252A is scarce, writing on our probe suggests it was used for debugging Intel's QuickPath Interconnect protocol, used by Intel between 2008 and 2017 [69]. Moreover, while using very different pinouts, QPI is very similar to DDR4 memory from an analog signal integrity perspective, with both protocols using both rising and falling clock edges, as well as a 3.2 GHz clock.

Step 3: DDR4 Interposer Construction. To build our DDR4 interposition probe, we used a hot air gun to heat up the N4252A's PCB, manually collecting the stacked pairs of capacitors and resistors using tweezers. Using a soldering iron, we then placed a capacitor-resistor pair on every signal carrying trace in the DIMM slot protector, allowing us to isolate the logic analyzer hardware from the target

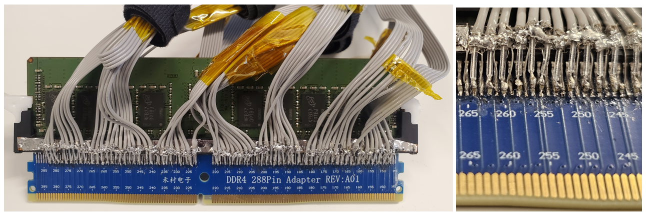


Figure 4: (left) Homemade DDR4 interposer. (right) Zoomed-in view on the probe isolation networks

machine. See Figure 4(left). This in turn allowed our machine to reliably boot, without reporting any memory reliability issues.

Step 4: Logic Analyzer Hookup. The final step is to connect the other side of the probe isolation network to the analyzer. To that aim, we used two Keysight N4834 SoftTouch probes ($22 each, second-hand), maintaining their top 90-pin pods and cutting off their bottom SoftTouch connector. We then soldered the signal wires for each channel directly to the terminals of the capacitor-resistor pairs placed earlier on the slot protector's PCB. This resulted in a reliable logic analyzer connection to the system's DDR4 bus, allowing us to observe signals at full DDR4 rates of 3200 MT/s without the target machine reporting memory reliability errors. See Figure 4(right). Moreover, our probe uses standard 90-pin Keysight logic analyzer pods, allowing it to be connected to a large variety of Keysight equipment.

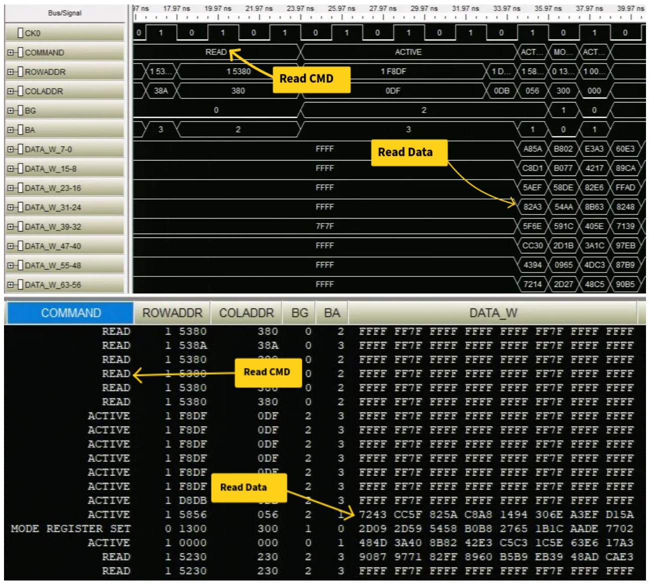
Step 5: Logic Analyzer Selection. With our probe successfully constructed, our next step is to obtain a suitable logic analyzer capable of tracking signals on the DDR4 bus. Here, we use our ability to slow the system's memory to 1333 MT/s (or 667MHz) from Section 4.1 to our advantage, allowing us to use highly obsolete (and thus cheap) Keysight acquisition hardware. More specifically, we used five Agilent 16950B acquisition cards, each featuring 4MB sample storage RAM, and capable of stateful acquisition of 667MHz signals by tracking the system's clock. The cards were placed inside a Keysight 16902A logic analyzer chassis, equipped with a Pentium III 1GHz single core CPU and 512MB of system RAM. Purchased for $550 from a secondhand marketplace (including cards), we upgraded our chassis to use Windows XP SP3, replaced the original spinning drive with an SSD ($30) and installed Agilent's Logic and Protocol Analyzer application version 5.9, the latest supporting this generation. See Figure 1 for a picture of our end-to-end acquisition setup and Table 1 for a bill of materials and cost breakdown.

Item	Qty.	Total Cost
SPD writer	1	$70
DDR4 Riser	1	$15
Agilent N4252A Probe	1	$40
Keysight N4834 Probe	2	$44
Agilent 16902A Chassis + Five Agilent 16950B Modules	1	$550
256GB Sata SSD	1	$30
Sata to IDE Adapter	1	$8
Lab supplies (solder, flux, etc.)	1	$100
Total		**$857**

Table 1: Bill of Materials for our DDR4 Bus Interposition Setup

4.3 Observing Bus Transactions

Figure 5(top) shows an example of acquired read transactions, as captured by our logic analyzer, which is then parsed into a listing in Figure 5(bottom). As can be seen, the logic analyzer allows us to observe a significant amount of information about the CPU's memory activity, including memory commands, addresses, as well as read/write data.



```
          COMMAND  ROWADDR  COLADDR  BG  BA  DATA_W
             READ   1 5380      380   0   2  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
             READ   1 538A      38A   0   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
             READ   1 5380      380   0   2  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
             READ   1 5380      380   0   2  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
             READ   1 5380      380   0   2  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
             READ   1 5380      380   0   2  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 F8DF      0DF   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 D8DB      0DB   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
           ACTIVE   1 5856      056   2   1  7243 CC5F 825A C8A8 1494 306E A3EF D15A
MODE REGISTER SET   0 1300      300   1   0  2D09 2D59 5458 B0B8 2765 1B1C AADE 7702
           ACTIVE   1 0000      000   0   1  484D 3A40 8B82 42E3 C5C3 1C5E 63E6 17A3
             READ   1 5230      230   2   3  9087 9771 82FF 8960 B5B9 EB39 48AD CAE3
             READ   1 5230      230   2   3  FFFF FF7F FFFF FFFF FFFF FF7F FFFF FFFF
```

Figure 5: (top) Waveform View of Acquisition Data and (bottom) Parsed Listing View of Acquisition Data

5 Controlling Enclave Memory Layouts

Having obtained the capability of observing activity on the DRAM bus of a single DIMM, we recall that to enable SGX, Intel requires that all channels for all memory controllers present in the system be populated [21]. Overall, we must configure our system to use eight DIMMs, each being 16 GB. However, as our logic analyzer setup is able to observe data on only a single DDR4 DIMM, we need to overcome several challenges in order to ensure that the enclave's memory traffic is indeed directed to the DIMM connected to the logic analyzer.

*C*1 First, we must be able to map system physical addresses to DRAM addresses and their corresponding DIMMs.

*C*2 Next, we must ensure that the enclave page of interest is mapped to the DIMM connected to the logic analyzer.

*C*3 Finally, we must control the execution of the target enclave, ensuring its ciphertexts are observable on the logic analyzer.

5.1 Reverse Engineering DRAM Addressing

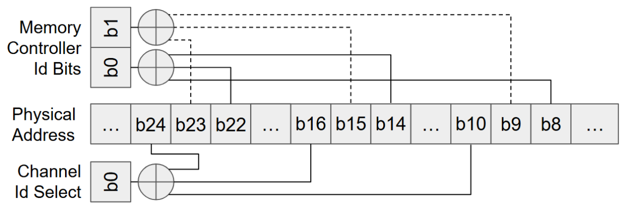
In order to map physical addresses to their corresponding DRAM addresses we must recover the DRAM addressing function. Prior works [28, 57] rely on the row-buffer conflict side channel to detect physical addresses mapped to the same memory bank, and based on that information solve for the addressing function. However, as the addressing functions grow more complex and the number of memory channels and DIMMs increases, two separate problems emerge: the row buffer side channel becomes noisier, and deliberately inducing these conflicts becomes much harder. Rather than using row buffer conflicts, we instead develop an approach using Intel's Memory Address Translation (ADXL) ACPI methods [20]. While this has the benefit of recovering DRAM addressing functions without the need to induce row buffer conflicts or deal with system noise, this method is only applicable on platforms that support ADXL (e.g., Intel Xeon Scalable processors).

Memory Address Translation. Recent BIOSes supporting Intel CPUs provide an ACPI device-specific method (DSM) for decoding physical addresses to DRAM addresses. Intel BIOS reference code contains address translation functions, but these are not directly accessible by the OS [20]. Instead, the DSM interface allows the OS to query decoding of physical addresses and receive their DRAM address components. Finally, the Linux kernel has implemented an interface for accessing ADXL methods, which is normally only triggered when decoding memory error information [38].

Reverse-Engineering using ADXL Feedback. To reverse engineer the machine's DRAM address functions using feedback from the ADXL mechanism we implemented a simple kernel driver directly exposing this interface to userspace. Next, using the ADXL methods, we have the ability to decode specific physical addresses and observe information on their DRAM address (memory controller ID, channel ID, bank group and address, row address), which are required to precisely determine an address's DRAM location. With this primitive, we recover DRAM addressing functions as follows.

We start with an arbitrary valid `baseAddr`, and decode it into its components. Then, for each set of n bits $S = \{i_1 ... i_n\}$, we decode the physical address `baseAddr` $+ 2^{i_1} + ... + 2^{i_n}$ and compare their resulting DRAM address components. Any differences in the DRAM addressing must be due to the differences in one or more i_x, allowing us to include S in the set of bits that affect this part of the DRAM address. For example, if the jth bit of the bank group is different between the two addresses, then some i_x is a bit used to calculate j. At the end we are left with a set of physical address bits that affect each bit of the DRAM address. Next, we calculate which of these relationships are linear via Gaussian elimination. For those that are linear, we then directly assign physical address bits to DRAM address bits. For the non-linear relationships, we proceed to populate a truth table using each physical address bit as inputs and each affected DRAM address bit as outputs. Finally, we minimize the truth table, resulting in functions that can be used to derive the DRAM address bits.

See Figure 6 for an example of a recovered DIMM selection function, as well as Appendix A for additional functions recovered via our method.



$$\text{Row}_{11} \longleftarrow \neg b_{31} \wedge (b_{32} \vee b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$$

Figure 6: DIMM Selection Functions on our SGX Machine

5.2 Pinning SGX Memory

Now that we can translate system addresses to their corresponding DRAM addresses, we need to ensure that we can map our target enclave page of interest to the DIMM that is attached to the logic analyzer. While previous works have used the `ewb` (Enclave Write-Back) and `eldu` (Enclave Load) leaf instructions to swap out enclave pages and reload those [3, 65, 67, 68], we instead resort to modifying Intel's SGX driver to provide a pinning mechanism. More specifically, we use the address translation functions recovered in Section 5.1 to determine which EPC pages map to the single DIMM whose traffic can be inspected by our logic analyzer. We then reserve these pages by filtering them from the driver's internal linked list structure used to track free pages. Although the OS cannot directly access EPC pages, it is responsible for allocating them to different enclaves running on the system. This is not considered a security risk as the processor-controlled EPC Map ensures valid mappings by the OS between EPC pages and enclaves, preventing aliasing attacks by the OS. In addition, we modify the flag variable of these pages to ensure that the driver's garbage collection and sanitization processes ignore these reserved pages. When allocating memory, the SGX memory allocator provided as part of the driver simply inspects whether the request originates from one of our target enclaves and whether the virtual address is included in the list of of targeted addresses, deciding whether to back the request using a physical page located on an interposed DIMM or not.

5.3 Forcing DRAM Traffic

With the ability to pin the target enclave page of interest to the DIMM attached to the logic analyzer, the final challenge is to ensure the enclave's memory accesses miss the CPU's cache, allowing us to observe the ciphertexts corresponding to the enclave's data via the logic analyzer. To overcome this hurdle, we need to be able to control the enclave's execution flow and then evict the enclave's data from the cache such that it is observable on the memory bus.

Enclave Control Channel. To that aim, our first step is to obtain the ability to halt the enclave at points of interest. As Intel SGX prevents debugging production enclaves with tools such as gdb, we rely on controlled-channel attacks [71] to control the execution flow of production enclaves. More specifically, we use `mprotect` to change the protection of the target enclave page of interest, such that accessing it causes a segmentation fault. Although the OS cannot directly access EPC pages, `mprotect` is able to modify the permissions of pages in the OS-controlled page table. We can then rely on `ptrace` to temporarily change the protection of our victim's pages, halt the enclave as it triggers a segmentation fault and then revert the protection to resume execution. This allows us to follow the execution of the enclave at a page granularity until we reach our points of interest.

Overcoming Caching. As the CPU keeps the enclave's data cached rather than directly sending it out to DRAM, the resulting ciphertexts are not immediately visible on the logic analyzer. Thus, to be able to observe the ciphertexts on the memory bus, we have to ensure that the CPU evicts the relevant cachelines from the cache. While it is normally possible to mark specific pages as uncacheable, the Processor Reserved Memory Range Registers (PRMRRs) control the cacheability of the EPC pages [9]. As we cannot override this cacheability, we can neither mark individual EPC pages or individual pages as uncacheable. Therefore we have to resort to other means of evicting the data of interest from the CPU's caches.

As discussed in Section 2, many works have demonstrated sophisticated cache attacks such as Flush+Reload [15, 72] and Prime+Probe [27, 32, 37, 55, 56]. However, in this case we note the logic analyzer's triggering functionality, which (among others) allows the logic analyzer to begin acquisition only when a certain DIMM location is accessed. Thus, we can simply resort to trashing the entire cache, relying on the logic analyzer to initiate acquisition for specific addresses. Implementing this, we run a process on a sibling CPU core that traverses an array that is twice the size of the LLC, thus evicting the victim's data from the LLC. Finally, to reduce noise from irrelevant data, and to more easily evict the victim's cache lines, we rely on Intel CAT to minimize the number of cache ways that the victim process can use, thus reducing the amount of irrelevant DRAM data.

6 Attacking SGX Encryption

As discussed in Section 2, Intel server platforms implement SGX on top of TME, which encrypts the entire address space using AES-XTS with a key determined at boot time [22]. Encryption is done by having the memory controller write ciphertexts to memory instead of plaintext data, at a granularity of 128-bit blocks. Furthermore, AES-XTS in TME includes a tweak function incorporating the physical address.

6.1 Verifying Determinism

To verify Intel's use of deterministic encryption as well as to empirically observe its behavior, we selected a fixed virtual address and a physical address located in the Enclave Page Cache (EPC) that would be mapped to the DIMM interposed by our logic analyzer. Within an enclave, as shown in Listing 1, we executed a sequence of memory accesses involving writes after reads. After the second read, we modified the data by incrementing it by one and wrote it back to memory. Then, following the third read, we decremented the data by one, restoring it to its original value before the final write.

Observing Determinism. Figure 7 below shows (a filtered) output from our logic analyzer. As can be seen, the ciphertext data written to memory is indeed deterministic, with the values of the first and third memory reads being the same. Meanwhile, the value in the second read is completely different, corresponding with an encryption of a different data value on our program's second iteration.

Behavior on Virtual Address. To examine the effects of the virtual address on SGX's encryption, we prepared two identical arrays inside enclaves located at different virtual addresses as illustrated in Figure 8(left). We adjusted the page offsets so that both arrays would appear in the DIMM connected to the logic analyzer. Using our modified SGX driver, we mapped these arrays to the same physical

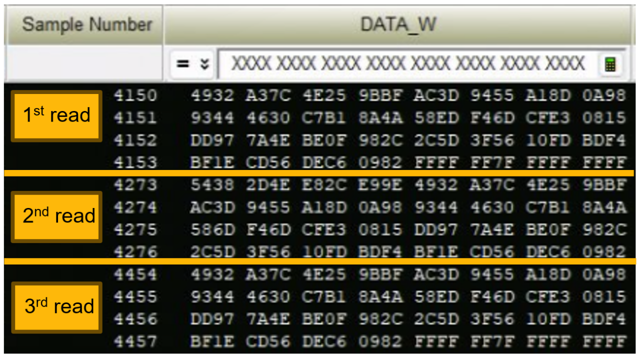

```
Sample Number   DATA_W
             =  XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
1st read   4150   4932 A37C 4E25 9BBF AC3D 9455 A18D 0A98
           4151   9344 4630 C7B1 8A4A 58ED F46D CFE3 0815
           4152   DD97 7A4E BE0F 982C 2C5D 3F56 10FD BDF4
           4153   BF1E CD56 DEC6 0982 FFFF FF7F FFFF FFFF
2nd read   4273   5438 2D4E E82C E99E 4932 A37C 4E25 9BBF
           4274   AC3D 9455 A18D 0A98 9344 4630 C7B1 8A4A
           4275   586D F46D CFE3 0815 DD97 7A4E BE0F 982C
           4276   2C5D 3F56 10FD BDF4 BF1E CD56 DEC6 0982
3rd read   4454   4932 A37C 4E25 9BBF AC3D 9455 A18D 0A98
           4455   9344 4630 C7B1 8A4A 58ED F46D CFE3 0815
           4456   DD97 7A4E BE0F 982C 2C5D 3F56 10FD BDF4
           4457   BF1E CD56 DEC6 0982 FFFF FF7F FFFF FFFF
```

Figure 7: Observing SGX's deterministic XTS encryption

address in EPC and observed that the encrypted data on the bus was identical for both. This confirmed that the virtual address does not contribute any tweak functionality to the encryption.

Behavior on Physical Address. Next, to check whether the physical address has an effect on encryption, we chose two different physical addresses (both in EPC) mapped to the DIMM connected to the logic analyzer. Using our modified SGX driver, we mapped two identical arrays inside enclaves to these addresses, see Figure 8 (right), and observed that the encrypted data on the bus was different. This confirms that there is a physical address tweak for memory encryption and identical data only varies in physical memory if the physical address is different.

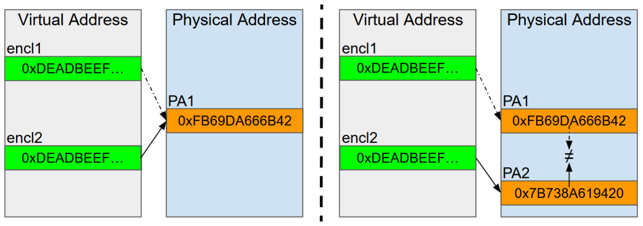


Figure 8: Encryption behavior of different enclaves with identical data to same (left) and different (right) physical address.

6.2 Attacking SGX Attestation

Having empirically confirmed Intel's use of deterministic XTS encryption for enclave memory, in this section we proceed to outline the necessary details of Intel's implementation of DCAP attestation, with key extraction attacks presented in Section 6.3.

DCAP Signature Chain. Before provisioning machines with secrets, clients must first verify that the requester is the intended enclave running in production SGX enclave mode on a machine in a fully trusted attestation status. To accomplish this, enclaves go through a remote attestation process. More specifically, a client requests a quote, which is a report of the enclave state signed by a Quoting Enclave (QE). The quote contains the signed enclave report, a signed report of the QE, and relevant certificates to verify the signatures. The QE is signed by a Provisioning Certification Enclave. This signature chain ends with a signature made by Intel's private key. Clients can verify the quote by verifying the signature chain. See Section 2.2 for a more extended discussion.

Listing 1 Enclave code performing a pattern of memory accesses to understand deterministic encryption behavior

```
sgx_ecall_access_memory(usize_t offset, uint64_t *data) {
    uint64_t mod[3] = { 0, 1, (uint64_t)-1 };
    for (i = 0; i < 3; i++) {
        *data = read_uncached(enclave_memory + offset);
        *data += mod[i];
        write_uncached(enclave_memory + offset, *data);
    }
}
```

Elliptic-Curve Digital Signature Algorithm (ECDSA). The signature scheme used in DCAP is ECDSA over curve p-256. Given a generator G of a group of order n over p-256, key generation consists of generating a random integer $1 \leq d \leq n-1$ and computing $Q=[d]G$ using a scalar by point multiplication operation. Signing a message m is done by first hashing m and subsequently converting the first $\lceil log_2 n \rceil$ bits into an integer z. Next, a random nonce $k \in \{1,n-1\}$ is generated and multiplied with G using scalar by point multiplication, resulting in $(x,y)=[k]G$. The signature (r,s) is computed by setting $r=x \bmod n$ and $s=k^{-1}(z+rd) \bmod n$. Finally, verifying a signature (r,s) on m is done by computing z as above, computing $w=s^{-1} \bmod n$, $u=zw \bmod n$, $v=rw \bmod n$, $(x,y)=[u]G+[v]Q$ and then checking that $x \equiv r \pmod{n}$.

Scalar by Point Multiplication. While other leakage sources might exist, we focus on the scalar multiplication as implemented by the Intel IPP library, used during SGX DCAP attestation. Algorithm 1 is (simplified) pseudocode of IPP's scalar by point multiplication routine. Here, the nonce k is converted into a series of booth-recoded digits $k_0,\cdots,k_n$, with each $-16 \leq k_i \leq 16$. Given a group generator G, in lines 1-4, the algorithm constructs a pre-computation table Q, where each $Q[i]=[i]\cdot G$, for $i=0,\ldots,16$. Next, in line 5, Algorithm 1 initializes an accumulator A, setting A to be $[k_n]G$. Next, for each booth-recoded digit k_i of k, the accumulator is multiplied by 2^5 and the value of $[|k_i|]G$ is stored in a temporary variable B, being negated in case k_i is negative (line 9). The value of B (which is now equal to $[k_i]G$) is then added to the accumulator A (line 10), allowing the algorithm to proceed to handling the next nonce digit. Finally, Algorithm 1 returns A once all digits of k have been processed. Inspecting IPP's implementation, we find that all memory accesses and control flow operations appear to be performed in constant time, without any immediate microarchitectural side channel weaknesses.

Algorithm 1 Simplified representation of scalar by point multiplication in Intel's IPP library

```
Input: A scalar $k$ where $k_0 \ldots k_n$ are booth-recoded digits of $k$ with
    each $-16 \leq k_i \leq 16$ and a group generator $G$.
1:  $Q[0]=0$
2:  $Q[1]=G$
3:  for $i=2$ to 16 do
4:      $Q[i]=Q[i-1]+G$
5:  $A=Q[k_n]$                                  ▷ done in constant-time
6:  for $i=n-1$ to 0 do
7:      $A=2^5 A$                               ▷ using 5 point doubling operations
8:      $B=Q[|k_i|]$                            ▷ done in constant-time
9:      if $k_i<0$ then $B=-B$                  ▷ done in constant-time
10:     $A=A+B$
11: return $A$
```

6.3 Recovering DCAP Attestation Keys

To recover the machine's ECDSA attestation key d, it is sufficient to recover the nonce k used during a scalar by point multiplication operation during a single attestation attempt. With the nonce k as well as the signature (r,s) on a message m in hand, it is possible to obtain d by solving $s=k^{-1}(z+rd) \bmod n$, see Section 6.2.

Constructing a Ciphertext Dictionary. To recover k, we first construct a ciphertext dictionary mapping the values of $[k_i]G$ to their corresponding ciphertexts, for all possible values of k_i. Moreover, as the ciphertexts produced by SGX are address dependent, we fix a specific physical address addr, which is also mapped to the DIMM interposed by our logic analyzer. Noting that k_i ranges from -16 to 16, we created a simple dictionary building (DB) enclave, which computes $[t]G$ for all possible 33 values. Our enclave then writes each $[t]G$ to addr, forcing the enclave to use addr via the pinning mechanism described in Section 5.2. We then flush addr out of the machine's cache, thereby inducing a DRAM write transaction. As addr is mapped to a DIMM interposed by our logic analyzer, we are able to observe the ciphertext c_t corresponding to $[t]G$, allowing our enclave to proceed to the next value of t. Overall, using this method we are able to obtain 33 ciphertexts $c_{-16},\cdots,c_{16}$, corresponding to all possible values of $[t]G$ as written to memory to address addr.

Nonce Recovery. Having constructed our ciphertext dictionary, we now proceed to recover an ECDSA nonce k used to sign a report m, resulting in a quote containing an ECDSA signature. To that aim, we launch the machine's quoting enclave in production mode, triggering an attestation request via Intel's sample DCAP attestation program [24]. Using the mechanism from Section 5.2, we ensure that the variable B from line 8 of Algorithm 1 lands on the physical address addr used to generate the ciphertext dictionary. Next, we use the approach from Section 5.3 to halt the quoting enclave's (QE) scalar by point multiplication operation at line 10 of Algorithm 1, namely at the end of each iteration of the main multiplication loop. By trashing the machine's cache at this point, we ensure that the memory access reading the value of B misses the caching hierarchy, triggering a DRAM read operation. Since addr is mapped to a DIMM interposed by our logic analyzer, we are able to observe the ciphertext corresponding to the value of B for each iteration of the main loop of Algorithm 1.

With the encrypted values of B in hand, we note that Intel's SGX implementation uses the same memory encryption key for all enclaves present in the system. In particular, our DB enclave uses the same TME memory encryption key as Intel's own QE. Thus, as both the QE and our DB enclave accessed the same physical address addr, we are able to use the dictionary constructed earlier, and map the encrypted values of B to their corresponding values of $[k_i]G$. With the booth-recoded digits $k_0,\cdots,k_n$ of k in hand, we are able to completely recover the nonce k used for producing the ECDSA signature during the machine's attestation process.

Attestation Key Extraction. Having obtained the ECDSA signing nonce, our next step is to recover the machine's attestation key. To that aim, we notice that unlike the attestation flow used on prior SGX implementations running on client machines [11], the SGX DCAP attestation flow does not encrypt the attestation quote before sending it to the DCAP attestation server. Thus, by observing the signed quote, we are able to obtain the values r and s which form the quote's ECDSA signature. Finally, we obtain the key d by solving $s=k^{-1}(z+rd) \bmod n$, where z is a hash of the machine's quote.

Experimental Results. To obtain an SGX attestation key from a machine in a fully trusted status, we executed Intel's sample DCAP attestation program on our target machine from Section 3. Triggering a DCAP attestation, we used the above-described approach to successfully recover 51 out 52 booth-recoded digits $k_0,\cdots,k_n$ of k. As the first booth-digit k_n of k is not accessed during the main loop (line

6) of Algorithm 1, we have iterated over all possible 33 options and attempted ECDSA key recovery for each, until the correct key was found. Overall, our nonce extract phase took 50 seconds per nonce digit (~ 45 minutes total), with the main bottleneck being the slow IO speed of our Pentium III based logic analyzer. The subsequent key extraction step was completed using an MacBook Air (Apple M1, 16GB RAM), taking about 16.44 milliseconds. To the best of our knowledge, this is the first end-to-end key extraction attack on an SGX system using DCAP attestation from a machine in a fully trusted status.

7 Breaching Confidentiality of Enclave Data

Equipped with an attestation key from a machine in trusted status, we proceed to investigate how real world SGX deployments fail in the presence of such key compromise. Here, we noticed that numerous systems which rely on SGX for confidentiality fail in a similar manner, allowing attackers equipped with an attestation key to forge quotes and extract data at will. We focus here on two primary SGX deployments, the PHALA network and SECRET network, and due to space constraints discuss INTEGRITEE in Appendix B.

Both PHALA and SECRET provide confidential smart contract execution in SGX enclaves. Smart contracts are stateful programs in blockchain systems with which users can interact. By only encrypting and decrypting data with keys available inside enclaves, these networks ensure that only the enclave should have access to the contents of contracts and data they operate on. However, we find a compromised attestation key allows for breaking this confidentiality of execution. We show how an attacker can extract multiple secret keys used for encryption from the networks by modifying enclaves with a forged quote. With these keys, an attacker can decrypt all confidential transactions on the networks without being detected.

Finally, we also demonstrate how even without attestation keys, one can directly extract secrets from within enclaves and thus break confidentiality. We show this by directly recovering a private ECDSA signing key from an enclave, without mounting attacks on the machine's attestation process.

7.1 PHALA Overview

PHALA provides confidential computation on top of a blockchain. PHALA users can request execution of arbitrary *phat* (smart) contracts, which are then executed off-chain in SGX enclaves. The blockchain itself ensures consensus of transactions and validates the results of this confidential contract execution [46]. Enclaves are also used for key derivation, distribution, and storage. Users running enclaves are rewarded with tokens on the blockchain. If consensus detects malicious behavior such as mis-execution of a contract, workers are punished by reducing rewards [45]. However, there is no method for detecting confidentiality attacks such as a breach of contract data.

Node Registration. PHALA uses signing and encryption keys generated in SGX enclaves to prove that messages come from an enclave running trusted code. It uses the Gramine library to run its code under SGX and produce a DCAP quote on request containing a worker's public key (i.e., `WorkerKey`), which itself is generated and sealed by the enclave [47]. PHALA collects the collateral for the quote from a PHALA server and submits it to the blockchain along with the quote, which is verified on-chain. The `mrenclave` and `mrsigner` are checked against a whitelist, and if present, the worker is accepted. The worker is assigned a confidence tier based on the trust level of the attestation result, with tiers 1-3 being suggested for highly secure applications such as financial computations, and tiers 4-5 recommended for other use cases [44]. This ensures that messages signed with the `WorkerKey` come from a trusted enclave.

Phat Contracts. Confidential smart contracts executed in PHALA are known as *phat contracts*, which, in addition to privacy, provide wider capabilities than traditional smart contracts as they are executed off-chain in enclaves. Specifically, phat contracts are able to access public networks and other chain data [48].

While phat contracts are also submitted in the clear, their state remains encrypted in the SGX runtime. Execution requests for phat contracts are encrypted under their individual `ContractKeys` then submitted to the blockchain, providing confidentiality for contract inputs. Execution requests are directed to specific clusters of workers, which have independent contract data and state. Clusters may require different fees for execution of phat contracts, and each worker can only belong to one cluster. We note that at the time of writing, the main PHALA network only consists of a single cluster, whose admittance is controlled by the chain's governance system.

Secrets. Not all SGX enclaves perform the same role in the PHALA network. The most trusted workers are *gatekeepers*, which store a `MasterKey`, from which all `ClusterKeys` are derived [50]. In order to prevent a leak of the `MasterKey` compromising all historical data, it is periodically rotated. New gatekeepers are approved by vote on the chain, and gatekeepers never directly execute phat contracts. Other workers belong to clusters and store a common `ClusterKey`, which itself is used to derive individual `ContractKeys`. These cluster workers perform the actual execution of the phat contracts inside the enclave. When communicating off-chain during contract execution, workers use their `WorkerKey` to identify themselves and prove they are communicating from inside the enclave.

7.2 Removing the Layers of Phat Contracts

Having extracted an attestation key (Section 6.3), we now proceed to evaluate the resistance of PHALA to key compromises.

Node Setup. We use an HPE ProLiant ML30 Gen10 Plus server equipped with an Intel Xeon E-2334 CPU running microcode `0x63` and BIOS version `2.20`. Our machine is running Ubuntu 24.04 with Linux kernel `6.11.0-21-generic`. For software, we emulate our own custom quoting enclave on the system, using the attestation key recovered in Section 6.3. This allows us to create and sign arbitrary quotes, containing any data we choose. Moreover, by signing enclaves in debugging mode as having production status, we are able to perform arbitrary inspections of SGX enclaves running on the system without being detected by SGX's attestation mechanisms.

Network Setup. Running a gatekeeper with the highest tier of confidence on PHALA requires confidence tiers 1 to 3, which is equivalent to passing quote attestation in `CONFIGURATION_AND_SW_HARDENING_NEEDED` status or better [44].

We set up a local testnet based on the PHALA documentation, using officially provided Docker images to get official enclave builds [2, 49]. We first start a node with fresh chain state and use the same enclave whitelist as the main network. We then create two workers with the official signed enclave, assigning one as a gatekeeper, and creating a cluster with the other. Finally, we also create the attacker enclave.

Modifying the PHALA Enclave. In order to show the consequences of an enclave breach on PHALA's security guarantees, we modify the application code where workers are registered. After fetching the quote from our running enclave, instead of directly uploading this quote to the blockchain, we first overwrite the quote measurements with those of the officially signed enclave. We re-sign the quote with our emulated QE and upload the new forged quote, which always passes the whitelist and DCAP attestation. Then, we modify the enclave code to print out all secret keys when received.

Extracting Keys. We first enter our attacker enclave into a cluster and note it is given access to the cluster key. Although the cluster key is not directly distributed to our worker upon joining a cluster, we initiate a transfer of the key from any other node in the cluster. This transfer is completed without on-chain interaction, given our worker is part of the cluster. This cluster key can then be used to decrypt all contract interactions within the cluster. Finally, when our testnet accepted our node's enclave as a gatekeeper, we directly receive a copy of the master key, which is used to derive all cluster keys and therefore all contract keys, allowing us to decrypt the entire testnet.

7.3 SECRET Overview

SECRET Network. Similar to PHALA, SECRET is also a privacy-preserving smart contract system. SECRET network was one of the first TEE-based blockchains to reach significant adoption, launching its privacy-preserving smart contracts feature in 2020.

SECRET's Architecture. SECRET consists of an SGX-based smart contract execution layer adapted to run within an enclave, with an independent consensus layer. To send messages to smart contracts, users derive an encryption key from a master key (`io_master_key`), and include the ciphertext in a transaction. The corresponding private key, derived from the *consensus seed*, is replicated throughout the network and sealed by the SGX enclaves. To provide the ability to roll the consensus seed in the event of compromise, the SECRET network maintains both the initial and current consensus seeds.

Registering Validator Nodes. New validator nodes use remote attestation to register with SECRET network. First, the new node creates an ephemeral keypair for use with the Curve25519 ECDH (Elliptic-Curve Diffie-Hellman) key agreement scheme. More specifically, Diffie-Hellman lets the new node and any validator node already part of SECRET's network derive the same shared secret from their own private key and the other node's public key. This shared secret serves as the key to encrypt and decrypt the consensus seeds using 128-bit AES-SIV with the new node's public key as additional data. Next, the node creates an attestation report used to authenticate with the blockchain, which also contains the new node's public key. To join the network, the new node broadcasts a transaction containing the sender address of its wallet and the attestation report to the blockchain.

Existing nodes in the network observe this transaction and use their own enclave to verify the attestation report. If the checks pass, the node unseals the original and current consensus seeds, and encrypts them with the ECDH-derived shared secret key. Finally, the node updates the transaction with the concatenated ciphertexts as the `encrypted_seed`. Next, the joining node queries the blockchain for the `encrypted_seed` with its own public key to retrieve the ciphertexts, and decrypts each of them using the ECDH-derived shared secret key to retrieve the consensus seeds. Finally, it seals these consensus seeds inside its enclave to ensure confidentiality.

7.4 Extracting Secrets from SECRET

Setup. We utilize the same hardware setup and emulated quoting enclave from earlier. To setup our own validator machine with this, we ran the official unmodified SECRET binary on their Pulsar-3 testnet.

Masquerading as a SECRET Enclave. We first generate our own ephemeral keypair outside of the enclave, retrieve the DCAP quote from the local filesystem and replace the the public key in the DCAP quote with our own public key. We sign the DCAP quote with the extracted ECDSA attestation key and produce the combined attestation report, which we then broadcast onto SECRET's blockchain. After an existing node validates our attestation report, we can then proceed to retrieve the `encrypted_seed` for our public key. We can verify that our forged DCAP quote is indeed perceived as genuine by the existing nodes, as the `encrypted_seed` would otherwise not be available for our public key. Using the ECDH algorithm with our private key and the consensus exchange seed public key, we derive the shared secret key to decrypt the concatenated ciphertexts to produce the initial and the current consensus seeds. As AES-SIV provides authenticated encryption with associated data (AEAD), we can easily verify whether we have the correct shared secret key or not, as we would otherwise fail to authenticate, and thus not be able to decrypt the individual ciphertexts.

Decrypting Transactions. With access to the initial and the current consensus seeds we were able to directly decrypt any transactions on SECRET's test network, thus allowing us to completely breach SECRET's confidentiality guarantees.

7.5 Directly Attacking Enclave Secrets

Beyond attacks on DCAP attestation, we note that SGX's deterministic memory encryption allows us to attack enclaves directly, potentially extracting the secrets within them. Thus, hardening the Intel-supplied SGX attestation primitives is not sufficient to mitigate our attack. Demonstrating this, we attack OpenSSL's constant-time ECDSA implementation as described in Cipherleaks [36]. To that aim, we create a victim enclave that performs an ECDSA signing operation using the Intel SGX SSL library, which itself is based on OpenSSL version 3.0.14 [26].

OpenSSL Scalar by Point Multiplication. Recalling the description in Section 6.2 of ECDSA and how an ECDSA key can be recovered from a nonce k, we now focus on OpenSSL's implementation of scalar by point multiplication, which differs from Intel's IPP library used in Algorithm 1.

More specifically, Algorithm 2 shows simplified pseudocode of the Montgomery ladder scalar by point multiplication implemented in OpenSSL [54]. The algorithm takes an elliptic curve point p and scalar k with N bits. First, in Lines 1-2, the points s and r are initialized to $2p$ and p respectively, and a bit `prev` is initialized to 1. Lines 3-7 loop over the bits of k, setting i to the index of the current bit. Then, a swap bit is computed as `bit` $= k_i \oplus$ `prev` in Line 4. Next, the CONDITIONALSWAP method is performed on variable `bit` and points r, s. The CONDITIONALSWAP method swaps the memory contents of the points if and only if the bit is 1. We note the implementation ensures the conditional swap is done in constant-time, avoiding leakage via branching. Line 6 computes the Montgomery ladder step, updating $r = 2r$ and $s = r + s$. Next, the `prev` bit is updated as `prev` = `prev` $\oplus$ `bit`, and the loop continues or ends at Line 7. After the loop, Line 8 performs a final ConditionalSwap on r and s, using `prev` as the condition.

Algorithm 2 Simplified representation of scalar multiplication using a constant-time Montgomery ladder in OpenSSL.

```
Input: An elliptic curve point $p$, and an $N$-bit scalar $k$, where $k_i$
    is the $i$th bit of $k$.
1: $r = 2p$, $s = p$
2: prev = 1
3: for $i = N-1$ to 0 do
4:     bit = $k_i \oplus$ prev
5:     $r, s$ = ConditionalSwap(bit, $r$, $s$)
6:     $r = 2r$, $s = r + s$
7:     prev = prev $\oplus$ bit
8: $r, s$ = ConditionalSwap(prev, $r$, $s$)
9: return $r$
```

OpenSSL Key Extraction. First, we identify the memory location of the variable of interest, the encrypted values of which could be used to reveal the nonce k. While we use r going forward, we note that a similar attack could be done using s as well. Next, as our memory interposition setup can only monitor a single DIMM, we ensure that r is visible by pinning its page to an observable physical address (see Section 5.2). Moreover, as Line 5 of Algorithm 2 operates over a single conditional bit at a time, we avoid the use of the pre-built ciphertext dictionary enclave used in Section 6.3, instead setting up our logic analyzer to trigger on reads to r's address. We then run the victim enclave, overcoming caching as described in Section 5.3. We utilize our enclave control channel to interrupt the victim enclave before and after each conditional swap, and obtain the encryption of r at each point. In Algorithm 2, this corresponds to obtaining r twice—once at Line 4, and again immediately after the execution of Line 5 but before Line 6. Because of SGX's deterministic memory encryption, we know that if we obtain the same ciphertext twice, then a swap did not occur in Line 5 (and vice versa). With the swap information, we recover the value of `bit` for each iteration and thus the value of k. Finally, replicating [36], we have empirically demonstrated this attack using the interposition setup outlined in Section 4, extracting random nonces from OpenSSL-based ECDSA signing operations, thus compromising the signing key inside our victim enclave.

8 Compromising Integrity Guarantees

For our third case study, we focus on CRUST, a decentralized blockchain storage system that utilizes SGX enclaves to guarantee proof of storage of files on the network. More specifically, we look at how a compromised attestation key can affect the integrity guarantees of applications that solely rely on SGX to enforce correctness of execution. We demonstrate how an attacker can fake proofs of storage by running a modified enclave, forging a valid quote for the verified enclave's state. This forged proof allows an attacker to claim rewards for storage without actually storing the required data.

8.1 CRUST Network Overview

CRUST is a blockchain network that provides decentralized guaranteed storage of files built on the peer-to-peer IPFS file sharing protocol [41]. CRUST allows users to store arbitrary data, utilizing SGX to ensure data is stored honestly without trusting the storage operators. Storage merchants run storage services that consist of an IPFS node and an SGX enclave, which produces proofs of file storage, and a proof of available storage space. Nodes must stake currency as collateral while storing files in order to receive rewards [40]. Users execute storage orders on the blockchain with a request to store a file, and the first four storage merchants that produce a proof of storage of this file are immediately paid a portion of the storage fee. If the storage merchant fails to produce a proof of storage for the file regularly, they will lose collateral.

New Node Registration. The CRUST SGX enclave produces its proofs of storage by signing data with an ECDSA key (the node key) that is generated and stored solely inside the enclave. To register a new storage node on the network, the enclave first generates its node key and then creates a DCAP quote containing the `mrenclave` and key, and additionally signs the quote with this same key. This quote is sent to a trusted DCAP remote attestation server operated by CRUST which verifies the quote using Intel's quote verification library and returns a new signature over both the enclave's node key and its `mrenclave` identifier. The DCAP server's public key is pre-approved and stored by the CRUST blockchain as the root of trust, along with a whitelist of allowed `mrenclave` values. Assuming the SGX quote is legitimate and its signature chain can be verified up to Intel's certificate authority, this guarantees the node key was generated inside a trusted CRUST enclave binary.

Generating Proofs of Available Storage Space. The CRUST enclave produces proofs of storage space by signing messages with its generated node key. This proof is produced by the enclave by first storing a requested amount of sealed random data, with the enclave computing a cryptographic hash of the random data as it is being generated. To later prove the storage space exists, it re-computes the hash on the stored data. If the hashes match, then the enclave uses the node key to produce a message and signature attesting that the requested amount of storage physically exists. This message-signature pair is then broadcast to the network.

Generating Proofs of File Storage. The proof of file storage operates similarly. When a file is initially stored, it is hashed, and this hash is stored in an SGX-sealed database on disk. To later prove to others that the correct file is still stored, the enclave hashes the file again and verifies the hash has not changed. A recent storage report signed by the node key must be uploaded to the blockchain every 600 blocks, or files are considered lost by the worker. Similarly, if verification of a file fails, this is included in the report. Additionally, within each storage report, the enclave includes a hash computed over all stored file hashes. When a report is uploaded, this hash is verified by the chain, ensuring the contents of files are correct. If the storage state changes between reports, nodes additionally check the file state transition is valid. Finally, an off-chain worker known as the `spower` worker checks these reports and performs calculations to determine rewards and maintain storage records [42].

CRUST Integrity Guarantees. CRUST relies solely on the initial node registration and SGX attestation to guarantee that the node is running a trusted, unmodified binary which will correctly execute proofs. However, as seen earlier, an attacker with an attestation private key can sign arbitrary report information. This means that an attacker can modify the enclave code at will but produce a quote attesting that the enclave's `mrenclave` is any desired value (including that of a trusted CRUST binary). Similarly, an attacker could also register a node key generated outside the enclave. Either of these attacks completely violate the integrity of the CRUST system, allowing the attacker to forge proofs at will and collect illegitimate rewards.

8.2 Storing Gigabytes of Data with Only 32 Bytes

Setup. Running a storage merchant on CRUST requires one to show a DCAP-capable CPU where Intel's quote verification library does not reject the quote (i.e., that is not invalid or revoked). We use the same hardware setup and emulated quoting enclave from Section 7.2 to emulate this configuration.

We then setup our own local testnet based on the CRUST documentation and the latest officially published Docker images [1, 43]. We first start two nodes that peer with each other on a fresh chain state, and an API service to interact with the chain. Once we start our nodes, we set the allowed `mrenclave` and DCAP attestation server public key to the same as that of the main CRUST network. Next, we initialize an `spower` node to verify and calculate storage rewards, an enclave manager, and IPFS node. Finally, we start an enclave, and can send storage orders on the network.

Modifying the CRUST Enclave. To demonstrate the impact of an SGX breach on CRUST, we modify CRUST's application code by hooking the SGX library functions `sgx_qe_get_quote_size` and `sgx_qe_get_quote`. When CRUST requests a quote for a locally verifiable report, we overwrite the `mrenclave` identifier with the original signed enclave's. We then directly assemble and sign a quote with our attestation private key. Then, we follow the same steps as in Section 7.2 to produce a valid quote by utilizing a reference quote.

Forging Storage Proofs. Since the network always trusts the node's stated value of available storage space (as the integrity of this measurement should be guaranteed by the enclave's security properties), we report our node as having an arbitrary amount of storage space up to the network's limit of 2 PB. To produce proofs of storage, we must still store the hash of each file though. We modify the enclave to simply store the hash instead of the file's data. This hash is only 32 bytes regardless of the size of the file, which could be up to 32 GB as dictated by IPFS limits. With this, we can prove we store files without actually storing any of the file's contents.

Utilizing this, we instantly store any size file, beating honest storage merchants and claiming storage fee shares. Furthermore, we easily maintain proof of storage of an arbitrary amount of files, and thus will never be penalized or caught by the network's on-chain validation mechanisms. However, when a user tries to retrieve their file, they will not be able to get a copy from our IPFS node. Instead, another IPFS node (from the original four storing the file) will transparently serve the file to the user, leaving our attack undetected.

9 Mitigations and Future Work

Avoiding Deterministic Encryption. One of the fundamental issues we observed was the use of 128-bit AES-XTS deterministic memory encryption by modern Intel SGX server machines. Drawing inspiration from AEGIS [63, 64], older (and now deprecated) client SGX implementations used on-die Merkle trees to provide the EPC with confidentiality and integrity guarantees. Most importantly, these implementations provided *freshness* guarantees, designed explicitly to prevent attackers from observing the same ciphertexts when the same plaintext is stored at the same physical address [9]. Unfortunately, storing these Merkle trees also carries substantial overhead, which in turn limited the EPC size to 512 MB. Determining how to reconcile these goals and obtain scalable implementations of encrypted memory without sacrificing security guarantees, remains an open, but clearly necessary, research problem.

Accordion Cipher Mode. Recently, NIST has been considering proposals for accordion mode for block ciphers, which would allow for adjustable block sizes beyond standardized 128, 192 or 256 bits [53]. We note that these modes are unlikely to prevent dictionary attacks such as ours, as cryptographic algorithms typically do not have sufficiently high entropy in their internal state during individual loop operations. Thus, to prevent dictionary-based attacks, is imperative to ensure there is sufficient entropy inside each encryption block.

Quote Encryption. Prior client-oriented SGX implementations used a defense-in-depth countermeasure, which encrypted the EPID signature inside the attestation quote using the public key of the attestation server [11]. Doing so for Intel's DCAP protocol would mitigate our attestation key extraction attacks, as it would prevent us from obtaining the information necessary to convert our extracted nonce to an attestation key. We note however that this mitigation is far from a complete solution, as attackers can still potentially directly target sensitive data not related to attestation.

Faster Bus Speeds. Our attacks were made significantly simpler by being able to slow the machine's memory speed to 1333 MT/s, which is below JEDEC DDR4 specifications. While being far from a complete mitigation, imposing higher bus speeds such as 3200 MT/s, will raise the bar for low-budget bus-level attackers.

Permissioned Systems and Secure Multiparty Computation (MPC). A common issue we observed with many SGX deployments is that they are permissionless, meaning a node solely needs to present a trusted attestation status, with no additional context, to be entrusted with security-critical roles. This has the unfortunate implication that secrets are often physically placed in adversarial hands, resulting in their extraction. Thus, SGX developers should consider either limiting the location of their nodes to highly reputable cloud operators with strong physical security practices, or accept the possibility that enclaves will be breached occasionally. To mitigate the latter, MPC-based systems can be used to distribute trust among multiple parties, requiring their simultaneous breach for key extraction [70]. However, given the overhead resulting from MPC-based constructions, we leave the task of efficiently implementing them in SGX deployments to future work.

SGX as a Single Point of Failure. We observed that many systems relied on SGX as a root of trust, resulting in SGX being a single point of failure. Although some systems require permissions by barring registration from unknown sources, they still contain single points of failure such as provisioning a single master key to all SGX enclaves. This can be mitigated by designing the system to distribute trust, ensuring no single party is able to control a single point of failure.

AEX-Notify and Other Mitigations. Recently introduced on Intel platforms, AEX-notify allows enclaves to register a trusted handler to be run after an interrupt or exception, thereby thwarting deterministic single-stepping [8]. While making interposition attacks such as ours harder, we note that this still will not completely mitigate the issue, as memory accesses to deterministically-encrypted secrets are still performed during computations. These memory accesses in turn can be potentially intercepted without single stepping, albeit requiring more sophisticated LA triggers. Finally, software mitigations might be used to harden enclaves against bus interposition attacks. However, these require collaboration from Intel (to harden Intel-provided enclaves), and a suitable compiler for countermeasure deployment, as well as the re-compilation of all SGX software.

AMD SEV and Intel TDX. While our setup is transferable to AMD platforms which also use deterministic encryption, AMD SEV-SNP already allows the hypervisor to issue memory accesses for reading SEV ciphertexts. Indeed, this has been exploited by prior work [35, 36, 73, 74] to mount ciphertext attacks on SEV-SNP using software-only means, negating the need for our techniques. Finally, another application of deterministic encryption appears to be Intel Trust Domain Extension (TDX). However, as our current setup is only able to capture DDR4 traffic and with TDX only being available on systems which use DDR5 memory, we leave the task of constructing a cheap DDR5 interposition setup, supporting DDR5's significantly higher bus speeds and multi-cycle transmission protocol, to future work.

10 Conclusion

In this paper we set out to examine the gap between SGX's stated threat model and those of actual real world deployments. We demonstrated how the security guarantees of scalable SGX can be undermined by hobbyists, constructing a logic analyzer based memory interposition setup for under $1,000. Combining our setup with the recent shift towards server deployments and scalable SGX, which relies on Intel TME's deterministic encryption mechanism, we were able to perform a new ciphertext side channel attack to extract a DCAP attestation key from a Xeon Scalable server in fully trusted status for the first time. Moreover, we investigated a number of case studies, showing how this can breach the confidentiality and integrity guarantees of real world SGX deployments such as PHALA, CRUST, and SECRET, allowing an attacker to recover private data and obtain illegitimate rewards. Finally, we discussed several potential mitigations.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0079; the Alfred P Sloan Research Fellowship; and gifts from Qualcomm, Zama and Zellic.

References

[1] 2025. Crustio | Docker Hub. https://hub.docker.com/u/crustio
[2] 2025. phalanetwork | Docker Hub. https://hub.docker.com/u/phalanetwork
[3] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security*.
[4] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One glitch to rule them all: Fault injection attacks against AMD's Secure Encrypted Virtualization. In *ACM SIGSAC*.
[5] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM CCS*.
[6] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security*.
[7] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *EuroS&P*.
[8] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. 2023. AEX-Notify: Thwarting precise Single-Stepping attacks through interrupt awareness for intel SGX enclaves. In *USENIX Security*.
[9] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. IACR Cryptology ePrint Archive 2016/086.
[10] Crust. 2025. Crust. https://crust.network/
[11] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. *TCHES* (2018).
[12] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. 2024. BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments. In *IEEE S&P*.
[13] Oasis Protocol Foundation. 2025. Oasis. https://oasis.net/
[14] Shay Gueron. 2016. A memory encryption engine suitable for general purpose processors. *Cryptology ePrint Archive* (2016).
[15] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–Bringing access-based cache attacks on AES to practice. In *IEEE S&P*.
[16] Integritee. 2021. Integritee Lightpaper. https://www.integritee.network/docs/Integritee_%20Lightpaper_2021.pdf
[17] Integritee. 2024. Integritee Network Docs. https://docs.integritee.network/
[18] Integritee. 2025. Enclave Dashboard for Integritee Network. https://enclaves.integritee.network/?rpc=wss://paseo.api.integritee.network
[19] Intel. 2015. Improving real-time performance by utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.
[20] Intel. 2018. Memory Address Translation DSM Interface. https://cdrdv2-public.intel.com/603354/338177-address-translate-dsm-specification-001.pdf
[21] Intel. 2021. Intel ® Software Guard Extensions (Intel® SGX) for SuperMicro* Servers Enabling Guide. https://cdrdv2-public.intel.com/646654/646654_Intel_SGX_Enabling_Guide_Rev1p0.pdf
[22] Intel. 2021. Intel® Hardware Shield – Intel® Total Memory Encryption. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf.
[23] Intel. 2021. Supporting Intel SGX on multi-socket platforms. https://web.archive.org/web/20220822150148/https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf.
[24] Intel. 2024. SGXDataCenterAttestationPrimitives. https://github.com/intel/SGXDataCenterAttestationPrimitives
[25] Intel. 2025. Intel Software Guard Extensions Attestation Service Utilizing Intel Enhanced Privacy ID. https://www.intel.com/content/www/us/en/developer/archive/tools/sgx-attestation-service-utilizing-epid.html
[26] Intel. 2025. Intel® Software Guard Extensions SSL for SGX SDK 2.25, with OpenSSL 3.0.14. https://github.com/intel/intel-sgx-ssl/releases/tag/3.0_Rev4
[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES. In *IEEE S&P*.
[28] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. 2024. ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In *USENIX Security*.
[29] JEDEC. 2019. Annex L: Serial Presence Detect (SPD) for DDR4 SDRAM Modules. JEDEC Standard 21-C, Section 4.1.2.L-4. https://www.jedec.org/standards-documents/docs/spd412l-4
[30] JEDEC. 2021. DDR4 SDRAM STANDARD. https://www.jedec.org/standards-documents/docs/jesd79-4a
[31] JESD79-4C 2020. *DDR4 SDRAM*. Standard. JEDEC.
[32] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A High-Resolution Side-Channel Attack on Last-Level Cache. In *DAC*.
[33] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P*.
[34] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip attack on hardware enclaves via the memory bus. In *USENIX Security*.
[35] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A systematic look at ciphertext side channels on AMD SEV-SNP. In *IEEE S&P*.
[36] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*.
[37] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*.
[38] Tony Luck. 2018. ACPI/ADXL: Add address translation interface using an ACPI DSM. https://github.com/torvalds/linux/commit/4cf841e398503990df640f7a7c5b2ea56f11c08c

[39] David McGrew, K Igoe, and M Salter. 2011. RFC 6090: Fundamental Elliptic Curve Cryptography Algorithms.
[40] Crust Network. 2021. Storage Merchant. https://github.com/crustio/crust-wiki/blob/f27cab9fcb7ba792903c96907b08014df17c88bb/docs/storage-merchant.md
[41] Crust Network. 2022. Crust Storage 101. https://github.com/crustio/crust-wiki/blob/49eaaee592c553afe4cb6c87ede49bf4ea3c2a5f/docs/build-101.md
[42] Crust Network. 2024. Crust Network System Optimization: Enhancing On-Chain Computation Efficiency and File Spower Processing. https://medium.com/crustnetwork/crust-network-system-optimization-enhancing-on-chain-computation-efficiency-and-file-spower-e49c006cd99c
[43] Crust Network. 2024. Crust Node. https://github.com/crustio/crust-wiki/blob/1cd68f06abee7336b588b252576186f3e7862ee0/docs/build-node.md
[44] Phala Network. 2023. Confidence Level & SGX Function. https://github.com/Phala-Network/phala-docs/blob/ebcd4355ca8a3f8911fa07fc6647f660488cf2c3/compute-providers/basic-info/confidence-level-and-sgx-function.md
[45] Phala Network. 2024. Gemini Tokenomics (Worker Rewards). https://github.com/Phala-Network/phala-docs/blob/3aed43d8faeec26a9c1a610155d609eada2ff84e/compute-providers/basic-info/worker-rewards.md
[46] Phala Network. 2024. Phala Blockchain in Detail. https://github.com/Phala-Network/phala-docs/blob/f64960997c1dbcf912e1d5295f75eeb151add2ba/tech-specs/blockchain/README.md
[47] Phala Network. 2024. phala-blockhain: pal_gramine.rs. https://github.com/Phala-Network/phala-blockchain/blob/c8b71e935999993b2931cbb0739c1426c675f658/standalone/pruntime/src/pal_gramine.rs#L46
[48] Phala Network. 2024. Phat Contracts. https://phala.network/phat-contract
[49] Phala Network. 2024. Run Local Testnet. https://github.com/Phala-Network/phala-docs/blob/3aed43d8faeec26a9c1a610155d609eada2ff84e/references/advanced-topics/run-local-testnet.md
[50] Phala Network. 2024. Secret Key Hierarchy. https://github.com/Phala-Network/phala-docs/blob/f64960997c1dbcf912e1d5295f75eeb151add2ba/tech-specs/blockchain/secret-key-hierarchy.md
[51] Secret Network. 2025. Secret Network. https://scrt.network/
[52] Secret Network. 2025. Secret Network Graypaper. https://scrt.network/graypaper
[53] NIST. 2025. PRE-DRAFT Call for Comments: NIST Launches Development of Cryptographic Accordions. https://csrc.nist.gov/pubs/sp/800/197/a/iprd
[54] OpenSSL. 2021. openssl/crypto/ec/ec_mult.c. https://github.com/openssl/openssl/blob/openssl-3.0.14/crypto/ec/ec_mult.c#L352
[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
[56] Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCan*.
[57] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security*, Thorsten Holz and Stefan Savage (Eds.).
[58] Zheng Leong Chua Peyman Momeni, Setareh Ghorshi. 2024. Multimodal Cryptography Series – Accountable MPC + TEE. https://hackmd.io/\spacefactor\@m{}Fairblock/rkSiU78TR
[59] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CROSSTALK: Speculative Data leaks Across Cores Are Real. In *IEEE S&P*.
[60] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. 2018. Supporting third party attestation for Intel SGX with Intel Data Center Attestation Primitives. https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf. (2018).
[61] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS*.
[62] Aaditya Shidham. 2025. Trusted Execution Environments (TEEs): A primer. https://a16zcrypto.com/posts/article/trusted-execution-environments-tees-primer/
[63] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing*.
[64] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. 2007. AEGIS: A single-chip secure processor. *IEEE Design & Test of Computers* (2007).
[65] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.
[66] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. Rogue In-flight Data Load. In *IEEE S&P*.
[67] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE S&P*.
[68] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. 2021. SoK: SGX.Fail: How Stuff Gets eXposed. In *IEEE S&P*.
[69] Wikipedia. 2025. Intel QuickPath Interconnect. https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect
[70] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. 2022. Hybrid Trust Multi-party Computation with Trusted Execution Environment. In *NDSS*.
[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*.
[72] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.
[73] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2024. HyperTheft: Thieving Model Weights from TEE-Shielded Neural Networks via Ciphertext Side Channels. In *ACM CCS*.
[74] Yuanyuan Yuan, Zhibo Liu, Sen Deng, Yanzuo Chen, Shuai Wang, Yinqian Zhang, and Zhendong Su. 2025. CipherSteal: Stealing Input Data from TEE-Shielded Neural Networks via Ciphertext Side Channels. In *IEEE S&P*.
[75] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *ACM CCS*.
[76] Shunfan (Shelven) Zhou. 2022. Technical Analysis of why Phala will not be affected by the Intel SGX chip vulnerabilities. https://phala.network/posts/technical-analysis-of-why-phala-will-not-be-affected-by-the-intel-sgx-chip-vulnerabilities-e045b0189dc2

A Miscellaneous Mapping Functions

A.1 ChannelAddress

ChannelAddress Bits	Physical Address Bits
$b_{[0..7]}$	$b_{[0..7]}$
$b_{[8..27]}$	$b_{[11..30]}$
b_{28}	$\neg b_{31} \wedge (b_{32} \vee b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{29}	$(b_{31} \vee b_{32} \vee \neg b_{33}) \wedge (\neg b_{32} \vee b_{33}) \wedge (\neg b_{31} \vee b_{33})$
	$\wedge (b_{31} \vee b_{32} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{30}	$(b_{31} \oplus b_{32}) \wedge (b_{31} \wedge b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{31}	$(b_{31} \vee b_{32} \vee b_{33} \vee \neg b_{34}) \wedge (\neg b_{33} \vee b_{34})$
	$\wedge (\neg b_{31} \vee b_{34}) \wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{35} \vee b_{36})$
b_{32}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee \neg b_{35}) \wedge (\neg b_{34} \vee b_{35})$
	$\wedge (\neg b_{33} \vee b_{35}) \wedge (\neg b_{32} \vee b_{35})$
	$\wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{36})$
b_{33}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{35}) \wedge b_{36}$

A.2 RankAddress

RankAddress Bits	Physical Address Bits
$b_{[0..7]}$	$b_{[0..7]}$
$b_{[8..27]}$	$b_{[11..30]}$
b_{28}	$\neg b_{31} \wedge (b_{32} \vee b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{29}	$(b_{31} \vee b_{32} \vee \neg b_{33}) \wedge (\neg b_{32} \vee b_{33}) \wedge (\neg b_{31} \vee b_{33})$
	$\wedge (b_{31} \vee b_{32} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{30}	$(b_{31} \oplus b_{32}) \wedge (b_{31} \wedge b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{31}	$(b_{31} \vee b_{32} \vee b_{33} \vee \neg b_{34}) \wedge (\neg b_{33} \vee b_{34})$
	$\wedge (\neg b_{31} \vee b_{34}) \wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{35} \vee b_{36})$
b_{32}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee \neg b_{35}) \wedge (\neg b_{34} \vee b_{35})$
	$\wedge (\neg b_{33} \vee b_{35}) \wedge (\neg b_{32} \vee b_{35})$
	$\wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{36})$
b_{33}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{35}) \wedge b_{36}$

A.3 Column

Column Bits	Physical Address Bits
$b_{[0..2]}$	$b_{[3..5]}$
b_3	b_{16}
b_4	b_7
$b_{[5..9]}$	$b_{[11..15]}$

A.4 BankGroup

BankGroup Bits	Physical Address Bits
b_0	$b_6 \oplus b_{23}$
b_1	$b_{20} \oplus b_{24}$

A.5 MemoryControllerId

MemoryControllerId Bits	Physical Address Bits
b_0	$b_8 \oplus b_{14} \oplus b_{22}$
b_1	$b_9 \oplus b_{15} \oplus b_{23}$

A.6 ChannelId

ChannelId Bits	Physical Address Bits
b_0	$b_{10} \oplus b_{16} \oplus b_{24}$

A.7 Row

Row Bits	Physical Address Bits
$b_{[0..2]}$	$b_{[17..19]}$
$b_{[3..10]}$	$b_{[23..30]}$
b_{11}	$\neg b_{31} \wedge (b_{32} \vee b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{12}	$(b_{31} \vee b_{32} \vee \neg b_{33}) \wedge (\neg b_{32} \vee b_{33}) \wedge (\neg b_{31} \vee b_{33})$ $\wedge (b_{31} \vee b_{32} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{13}	$(b_{31} \oplus b_{32}) \wedge (b_{31} \wedge b_{33} \vee b_{34} \vee b_{35} \vee b_{36})$
b_{14}	$(b_{31} \vee b_{32} \vee b_{33} \vee \neg b_{34}) \wedge (\neg b_{33} \vee b_{34})$ $\wedge (\neg b_{31} \vee b_{34}) \wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{35} \vee b_{36})$
b_{15}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee \neg b_{35}) \wedge (\neg b_{34} \vee b_{35})$ $\wedge (\neg b_{33} \vee b_{35}) \wedge (\neg b_{32} \vee b_{35})$ $\wedge (b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{36})$
b_{16}	$(b_{31} \vee b_{32} \vee b_{33} \vee b_{34} \vee b_{35}) \wedge b_{36}$

A.8 Bank

Bank Bits	Physical Address Bits
b_0	$b_{21} \oplus b_{25}$
b_1	$b_{22} \oplus b_{26}$

B Examining INTEGRITEE

B.1 INTEGRITEE Overview

INTEGRITEE is a project in the Polkadot community aiming to address the issues of scalability, interoperability, and confidentiality that typically affect blockchains [16]. To help solve the aforementioned problems, INTEGRITEE looks towards TEEs, and SGX in particular. INTEGRITEE users can benefit from a number of SGX-backed offerings including: sidechains; trusted off-chain computation; and oracles, which provide on-chain-access to off-chain data.

Uniquely, they also offer attestation infrastructure: Attesteer is a service the INTEGRITEE network provides for remote attestation. The network maintains a registry of properly attested enclaves [18], along with their DCAP attestation status, which obviates the need to interact with Intel directly. When an enclave registers on INTEGRITEE, it provides a quote as part of the standard remote attestation process. Assuming it verifies, this quote is then saved by the network. When users look to perform some off-chain operations, they can find a pre-populated list of enclaves that they could use immediately. Without Attesteer, the enclave would need to perform remote attestation with its client to prove enclavehood. But if the client trusts this registry, then there is no need.

Components. The INTEGRITEE blockchain is a member of the wider Polkadot ecosystem. These members, dubbed "parachains", all share the same base "relay" blockchain (Polkadot) for consensus. INTEGRITEE has three principal components: an enclave worker component, the Polkadot relay chain, and the INTEGRITEE parachain itself. The parachain is a custom blockchain architected by INTEGRITEE which implements its SGX feature set and hosts its native TEER tokens. The relay chain is the overarching blockchain maintained by Polkadot itself that handles inter-chain communication and maintains overall consensus and shared security. The enclave worker component is responsible for handling INTEGRITEE's various features. This worker can be configured to be a sidechain validator, a trusted off-chain worker, or an oracle. The sidechain validators and off-chain workers both increase the maximum throughput of the network in addition to providing support for private transactions and execution. Their oracle, named TEEracle, is a blockchain oracle framework that allows for on-chain contracts to access off-chain data. Typically, traditional smart contracts are limited to accessing only the data on the blockchain, but TEEracle facilitates on-chain access to real-world data through SGX.

INTEGRITEE Security Guarantees. The aforementioned use cases all require some guarantee provided by SGX. As an intermediary of data, TEEracle must not be able to lie and must faithfully represent real-world data. INTEGRITEE uses TEEracle directly for fetching TEER exchange rates, for example. A malicious oracle operator could falsify the exchange rate to manipulate the market for their own gain. Ordinarily, blockchain consensus is used to enforce integrity of execution, but here SGX fills that role instead. But replication of work is expensive and inefficient, so the sidechain validator and trusted off-chain worker both substitute consensus with SGX for its guarantees of correct execution. Without it, validators could forge transactions and workers could claim credit for work not done (like with Crust). Additionally, SGX provides privacy which allows for confidential transactions and computation.

Enclave Registration. When a node operator wishes to run a worker node, they must first deposit some TEER in the worker enclave's account. This deposit allows the node to interact with the network. The worker enclave then generates an asymmetric key pair to facilitate communication and to serve as an identifier. Similar to prior projects discussed, since this key is generated inside the enclave, communication encrypted with this public key is ostensibly readable by the enclave itself only. During startup, the node will generate an enclave report including its public key in the report data. (Thus attesting that this public key *is* the enclave.) The enclave generates a quote for this report and submits it to the network, where it is then verified and stored. Since INTEGRITEE maintains a registry of attested enclaves and their respective public keys, users and applications communicating with enclaves will not need to verify attestations themselves. They can simply check the registry for the node's status.

B.2 Extracting Data From INTEGRITEE

Setup. We use the same hardware setup as in all prior attacks. We first set up a node on the INTEGRITEE testnet (with both the parachain and relay chain components), following the INTEGRITEE documentation [17], and we allow them both to synchronize. Note that this setup is strictly unnecessary, as the quote verification happens on-chain. We could choose to submit our quote to any node, but we ran our own for easy log access during testing. We compile and run a worker configured as a sidechain validator.

Modifying the INTEGRITEE Enclave. We patch the worker enclave to return a custom report. Rather than its own `mrsigner`, we choose that of an unmodified production enclave. We chose a random `mrenclave` value, clear the debug flag and fill the rest of the report correctly. With the attestation key extracted in Section 6.3, we create our own quote and forge the signature. Then execution resumes as usual. The enclave sends the quote to the network where it verifies successfully and is placed on the registry. At this point it is as if we had a proper production enclave on the network.

Reading Data From INTEGRITEE. For ethical considerations, we created a local testnet to demonstrate the effects of our compromised enclave. Simply by using GDB (as our "production" enclave is really a debug one in disguise), we are able to access private transactions for INTEGRITEE's `incognito` accounts (and those of other parachains when shielded to the INTEGRITEE network); and more generally, we can access any off-chain computations submitted to our node. We can also modify oracles to return falsified information of our choosing, thus causing them to provide any view of real world data to the chain that we choose.