TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition

Jalen Chuang
Georgia Tech
jchuang@gatech.edu

Alex Seto
Purdue University
aseto@purdue.edu

Nicolas Berrios
Georgia Tech
berrios@gatech.edu

Stephan van Schaik
van Schaik, LLC
stephan@synkhronix.com

Christina Garman
Purdue University
clg@cs.purdue.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

***Abstract*—Trusted execution environments (TEEs) aim to offer strong privacy and integrity guarantees even in the presence of root level attackers capable of arbitrarily modifying the system's software. Recently however, there has been a pivotal shift in TEE deployment, moving TEEs from enclaves running on PC-oriented hardware to confidential virtual machines executing on server-grade CPUs. Under the hood, this change has also resulted in significant modifications to the underlying memory encryption engine, removing integrity guarantees and protections against replay attacks. While Intel's and AMD's change in TEE implementation is clearly significant and substantial, most TEE deployments appear to fail to acknowledge the difference in security guarantees, assuming a stronger security model than truly afforded by the implementation.**

Thus, in this work we investigate the true protection offered by Intel's and AMD's newest TEE offerings against entry-level physical side-channel attacks. We show that, contrary to popular belief, bus interposition attacks on DDR5 server memory can be constructed cheaply by hobbyists, using parts easily obtained on e-commerce websites. Next, combining our ability to monitor DDR5 bus transactions with deterministic memory encryption used by Intel's SGX and TDX as well as AMD's SEV-SNP, we are able to extract secret key material (such as attestation keys in some cases) from machines in fully trusted status. Finally, we demonstrate the implications of our attacks on multiple real world TEE deployments.

1. Introduction

Starting from humble origins as secure enclaves running on Intel client machines, Trusted Execution Environments (TEEs) have undergone a revolution, and are now present on nearly all server-grade hardware deployments. Offering strong confidentiality and integrity guarantees even against root-level adversaries, users can now run enclaves or even entire virtual machines (VMs) on remote third party hardware, while being assured that their data and code is being offered strong hardware-backed security guarantees via a robust attestation process. Building on these foundations TEEs are now even supported by GPUs, aiming to allow confidential AI applications.

However, the deployment of TEEs on server hardware by vendors such as Intel, AMD and NVIDIA, coupled with the sun-setting of SGX on client machines, has resulted in significant changes in the security guarantees offered by server-based TEE implementations compared to Intel's original (now deprecated) client-oriented SGX implementation. More specifically, unlike client-based SGX, server TEEs use deterministic AES-XTS for memory encryption, and do not offer Merkle tree-based integrity or replay protections in the case of hardware attacks on the machine's memory bus. This degraded protection in turn comes with usability and performance benefits, allowing server TEEs to support large regions of protected memory (up to terabytes in some cases), thus making VM support feasible while reducing latency and bandwidth for memory accesses [1, 2].

While prior work [3, 4, 5] had demonstrated the risk of deterministic encryption on server hardware via cheap home made DDR4 bus interposition setups, the latest generation of TEE hardware (e.g., Intel TDX, AMD SEV-SNP with ciphertext hiding, or NVIDIA's Confidential Compute) run exclusively on systems equipped with DDR5 memory and thus remain out of scope. Thus, in this paper we investigate the true feasibility of mounting memory bus attacks against modern DDR5-based TEE deployments. More specifically, in this paper we ask the following questions:

How can adversaries mount memory interposition attacks on DDR5-based TEE implementations? What are the security implications of these attacks and can they be mounted by hobbyist-level attackers with limited budgets?

1.1. Our Contributions

In this paper we demonstrate for the first time a memory bus interposition attack on server grade DDR5 memory. Moreover, our attack can be done in under $1000 by computer hobbyists using equipment readily available on the secondhand market. Next, we use our interposition setup to observe memory bus transactions on Intel Xeon

Scalable 5th Generation and AMD Zen 5 systems, thereby (re-)enabling ciphertext-based attacks on these machines. More specifically, we break SGX and TDX security guarantees by extracting attestation keys from machines in a fully trusted `UpToDate` attestation status. With extracted attestation keys in hand we demonstrate the effectiveness of our attacks by breaking the security guarantees of realistic deployments, including running GPU workloads outside TEE protections while passing attestation for NVIDIA Confidential Computing. Finally, we attack the newest version of AMD SEV-SNP present in Zen 5 EPYC processors, which includes ciphertext hiding features for preventing prior software-based ciphertext attacks [6]. To the best of our knowledge, these are the first ciphertext attacks on DDR5-based systems, including SGX, TDX and SEV-SNP. Our results impact nearly all server-based TEE implementations with commercially-available hardware at the time of writing.

Interposing DDR5 Memory. We begin our attack by extending prior bus interposition attacks [3] on DDR4 memory running at 1333 MT/s to DDR5 memory. While we empirically observe that server DDR5 RDIMMs cannot operate below 3200 MT/s, we do notice that unlike their DDR4 counterparts, DDR5 DIMMs contain two independent channels on a single memory module. Thus, while DDR4 interposition requires about 136 logic analyzer inputs per memory channel, DDR5 setups only require 68, allowing us to use the remaining hardware capacity for increasing acquisition speed. Overall, this allows us to design a logic analyzer probe that can be constructed by computer hobbyists operating on a sub-$500 budget, with the entire DDR5 bus snooping setup costing under $1000. 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.

Controlling Enclave Execution. With our ability to observe DDR5 memory transactions, our next step is to obtain a sufficient level of control over SGX and TDX execution. First, while Intel requires at least 8 DIMMs (e.g., 16 channels for DDR5 memory) for TDX and SGX activation, our setup is only capable of observing a single channel. Thus, we first reverse engineer the mapping between physical address and DIMM locations on 5th generation Intel Scalable Xeon systems. Next, we modify the OS kernel to place virtual addresses of interest on the DIMM and channel connected to our logic analyzer, allowing us to observe memory transactions to these addresses. Finally, we overcome the system's caching and force DRAM traffic via flushing, while using a control channel attack to precisely control TEE execution, synchronizing it to our logic analyzer triggering logic.

Attacking TDX and SGX Attestation. With our ability to control TEE execution in hand, we notice that both TDX and SGX attestation rely on a single source of trust, namely an Intel-signed Provisioning Certification Enclave (PCE), running under the machine's SGX protections. Combining this observation with Intel's use of deterministic AES-XTS encryption, we are able to use our memory interposition setup to break the confidentiality of the PCE enclave, extracting its Provisioning Certification Key (PCK). With a PCK from a machine in a fully trusted `UpToDate` status in hand, we are able to to sign our own attestation keys which do not originate from an Intel-signed QE. This in turn allows us to sign arbitrary SGX or TDX reports without any TEE protections, thereby completely breaking SGX and TDX security guarantees. To the best of our knowledge, this is the first end-to-end provisioning certification key extraction attack on a TDX system in a fully trusted `UpToDate` status.

Breaking TEE Confidentiality and Integrity. Having utilized our attack to subvert attestation, we now proceed to examine the confidentiality and integrity guarantees of real-world applications relying on SGX and TDX. We notice a clear gap in threat model. More specifically, while server TEEs are designed to run in data centers (and thus rely on the operator for physical security) [1, 2], many applications use TEEs with the explicit goal to avoid operator trust shifting it instead to the underlying hardware.

We first investigate BUILDERNET, a part of the Ethereum blockchain ecosystem that uses TDX to provide integrity, confidentiality, and trustworthiness for block builders and users, processing millions of dollars in value each month. We break these guarantees, demonstrating how a malicious operator could both extract configuration secrets and gain the ability to frontrun (and profit) without being detected.

NVIDIA Confidential Computing. Using our subverted TDX attestation process, we investigate Phala Network's DSTACK, an SDK for deploying docker containers into TDX. At a high level, DSTACK aims to extend the protections of TDX to docker containers with minimal code changes, easing deployment. We are able to break TDX's guarantees for all applications relying on DSTACK. Notably, we show that we are able to use an NVIDIA Confidential Computing attestation from a different computer as if it were our own. We then break two applications relying on DSTACK, a tool for hosting Jupyter notebooks and an LLM frontend that attests to running in TDX, allowing us to pass NVIDIA Confidential Computing attestation while running the workload without any TEE protections.

Targeted Data Extraction Beyond Attestation. Having explored the implications of PCK extraction, we now proceed to demonstrate that even securing the system's PCE and QE is insufficient for mitigating bus interposition attacks. To that aim, we examine SECRET, one of the first privacy-preserving smart contract systems and wide-spread production deployments of SGX, featuring a $57M USD

market cap. To ensure the confidentiality of contract data and execution, SECRET relies on a single master key which is shared among all validator nodes and stored in an enclave. While it would be possible to utilize our already-extracted PCK to falsely attest an attacker, we instead elect to use our memory interposition setup to directly extract a node's SECRET-specific ECDH private key from the SECRET enclave. This allows us to obtain the network's master key directly, thereby completely breaching SECRET's confidentiality guarantees, without attacking the system's quoting enclave or needing the machine's provisioning key.

Attacking AMD SEV with Ciphertext Hiding. Going beyond Intel's SGX and TDX, we also investigate AMD's SEV-SNP implementation. Here, in an effort to mitigate prior software-based ciphertext attacks [7], SEV-SNP VMs running on EPYC processors based on the Zen 5 microarchitecture support Ciphertext Hiding [6], which aims to hide the VM's encrypted state from malicious hypervisors. However, as this does not protect against attackers with physical bus access, we re-enable ciphertext attacks on these machines by demonstrating the extraction of signing keys from OpenSSL's constant-time ECDSA implementation.

Summary of Contributions. We contribute the following:

- We build a low budget DDR5 interposition setup capable of observing DRAM bus transactions (Section 4).
- We demonstrate how to achieve control over TEE execution, allowing us to observe virtual addresses of interest using our memory interposer (Section 5).
- We breach TDX's and SGX's security guarantees by extracting a Provisioning Certification Key (PCK) from a Xeon server in a fully trusted status (Section 6).
- Using our PCK we breach BUILDERNET's use of TDX, violating its confidentiality and integrity (Section 7).
- We break TDX and GPU attestation, running CPU and GPU workloads outside of TEE protections (Section 8).
- We demonstrate that even protecting the system's PCE and QE is not sufficient by breaching the SECRET network through a targeted ECDH key extraction (Section 9).
- We extract signing keys from OpenSSL's ECDSA implementation running inside a VM protected by AMD SEV-SNP with ciphertext hiding enabled (Section 10).

1.2. Responsible Disclosure and Ethics

Following the practice of coordinated vulnerability disclosure, we shared our findings with Intel in April 2025, AMD in August 2025, NVIDIA in June 2025, as well as with the security teams of the affected deployments over the May to July 2025 timeframe. Intel, AMD, NVIDIA, and all affected deployments have acknowledged our findings, and are considering releasing statements simultaneously with the public release of this paper. The affected deployments are currently working on mitigations as well as adapting their threat models in response to our attack.

All attacks discussed were performed on our own computers either using local testnet setups or the official project testnets designated for such activities. Any data involved was our own, created specifically for these purposes.

2. Background and Related Work

2.1. Intel's Trusted Execution Environments

In this section we provide background regarding Intel's TEE implementations, namely SGX and TDX.

Intel Software Guard eXtensions (SGX). Intel's Software Guard eXtensions (SGX) introduces `x86_64` instructions to create trusted execution environments, called enclaves, that support secure code execution. Intel SGX aims to shield the code and data within enclaves from both inspection and modification, even in the presence of root-level adversaries with full control over the system's software stack. Furthermore, SGX employs an attestation process, allowing remote parties to ensure that enclaves are running on genuine trustworthy Intel hardware. To identify enclaves, SGX maintains a measurement of the enclave's initial state (`MRENCLAVE`) and the enclave developer's identity (`MRSIGNER`).

At boot time, the CPU reserves a region of memory known as Processor Reserved Memory for exclusive use by Intel SGX, isolating it from all untrusted code. The reserved pages are initially unallocated and stored in the Enclave Page Cache (EPC), a list of free pages maintained by the untrusted kernel. When an enclave is loaded, the kernel allocates pages from the EPC, and initializes them with code and data. Upon initialization, the kernel marks the pages as ready for execution. Once initialized they cannot be read or written by the kernel, even in the event of swapping, as the kernel can only read or write pages upon CPU encryption [8].

Intel Trusted Domain Extensions (TDX). To widen adoption of confidential computing, Intel released Trusted Domain Extensions (TDX) in 2021, which extends confidentiality and authenticity guarantees to complete virtual machines (called trust domains, or TDs) [9]. Like SGX protects enclave memory, TDX protects the virtual machine state from the hypervisor. However, TDs do not require the engineering effort of SGX enclaves, as they behave like normal virtual machines with their own secure guest physical address space. To support loading confidential VMs, Intel added *Secure Arbitration Mode* (SEAM) to the CPU to load trusted Intel-signed virtual machine management software known as the TDX module [10]. Also similarly to SGX, TDX provides measurements for identifying a running TD. The `MRTD` measures the TD's initial state, and `RTMR` measurements measure runtime state of the TD.

Trusted Compute Base (TCB). For SGX to operate securely, the SGX TCB consists of the trusted components that must operate correctly, and may not be compromised or malicious. Among these are the CPU itself, its microcode, and the CPU's root keys. In terms of software, SGX relies on the correctness of Intel's Provisioning and Quoting enclaves, which handle the machine's Provisioning Certificate Key (PCK) and attestation keys. Other system components, such as the BIOS, machine's DRAM or even the DIMM modules themselves, remain untrusted, thus they are not part of the TCB. The TDX TCB includes the SGX TCB in addition to Intel's SEAM loader and TDX Module software.

Memory Encryption. To harden against physical memory attacks, such as cold boot attacks, Intel SGX on Xeon Scalable platforms employs a memory encryption engine called Intel Total Memory Encryption (TME) [11]. More specifically, the DRAM controllers implement TME, encrypting the entire address space using AES-XTS with a key determined at boot time. Rather than writing plaintexts, Intel TME writes ciphertexts to memory at a 128-bit granularity. In addition, TME's AES-XTS implementation includes a tweak function to incorporate the physical address to ensure that data written to different physical addresses produces different ciphertexts. TDX further extends the memory protection model of SGX to multiple tenants by giving TDs separate unique ephemeral encryption keys for AES-XTS encryption. Furthermore, integrity protection utilizing SHA-3 based MACs is optionally available for TD memory [12].

Attestation. A key feature of both SGX and TDX is remote attestation. This allows a TEE to prove to a remote verifying party that it is indeed running on trustworthy and genuine Intel hardware, guaranteeing its confidentiality and integrity. Subsequently, this allows the remote party to provision the TEE with secrets, with the assurance that these secrets can never leave. In this paper we focus in particular on Intel Data Center Attestation Primitives (DCAP), the remote attestation process used by Intel's Xeon Scalable platforms, which is used by both SGX and TDX. See Section 6.2 for a more in-depth overview and Appendix Bfor extended details.

2.2. Attacks on TEEs

SGX. While micro-architectural data sampling attacks (MDS) [13, 14, 15] exfiltrate data from enclaves, Foreshadow, CacheOut and AEPICLeak [16, 17, 18] extract SGX attestation keys. Crosstalk [19] shows how to extract ECDSA nonces and recover ECDSA keys from an SGX enclave using IPP crypto's ECDSA implementation. Furthermore, SGX.Fail [20] provides an overview of various SGX attackers and how these affect real-world applications.

TDX. While TDX hardens against single-stepping attacks, TDXdown [21] demonstrates that this prevention mode can be sidestepped to re-enable single-stepping. Despite TD state being isolated from the hypervisor, Heckler [22] leverages the hypervisor's control over interrupts to modify TD registers and control flow, demonstrating authentication bypasses in OpenSSL and sudo executing within a TD.

AMD SEV. Like with TDX, Heckler [22] demonstrates the same malicious interrupts can achieve similar register modification and control flow primitives against SEV. WeSee [23] further shows how the AMD-specific `#VC` exception can be used to skip instructions and modify the `rax` register, building an arbitrary VM memory read primitive.

AMD SEV uses a similar 128-bit AES-XEX memory encryption, but unlike SGX does not prevent the hypervisor from reading enclave ciphertexts through memory accesses. Cipherleaks [24] exploits this by observing ciphertexts to the corresponding VM Save Area (VMSA) to break OpenSSL's constant-time ECDSA and RSA implementations in AMD SEV-SNP. Next, rather than just the VMSA, [25] demonstrates that this applies to *all* memory pages. Finally, CipherSteal [26] and HyperTheft [27] exploit ciphertext side channels to recover input data and weights from TEE-shielded neural networks respectively.

Hardware Attacks. DRAMA shows that attackers can infer whether the victim accessed the same row by observing DRAM access latency [28]. Targeting Intel DCAP's predecessor, Membuster [29] uses a professional $170,000 memory interposition setup to extract fine-grained memory access patterns of enclaves. However, the cryptographic hardening prevents Membuster from fully breaching SGX and recovering the attestation key. WireTap and BatteringRAM [3, 4] recently extended this result to DDR4-based servers running SGX and SEV, exploiting deterministic encryption in order to break TEE security guarantees. Next, Buhren et al. [30] glitch the voltage of the AMD-SP to execute custom SEV firmware, allowing the decryption of VM memory as well as extraction of endorsement keys. Finally, VoltPillager [31] breaches the confidentiality and integrity of SGX enclaves by controlling the CPU core voltage to inject faults.

2.3. The Memory System

Computer memory is organized in a hierarchy, with caches providing smaller low-latency storage, to dynamic random access memory (DRAM) chips providing large amounts of high-latency storage. We now discuss the structure of caches, physical memory, and memory management.

Cache Organization. As most programs access the same memory locations repeatedly, CPUs cache memory data to decrease latency. Memory is cached in units of cache lines. These cache lines are stored in multiple buffers increasingly further from the CPU in order of the per-core Level 1 (L1) and Level 2 (L2) cache, and system-wide Last Level Cache (LLC). Cache lines are stored until the cache fills and must evict a cache line. When a program tries to access memory for reading or writing, the processor first checks if the cache line is present in any cache and does not issue a command to the underlying DRAM unless needed. In modern Intel desktop processors, cache lines are 64 bytes wide [10].

Memory Organization. As of DDR5, a single DIMM contains separate two channels, each of which also contain multiple DRAM chips. DRAM chips are first identified by their rank. Ranks are further broken up into bank groups and individual banks. Finally, the row and column address determine the placement of the memory within the bank.

DDR5 Bus Overview. The DDR5 specification for DIMM was released in 2020. All CPUs supporting Intel TDX utilize server-grade registered DDR5 memory (e.g., RDIMMs) as of the time of writing. A registered DDR5 DIMM contains 288 pins, split into two independent channels, which operate separately and concurrently. Each channel has 40 pins for data, 8 of which are allocated to error correction code (ECC) bits. 7 pins in each channel are also allocated for command and address (CA) usages, but the interpretation

of each pin depends on the command. DDR5 CA and data are furthermore encoded as a falling and rising edge pair, meaning each cycle transmits 80 data bits and 14 CA bits.

To read memory from a DIMM, the memory controller first determines which channel to read from, then issues a two-cycle row activation command. The activation command encodes a 17-bit row address, 3-bit chip ID (rank), and 5 bits for the bank group and address on the CA pins. After the activation delay, a two-cycle read command can be sent containing the same chip ID, bank group, and address, plus a 9-bit column address all encoded in CA pins. After the read latency, the DIMM returns the data as a 16-burst of 32 data pins, totaling to 64 bytes per transaction.

Physical and Virtual Addressing. The physical address is an integer that uniquely determines the rank, bank, row, and column where memory is located. The CPU's memory controller is responsible for decoding a physical address to its individual DRAM components.

To isolate multiple applications running on the same physical memory, modern CPUs and operating systems support virtual addressing. When a program requests memory, the requested virtual address is translated to a physical address, which determines the actual location of the memory. Similarly, TDs support guest-physical addresses, which appear like a physical address to the guest operating systems and can be mapped to guest-virtual addresses, but go through a second translation to host-physical addresses [9].

3. Threat Model and Target Setup

We assume an attacker with physical and root-level access to the target machine. For physical access, we assume adversarial capabilities of an electronics hobbyist, capable of simple electrical assembly operations (i.e., soldering) as well as installing components into the target machine. However, we note that our attacks do not require more advanced capabilities such as PCB circuit editing, or chip-level inspections using electron microscopes.

For software access, we assume a root-level adversary, capable of performing arbitrary configurations to the target. This includes modifying settings, installing a custom kernel and drivers, as well as adversarially manipulating the target's userspace setup and memory mappings.

Target Machine. We targeted a server with an Intel Xeon Silver 4509Y processor installed on a Supermicro X13SEI-F motherboard, running BIOS version 2.5a with CPU microcode version 0x2b000639. The machine has eight 16 GB registered DIMMs (RDIMMs) of DDR5 memory, capable of 4800 MT/s. For software, our target machine runs a custom Linux kernel based on version 6.8.2 with Intel and Canonical's TDX patches for Ubuntu 24.04 [32], and uses Intel SGX and TDX libraries version 2.25 with a DCAP library version 1.22. Querying Intel's PCS service resulted in our machine obtaining an `UpToDate` status, meaning Intel considers our machine to be in a fully trusted status, not susceptible to any known TDX or SGX vulnerabilities.

4. Observing Memory Bus Traffic

4.1. Interposer Construction

To observe the DDR5 memory bus, we first need to construct a custom interposition probe. While prior work [3, 4] demonstrates similar setups for DDR4 systems, in this section we extend these to DDR5-based devices.

Step 1: Slowing Down Bus Speeds. The cost of acquisition equipment is highly dependent on the speeds of the signals being interposed. Thus, our first step is to slow down the speed of the system's memory bus, allowing us to simplify our interposition setup. To that aim, while our DIMMs can support a speed of 4800 MT/s (i.e., 2.4 GHz clock), we set the memory speed in our system BIOS to the lowest supported level of 3200 MT/s (1.6 GHz clock). We were also able to lower the memory speed of a single DDR5 RDIMM to 3200 MT/s via modifying its SPD data, which automatically drops all other DIMMs to this lower speed upon its insertion. Finally, we notice that both SGX and TDX consider the machine's memory speed to be outside the TCB, meaning this setting does not affect the machine's `UpToDate` attestation status.

Step 2: Obtaining Access to Memory Bus Traces. With the machine's bus speed set to its lowest supported setting, our next step is to obtain a convenient physical access to traces of the machine's memory bus. To that aim, we use a DDR5 RDIMM riser ($12), which is a simple PCB containing a female and male DIMM connector, designed to act as a pass through between the DIMM and motherboard. See Figure 2 (left). Most importantly, the riser board allows for easy access to the traces of the machine's memory bus, avoiding the need for motherboard or DIMM modifications.

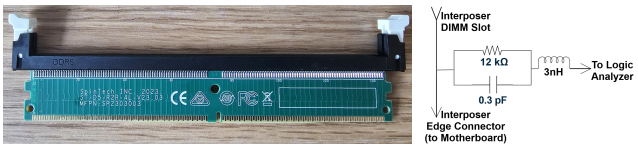


Figure 2: DDR5 RDIMM riser board and probe isolation network schematic

Step 3: A Probe Isolation Network. With the machine's memory bus traces accessible, it is tempting to connect these directly to the logic analyzer inputs via wires. However, we empirically observed that this places significant electrical loads on the CPU's memory circuitry, preventing the machine from booting due to RAM recognition errors. To reduce the electrical load on the machine's RAM circuitry we disassemble and reference a Keysight SoftTouch probe, intended for high frequency buses. We note that these probes use a three component isolation network, consisting of a resistor, capacitor and inductor. See Figure 2 (right).

Step 4: Locating Interposer Parts. While high frequency components are typically expensive, a particularly useful source of parts is Keysight (formerly known as Agilent) N4252A Transition Probe Adapters, which are available on a second hand marketplace for around $40 at the time

of purchase. Inspecting the N4252A's PCB, it appears to contain 102 channels, all outfitted with the probe isolation network from Figure 2 (right). While we were unable to find a datasheet for the N4252A, marking on our unit suggests it was used for the QuickPath Interconnect protocol, which was used by Intel between 2008 and 2017 [33, 34].

Step 5: Interposer Construction. Thus, to build our DDR5 RDIMM interposer, we heated up the N4252A's PCB with an air gun, and manually collected the stacked pairs of capacitors and resistors using tweezers. Next, we used a soldering iron to manually place the capacitor and resistor pairs on traces corresponding to Channel A on the DDR5 RDIMM slot protector. This resulted in us successfully isolating the logic analyzer from the target machine, allowing it to reliably boot without memory issues. See Figure 3.

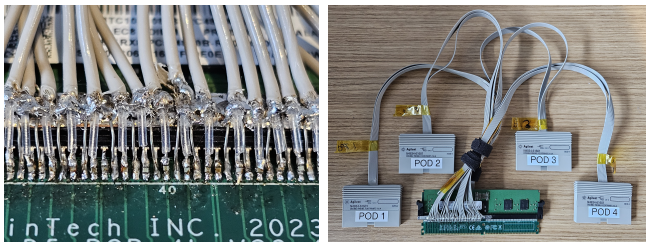


Figure 3: (left) Zoomed-in view on the probe isolation networks. (right) DDR5 RDIMM interposer and logic analyzer connecting pods.

Next, to connect the other side of the probe isolation network to the logic analyzer, we used a Keysight N4834 SoftTouch probe ($22, secondhand) cutting off its bottom SoftTouch connector. We then soldered the wires coming from the probe's four 90-pin pods (which also contain the inductor) directly to the capacitor-resistor pairs placed on the slot protector's PCB. Overall, this resulted in a reliable memory interposition setup requiring an amount of skill akin to a computer technician, without adverse effects on the target machine. Moreover, as we used standard Keysight 90-pin connectors, it is possible to use our probe with a large variety of Keysight equipment.

Step 6: Logic Analyzer Setup. Having constructed our DDR5 RDIMM interposition probe, we now describe our choice of Keysight logic analyzer. Recalling our ability to slow the system's memory clock to 3200 MT/s (1.6 GHz clock), we used two Agilent 16962A acquisition cards capable of acquisition of 1.6 GHz signals and featuring 64MB of DDR2 RAM ($110 each, secondhand). We then placed the cards inside a Keysight 16902A logic analyzer chassis ($550, secondhand), upgrading its internal computer from a Pentium III-based CPU to a Lenovo ThinkCentre M920Q MFF i5-8500T ($150, secondhand). Finally, we installed Agilent's Logic and Protocol Analyzer application version 5.9, the latest supporting this hardware. See Figure 1.

4.2. Observing Bus Transactions

Now that we have built an interposer and connected it to a logic analyzer, we are able to observe bus transactions. See Figure 4. First, a row activation is sent containing the row address, bank address, and bank group. Some time later, a read is issued with the same bank address and group, along with the column address. Finally, after the read delay, the read data of a full burst is observed in the data lines.

Figure 4: (top) A read command on the logic analyzer in the Waveform view. (bottom) A read command on the logic analyzer in the Listing view.

5. Controlling Enclave Memory and Execution

To enable SGX and TDX on scalable Xeon servers, Intel requires the first slot of each memory controller channel to be populated [35]. Overall, this results in our machine having eight 16GB DDR5 RDIMMs, each supporting two independent channels (e.g., 16 channels total). With our interposer only able to observe bus transactions on a single channel of the machine's memory, we now outline our techniques for ensuring that encrypted data present in virtual addresses of interest will be visible using our interposer.

5.1. Obtaining Physical Address Mappings

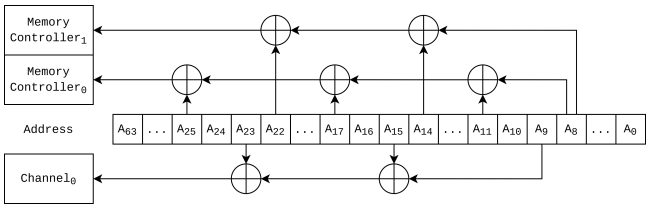
Before forcing the data present in the virtual addresses of interest to be present on the single memory channel observable using our interposer, we must first recover the mappings between the system's physical address space and the corresponding locations on the machine DIMMs. While prior work has relied on row buffer conflicts [28, 36] to recover this information, we find that this side channel is much noisier on DDR5 systems. Beyond noise, it is much harder to induce the row buffer conflicts on our target machine, presumably due to the increasingly complex addressing functions used for the case of a fully populated eight-DIMM system. Instead, we discover that Intel exposes an address decoding interface, and utilize a simple algorithm to recover the actual mapping functions from decoded components.

More specifically, we find that Intel exposes an interface for physical address to component translation known as the Memory Address Translation (ADXL) [37] to the Linux kernel [38, 39]. We then further expose this decoding

interface to userspace via `sysfs`, allowing us to obtain for each physical address its corresponding memory controller ID, channel ID, bank group and address and row address, which determine the physical addresses' DIMM location.

Next, we recover the actual mapping functions from physical addresses to DIMM locations used by our target machine. To that aim, we start with a known valid physical address `addr` and trigger a decoding of this address using the above primitive, obtaining its location on the machine's DIMMs. Then, for each pair of address bits (i_0, i_1) we flip the i_0th and i_1th bits inside `addr` and decode the resulting new addresses, recording the results. Once we obtain which address bits affect what part of the DRAM address, we use Gaussian elimination to determine which bits form a linear function (in other words, whether the output bit is the result of XORing all input bits together).

If the resulting system of linear equations is found to be inconsistent for an output bit, we directly construct a truth table for this output bit by decoding addresses corresponding to all possible combinations of input address bits. Then, we minimize this truth table, obtaining logical functions for the remaining non-linear output bits.



$$\text{Row}_{11} \leftarrow (A_{31} \vee \neg A_{32}) \wedge (\neg A_{31} \vee A_{32}) \wedge (A_{33} \vee A_{34} \vee A_{35} \vee A_{36} \vee A_{37}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{33} \vee \neg A_{37}) \wedge (A_{31} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{36})$$

Figure 5: Example recovered mapping functions.

5.2. Controlling SGX Page Allocation

Having recovered the machine's mapping from physical addresses to DIMM locations, our next step is to ensure that virtual addresses of interest from SGX's enclave page cache (EPC) are visible to our interposer. As SGX relies on the OS kernel for physical memory allocation, we must modify the kernel to allocate pages to the specific DIMM channels we can observe. To that aim, we modify the kernel's SGX driver to accept a virtual and physical address pair as a parameter to be stored in global kernel memory. Each pair is a desired mapping: a virtual address to be pinned to the corresponding physical address. Next, when an enclave is to be initialized and the kernel attempts to allocate memory, we check if the virtual address corresponds to a pinned virtual address. If the virtual address does not correspond to a pinned address, the modified SGX driver returns the first page that is not a target physical address. However, in case the virtual address is a pinned address, we search through desired mappings passed to the kernel earlier, and ensure that the virtual address is allocated to the mapping's corresponding physical address. Overall, this allows us to precisely place virtual addresses of interest on desired physical addresses, which in turn correspond to DIMM locations observable via our interposer.

5.3. Controlling SGX Enclave Execution

Having obtained the ability to pin virtual addresses of interest to our desired DIMM locations, our next step is to control the enclave's execution in a way which allows us to synchronize data collection with our logic analyzer. To that aim, we use two primitives, which we now describe.

Enclave Control Channel. To arm the trigger on our logic analyzer in preparation for acquisition, we first need the capability to suspend enclave execution at specific points of interest. As SGX does not allow even root-level users to set breakpoints inside an enclave memory in production status, we instead resort to controlled-channel attacks [40] to control the execution flow of production enclaves.

More specifically, we start by `ptrace`-ing our target program and wait until the enclave pages are initialized. We then modify the permission of the code page where we wish to interrupt execution, disallowing reads and writes. When the target enclave attempts to execute this code page, our tracer will be notified and can restore the page's permissions. Simultaneously, we program our logic analyzer to acquire data for an address of interest as we know what code path the enclave is currently executing. This allows us to trace through the execution of an enclave, interrupting whenever we wish to utilize our interposer. We note that this is particularly effective for interrupting function call boundaries, as the enclave often jumps to a separate code page.

Cache Thrashing. Next, we note that the CPU caches by their very nature are designed to mask memory access latency by removing the need for repeated DIMM accesses to the same address. Unfortunately, this has the effect of containing data read and write operations within the CPU, thus making them invisible to our memory interposition setup. As SGX does not allow us to flush enclave memory nor individually mark pages uncachable[1], we overcome caching with a cache thrashing technique which we now describe.

First, we allocate a large region of memory (four times the size of the LLC) and fill the contents with random data. Then, we move our program to a sibling core of the enclave, ensuring that the program shares all caches with the victim enclave. Finally, we repeatedly access our large memory space in sequential order. As more data is accessed than the size of all caches, they are forced to constantly evict all of their contents to cache new data. This allows us to trigger the logic analyzer as soon as the enclave variable of interest is read from the machine's memory by the target enclave.

6. Attacking Intel TEE Attestation

As outlined in Section 2.1, TEE implementations present on Intel Xeon Scalable server platforms use TME for encrypting the machine's memory, which in turn relies on deterministic AES-XTS with boot-time generated keys [12].

1. Possibly to avoid TLB poisoning attacks as described by Li et al. [41]

In this section we empirically verify the presence of this deterministic encryption, and utilize our memory interposer to recover a critical ECDSA signing key in the attestation chain. Finally, we show how this signing key can be used to forge arbitrary SGX and TDX quotes, resulting in a complete breach of the SGX and TDX ecosystems.

6.1. Verifying Deterministic Encryption

We begin by verifying that the encrypted ciphertext we observe on our memory interposer is a deterministic function of the physical address of memory and its contents.

Observing Determinism. We begin by creating an SGX enclave that writes and reads a specific virtual address repeatedly. To check that encryption is deterministic, we instruct our enclave to perform a series of write and read operations to a fixed virtual address in enclave memory, capturing the ciphertext read data after each step using our logic analyzer. Listing 1 is a simplified outline of our enclave, where we first write and read a fixed value `0x00`, then `0xFF`, and finally the initial value `0x00` again.

```
void ecall_experiment() {
    memset(global_memory, 0x00, burst_size);
    uncached_read(global_memory);
    wait_for_logic_analyzer_collection();

    memset(global_memory, 0xFF, burst_size);
    uncached_read(global_memory);
    wait_for_logic_analyzer_collection();

    memset(global_memory, 0x00, burst_size);
    uncached_read(global_memory);
    wait_for_logic_analyzer_collection();
}
```

Listing 1: SGX Enclave function for writing then reading a cache line.

Finally, we observe that the ciphertexts corresponding to the first and third read are identical, while the ciphertext corresponding to the second read is different. See Figure 6.

Impact of Virtual and Physical Addresses. Next, we check that the virtual address of memory does not affect the ciphertext. To that aim, we run our enclave at a different virtual address, but pin the new virtual address to the same physical address as in the first test. We observe that writing a fixed value of `0xFF` produces an identical ciphertext to that observed earlier. To check the impact of the physical address, we execute the enclave at the original virtual address but pin it to a different observable physical address. We find the encrypted memory contents are different, confirming the physical address affects the ciphertext.

Impact of SGX Enclave Measurements. Finally, as our platform supports multi-key TME, we verify that the measurements of an SGX enclave do not affect its AES-XTS key. To that aim, we modify the code of our enclave without changing the functionality and resign it with a different developer identity. This ensures that the enclave's `MRSIGNER` and `MRENCLAVE` are changed. We pin the new enclave to our chosen physical address, and observe writing

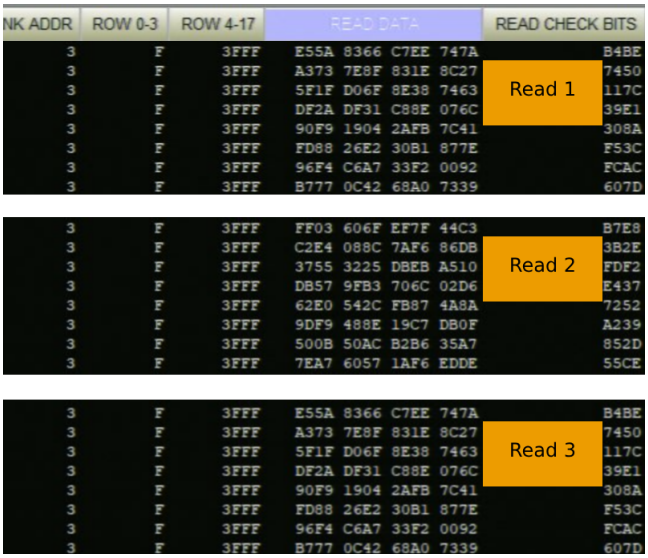


Figure 6: Ciphertext from three reads of enclave data. The first and last reads contain the same plaintext, while the second is different.

the same plaintext to this physical address has an identical ciphertext to our original enclave. Thus, we confirm that all SGX enclaves running on our system share the same memory encryption key and individual TME-MK keys are only applied to TDX TDs.

6.2. Secure Attestation Overview

We now give a high-level overview of the secure attestation process. For the interested reader, we provide extended details of the attestation process in Appendix B. The attestation mechanism is composed of two parts: (1) *local attestation*, where an SGX enclave or TD proves its own identity to the corresponding SGX/TDX Quoting Enclave (QE) and (2) *remote attestation*, where the QE uses this data to produce and sign an SGX/TDX quote, which is used to authenticate the proving enclave or TD to remote parties.

Local Attestation. Before an enclave or TD can prove (or attest) to a remote verifier, it must first prove its own identity to the Quoting Enclave (QE) through *local attestation*. First, the prover generates a report containing its measurements and optional provider-defined data. For an SGX enclave, the `EREPORT` instruction generates a report containing the enclave's `MRENCLAVE` and `MRSIGNER` values. TDs use the `SEAMREPORT` instruction to generate a report containing the TD's `MRTD` and `RTMR`, TDX module measurements, and version plus security state of the SEAM code. This report is verifiable with the `EVERIFYREPORT` instruction.

Remote Attestation. To create a remotely verifiable quote, the prover first passes its report to the corresponding QE. The QE verifies the report and uses it to produce and sign a quote, which is used to authenticate the proving enclave to remote parties. Intel's QE derives the repeatable signing key (its attestation key) using `EGETKEY` on a sealing key.

Provisioning Certificates. To prevent arbitrary key pairs from being used to sign and verify quotes, the QE provides

its attestation public key and measurements to Intel's PCE. The PCE derives a device-specific provisioning certification key (PCK) via `EGETKEY` on a root provisioning key and CPU version info. This PCK is a 256-bit ECDSA key and is used to certify the QE's identity and its attestation key. Unlike the memory encryption keys in Section 2.1, the PCK does not rotate on boot. In addition, Intel publishes certificates and certificate revocation lists for the PCKs in all genuine Intel platforms, ensuring a complete signature chain from DCAP quotes to Intel's Certificate Authority. By following this certificate chain, the remote verifier can ensures a quote originates from a genuine Intel platform.

6.3. Attacking Secure Attestation

We focus on the provisioning certification key (PCK) described above and notice that it is used to produce ECDSA signatures on both the SGX and TDX QE's attestation keys. Thus, extracting the PCK from the machine's PCE is sufficient to compromise SGX's and TDX's attestation process, allowing us to certify arbitrary attestation keys and subsequently forge remotely verifiable quotes, thereby compromising Intel's TEE guarantees.

Elliptic-Curve Digital Signature Algorithm (ECDSA). To sign and verify digital signatures, Intel's DCAP attestation uses ECDSA over the curve p-256 [42]. Given a group generator G for an additive group of order n over p-256, key generation is done by picking a random integer $d \leq n$ for the private key, and computing the public key $Q = [d]G$ using a scalar by point multiplication operation over p-256. Next, to sign a message m, the message is first hashed into $\log_2(n)$ bits, resulting in an integer z. The signer then picks a random nonce $k < n$ and computes $(x, y) = [k]G$, $r = x \bmod n$, and $s = k^{-1}(z + r \cdot d) \bmod n$, outputting (r, s) as the signature for m. To verify a signature, the verifier first computes the hash z from m, $u = z \cdot s^{-1} \bmod n$ and $v = r \cdot s^{-1} \bmod n$. Next, the verifier computes $(x', y') = [u]G + [v]Q$ and finally checks $r = x'$.

Scalar-Point Multiplication. To implement ECDSA, the PCE uses a custom cryptographic library called Intel Integrated Performance Primitives Cryptography (IPPCrypto) [43]. Algorithm 1 shows simplified pseudocode for the scalar-by-point multiplication function used for ECDSA signing operations. The inputs are a windowed and Booth encoded scalar value k and point P, and the output is another point $R = [k]P$. We emphasize that all table lookups and point additions are performed in constant time and do not leak information via memory access patterns.

To perform a scalar by point multiplication, Algorithm 1 first initializes a precomputation table T, such that $T[i] = [i] \cdot G$ for all $i = 0, \cdots, 16$, whose contents does not depend on the scalar k (Lines 1-5). The output point R is initialized by looking up $T[k_n]$, the most significant digit of k (Line 6). For each remaining digit k_i, Algorithm 1 multiplies R by 2^5 to account for moving to the next digit and then updates H to be the output of looking up $|k_i|$ in the table T, negating H if necessary based on the sign of k_i (Line 9). Next, H (which is now equal to $[k_i]G$) is added to R (Line 10), allowing

Algorithm 1 Simplified pseudocode of scalar-point multiplication algorithm in Intel's IPPCrypto library.

Require: A scalar k with Booth encoded digits $k_0 \cdots k_n$ valued $-16 \leq s_i \leq 16$ and a elliptic curve point G
Ensure: $R = [k]G$
1: $T[0] = 0$
2: $T[1] = G$
3: **for** $i = 2$ **to** 16 **do**
4: $\quad T[i] = T[i-1] + G$
5: $R = T[k_n]$ ▷ done in constant-time
6: **for** $i = n - 1$ **to** 0 **do**
7: $\quad R = 2^5 R$ ▷ using 5 point doubling operations
8: $\quad H = T[|k_i|]$ ▷ done in constant-time
9: $\quad$ **if** $k_i < 0$ **then** $H = -H$ ▷ done in constant-time
10: $\quad R = R + H$
11: **return** R

the algorithm to proceed with handling the next digit of k. Finally, once all the Booth encoded digits of k have been processed, Algorithm 1 returns $R = [k]G$ (Line 11).

6.4. PCK Key Extraction

As the nonce k for an ECDSA signature can be used to recover the private key d with straightforward algebra, we now recover k from a PCE signing operation and thus the PCK itself. Observing Algorithm 1, we note the variable H in Line 9 contains an elliptic curve point that is the result of a table lookup which directly depends on k_i. As the precomputation table T only contains 16 entries, there are only 32 possible values of H. Finally, we note that the set possible values of T, and therefore H, is public and does not depend on the value of the nonce k.

Creating a Ciphertext Mapping. To extract k, we construct an SGX enclave that simply writes known plaintext to a target virtual address, which we pin to a physical address `addr` observable with our interposer. Next, having ascertained in Section 6.1 that all SGX enclaves in our system share the same TME key, we write and read back each possible value of H one by one, collecting corresponding ciphertext samples. We then create a direct mapping from each observed ciphertext to its corresponding plaintext value of H as well to its corresponding k_i. This mapping is valid until the encryption keys change after a reboot.

Recovering k. With our control over enclave execution from Section 5.3, we execute the PCE and trigger a QE report certification which in turn triggers the PCE to perform an ECDSA signature on the QE's attestation keys. During the signing operation, we pin the physical address of H to a fixed `addr`, allowing us to observe the (encrypted) values of H using our interposer. Next, we use our enclave control channel from Section 5.3 to pause the execution at each loop iteration at Line 6 of Algorithm 1, arm the logic analyzer to trigger on memory accesses to `addr`, and record the data returned on the next read of H in Line 10. Once we recover all ciphertexts of H, we use our mapping to recover corresponding values of k_i, allowing us to obtain the ECDSA nonce k used by the machine's PCE. Finally,

we also record the public signature (r, s), which is the result of the PCE signing the QE's attestation key.

Recovering the Machine's PCK. Performing the above steps on our target machine described in Section 3 in an `UpToDate` attestation status, our attack took about 2 minutes to create the ciphertext mapping, and 13 minutes to execute the PCE and trace the ECDSA signing. With the ECDSA signature (r, s) and the corresponding nonce k in hand, we are able to recover the PCK from the machine's PCE, matching its corresponding public key to the Intel-signed certificate for our target machine.

Forging SGX and TDX Quotes. With the machine's PCK in hand, we recall from Section 6.2 that the PCK is used by the Provisioning Certification Enclave (PCE) to sign the quoting enclave (QE)'s attestation key. Thus, using the PCK we are able to sign our own attestation keys which do not originate from an Intel-signed QE. This in turn allows us to sign arbitrary SGX or TDX reports, creating fully verifiable attestation chains without actually running TDX virtual machines or SGX enclaves under any TEE protections, thereby completely breaking SGX and TDX security guarantees. We confirmed that the reports signed with our extracted PCK are fully verifiable by Intel's DCAP Quote Verification Library [44]. To the best of our knowledge, this is the first end-to-end complete breach of TDX protections on a machine in a fully trusted `UpToDate` status.

7. Attacking BuilderNet

Using our ability to forge TDX and SGX attestations, we now look at real-world applications relying on SGX and TDX. We first investigate BUILDERNET, a part of the Ethereum blockchain ecosystem utilizing TDX to provide guarantees of integrity, confidentiality, and trustworthiness. We run BUILDERNET software outside of TDX while still providing valid attestation, breaking these security guarantees, and demonstrate how a malicious operator could both extract configuration secrets and gain the ability to build arbitrary blocks or frontrun without being detected.

7.1. BuilderNet Overview

BUILDERNET is a network of *block builder* operators for Ethereum, a decentralized smart contract ecosystem. In Ethereum, transactions are committed in blocks, i.e., batches of transactions with a fixed maximum size. Every twelve seconds, a random Ethereum validator node is chosen to construct the contents of the next block, which is then validated by the other nodes [45]. The constructor decides not only the transactions to include but also their ordering, all of which affect the maximum extractable value (MEV) claimable by the validator [46], that is, the most value they can extract from a given block by including/excluding/reordering transactions. In order to maximize their MEV reward, many validators run middleware known as MEV-Boost, which allows them to source blocks from a decentralized block building market. Blocks from the market are built and proposed by block builders, who run algorithms to maximize MEV and have access to wide sources of transactions to build from [47]. BUILDERNET is one such group of builders that contribute to the MEV-Boost market, which end up building more than 90% of Ethereum blocks.

BUILDERNET's builders execute in TDX TEEs in order to provide fairness, confidentiality, and provable redistribution of MEV [48]. BUILDERNET has been operating since November 2024, and in May 2025, built at least 19,982 blocks worth approximately $2 million in MEV. The Beaverbuild network, which built at least 68,499 blocks worth approximately $7.5 million the same month, is in the process of to transitioning its builders to BUILDERNET as well [49].

BUILDERNET Security Guarantees. BUILDERNET relies on TEEs to ensures that all operators are running the correct builder software [50]. This aims to prove that blocks are built fairly with open, verifiable algorithms and software. Furthermore, attestation is used to construct confidential and attested TLS channels within builder services operating in the TDX VM. These channels are used to share transaction data and configuration secrets. With these channels, BUILDERNET additionally provides confidentiality of transactions, which is necessary to avoid frontrunning attacks [51]. In frontrunning attacks, a malicious party gains knowledge of a pending transaction and then constructs a new transaction with higher fees to ensure theirs is executed first. Frontrunning has a significant impact on users, undermining both fairness and integrity of the system, as well as driving up fees. To avoid being frontrun, users can send their transactions to private pools which are only shared with trusted parties such as BUILDERNET. In 2023, a malicious MEV-boost validator used frontrunning to extract $25 million from Ethereum [52], and 2021 work estimated that frontrunning had already lead to $34 million USD in profit [53].

BUILDERNET's Architecture. BUILDERNET consists of individual node operators, which communicate with a central BuilderHub service. The BuilderHub service is responsible for maintaining the registry of nodes and provisioning configuration and key information [54]. While BUILDERNET currently requires manual approval from Flashbots to participate as an operator, it aims to be permissionless, allowing any entity to be a node operator if proper attestation can be provided [48]. As a TDX virtual machine, BUILDERNET nodes run many services that communicate internally within the virtual machine's network. Internal services include a block builder, transaction pool, and Ethereum client. External requests are protected through the use of TLS and remote attestation [55]. A custom proxy is used to provide attested TLS termination for externally exposed VM ports and allow trusted services to make attested requests [56]. We note that the custom proxy is the only component of BUILDERNET that directly utilizes features of TDX, and other components do not distinguish whether they are run inside a TEE or not.

BUILDERNET Attestation. At the time of writing, BUILDERNET only supports remote attestation for Microsoft's Azure cloud offering of TDX [57]. To provide a more unified attestation across OSes and TEEs, Azure utilizes a virtual trusted platform module (TPM) for per-

forming measurements and producing quotes [58]. In TDX attestation, Azure's TDX quote is primarily used to authenticate the TPM, and the attestation process outputs two quotes, a TPM quote and a TDX quote. The TPM carries its own attestation key and can produce its own quotes following the TPM 2.0 specification. We now describe the overarching TDX attestation flow in Azure. First, the TPM's attestation key is combined with other information about the VM instance to produce *runtime data*. This *runtime data* is hashed and provided to the TDX quote generation service, which produces a TDX quote following the steps in Section 6.2. This TDX quote is used to authenticate the TPM's public key. Next, the user's requested report data is combined with the TPM's measurements and signed by the TPM's attestation key. In BUILDERNET, the requested report data is a nonce provided by the remote party, to prevent replaying of attestations. Finally, the original runtime data, signed measurements, and an event log used to reproduce measurements are combined and form the final TPM quote.

7.2. Attacking Private Transactions and Secrets

We now show how an attacker can utilize the ability to forge quotes detailed in Section 6.3 to break BUILDERNET's security guarantees, receiving confidential transactions, obtaining configuration secrets, and bypassing integrity checks.

Setup. We first set up a BuilderHub instance v0.2.1 [59], which acts as an innocent host. We initialize the instance to only allow officially published TEE measurements corresponding to Azure TDX instances with the BUILDERNET v1.4.0 image [60]. On a separate machine that does not support TDX, we act as an attacker only running a modified reverse proxy to perform forged attestation, based on proxy code with version 0.1.7 at [56]. We start a local transaction proxy and builder instance to receive blocks. As mentioned above, these components do not distinguish whether they are running outside of a TEE and can be deployed as-is.

Forging Azure Attestation Chains. We first obtain a set of measurements and Azure TEE quotes (TPM quote and TDX quote) from a public, trusted BUILDERNET node provided at [50]. We modify the attested proxy code to produce forged attestation information with the following steps. First, we stub out code that attempts to talk to the Azure vTPM and instead create our own RSA keypair to act as the TPM attestation key. We directly construct the TPM runtime data by starting with our reference data, overwriting the embedded attestation public key. We then hash the new runtime data, and create a forged TDX report attesting the new runtime data. Finally, we construct a TPM quote by re-signing the reference TPM measurements with our new TPM attestation key, and include the TPM event log verbatim from our reference. We verify the constructed attestation documents by starting a proxy server then utilizing the `attested-get` CLI tool from unmodified BUILDERNET code, which is recommended in BUILDERNET documentation [61]. We find that our documents validate and return identical measurements to the trusted public node.

Registering and Obtaining Confidential Data. We now utilize our forged attestation flow to register a BUILDERNET node. We start the proxy as a client to access BuilderHub endpoints protected by TEE attestation. First, we make requests to register our credentials and receive transactions from other BUILDERNET nodes [59]. Next, we retrieve secrets and configuration information from protected endpoints, along with the list of active BUILDERNET operators.

After following these steps, a malicious node operator now has access to secrets including a Ethereum account key, a signing key for relay submissions, access to the block bidding service, and information for accessing the private transaction archive [54]. The Ethereum key is used to sign built blocks and pay the validator, and contains $\$200,000$ at the time of writing [62]. The ability to see orderflow also enables a malicious operator to frontrun private transactions, and their ability to claim integrity allows the attacker to maintain trust and deny malicious activity while profiting.

8. Attacking Confidential TDX VMs and GPUs

We now explore two attacks that target a software stack reliant on TDX. More specifically, we focus on applications that build on top of DSTACK [63, 64], a software development kit (SDK) which allows for deployment of containerized applications to TEEs. DSTACK has already seen adoption as a confidential virtual machine (CVM) backend among multiple projects [65, 66, 67]. The DSTACK SDK exposes APIs for users to deploy applications on CVMs hosted on Intel TDX hardware. We note that in some instances, CVM projects have hinted at being decentralized networks themselves [68] or directly claim that upcoming updates will allow third-parties to become external hardware providers [69]. As such, we study a common threat model in which a user's application gets assigned to an attacker-controlled DSTACK node or deployed to a fully dishonest DSTACK cloud. We construct an attack producing outputs equivalent to that of legitimate DSTACK API calls, effectively mimicking the user's requested (trusted) environment outside of TDX, bypassing TEE security guarantees.

We portray this attack in two scenarios. First, we show how modified DSTACK Python SDK calls can deceive the user into believing their cloud provider's Jupyter Notebook is running on a confidential environment, when in reality it is running outside of TDX. This allows us to view the notebook's contents in plain text. Then, we demonstrate how we can use NVIDIA Confidential Computing (CC) attestations from an independent device as if they were our own, undermining many of the guarantees provided by CC. Concretely, we break an LLM frontend which provides Intel and NVIDIA attestations to prove it is running inside a trust domain, and spin up an instance outside of TDX and CC.

8.1. DSTACK Attack Methodology

We use the ability to forge quotes detailed in Section 6.3 to build a malicious version of DSTACK's API in which attestation calls are intercepted and forged.

Setup. First, we initialize an innocent DSTACK instance in TDX as a ground truth. We use this instance to obtain baseline measurements and save them for the next steps in our attack. Separately, we instantiate the DSTACK software outside TDX, modifying it to produce forged attestations.

Forging TDX Attestations. As DSTACK starts, it logs a series of *events* and related hashes. (e.g., a hash of the filesystem). We use these event hashes to calculate four RTMRs for the TDX report. Taking these RTMRs and the previously collected baseline measurements, we compile our TDX report and create a quote. After signing it with our attestation key, we have a forged quote that will verify, attesting our malicious API is an unmodified one. Since DSTACK is a CVM deployment platform, we continue by breaking security guarantees of several products built on it.

8.2. Constructing Un-Confidential VMs

We first look at providers that offer Jupyter notebooks running inside CVMs, such as PHALA NETWORK [65]. A notebook hosted through DSTACK allows users to run confidential Python workflows within an attested environment. When a user connects directly to the notebook, they can further verify its TEE status within the notebook's Python runtime by utilizing the DSTACK Python SDK [63] or directly querying DSTACK's RPC API [70], obtaining TDX quotes. Both methods also allow a custom data field to be included in the report to prevent replay attacks. After obtaining the quote, the user must then verify it independently.

Our attack creates a malicious API as described in Section 8.1. To this end, quote requests to the modified API return valid TDX quotes irrespective of the underlying hardware. We then expose our API to the user, posing as DSTACK's native RPC endpoint. Since DSTACK's Python SDK also uses this endpoint for quotes, both direct RPC calls and Python function calls return valid forged attestations. Lastly, we expose a regular Jupyter notebook through the provider's desired method, allowing users to connect to it. After these steps, we have successfully created an environment in which the DSTACK tools accessible to the user return legitimate TDX quotes which pass remote attestation, while in reality the underlying notebook runs outside of TDX protections, with no security or privacy guarantees.

8.3. GPU Relay Attacks

Another product that builds on top of DSTACK is PHALA NETWORK and NEAR AI's private machine learning SDK [66, 71]. In particular, we investigate their VLLM-PROXY, a privacy-preserving LLM frontend that attests to running inside TDX. The attestation and its measurements aim to ensure the frontend must be running with a specific backend and configuration that enforces desired security requirements, such as in-GPU confidentiality of user's queries.

Securing GPU Workloads. This SDK is intended to run in a CVM equipped with an NVIDIA Confidential Computing (CC) enabled GPU, such as the H100 [72]. An NVIDIA CC GPU encrypts PCIe traffic, which extends the trust domain to encompass the GPU too. To accomplish this, it provides a hardware attestation that proves the CPU interfaces directly with a genuine CC GPU. However, the attestation does not identify *what* the GPU is running. Ordinarily, this is not an issue as the trust domain is expected to perform that role.

Attack Setup. We then install an NVIDIA 3060 GPU in our local attack device, set up VLLM-PROXY outside of TDX, and override all functionality that interfaces with TDX or CC. We modify it to forward the GPU attestation call to an external H100-equipped server. When a user requests attestations from our malicious VLLM-PROXY server, it generates a forged TDX quote and on-demand fetches a GPU attestation from a rented server equipped with an H100 running our TDX VM.[2] As NVIDIA does not bind the H100 to identities of specific VMs, our malicious VLLM-PROXY successfully passes both TDX and CC attestations.

Consequences. As part of our attack, we are able to setup the VLLM-PROXY end-user software, and have it fetch an attestation from our server. The software successfully verifies the attestation with NVIDIA, and recommends using AUTOMATA's on-chain DCAP verification service to verify the TDX quote. We additionally demonstrate how we subvert AUTOMATA's offerings in Appendix D as this also relies on TDX. As such, a user is (incorrectly) cryptographically convinced that their GPU calls are executing inside a TDX VM with an NVIDIA CC instance, while in reality it is running outside of any TEE guarantees. Generative AI is a rapidly growing field, estimated to grow to $4.8 trillion by 2033 [73]. With this attack, a provider could multiplex a single H100 to provide attestations for hundreds of instances. Similarly, a malicious LLM host could covertly log all LLM requests while claiming confidentiality. While an individual H100 still retails for over $20,000 as of the time of writing, a spot instance costs less than $2.50 per hour [74]. With on-demand attestations, a provider could also reuse older hardware yet still attest a workload is running in an H100-equipped trust domain.

9. Directly Extracting Secrets from Enclaves

While our previous attacks focus on utilizing our extracted PCK and attestation keys to forge arbitrary DCAP quotes, we now show that the use of deterministic memory encryption allows us to break the security guarantees of an SGX enclave without ever even looking at the attestation process. We are able to directly reconstruct a cryptographic private key during the execution of an application enclave from the SECRET network. Then, we utilize our recovered key to completely break the confidentiality requirements of SECRET without needing to forge attestation information.

9.1. SECRET Network Overview

SECRET network was one of the first TEE-based blockchains to reach significant adoption, launching its

2. We note that this TDX VM honestly performs the CC attestation process on the cloud server, returning the results to us. No attack-specific code is ever run on the rented server and no data is retained afterwards.

privacy-preserving smart contracts feature in 2020. Most blockchains are completely transparent by design, allowing users to review all contract state and transaction data. To preserve the privacy of this data, numerous projects have proposed an alternative TEE-based approach [75, 76, 77, 78, 79, 80] by moving smart contracts into the enclave.

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 public key, and include the ciphertext in a transaction. The corresponding private key, derived from the *consensus seed*, is replicated throughout the network and sealed within SGX enclaves. To allow rolling the consensus seed in the event of compromise, the SECRET network maintains both the initial and current seeds.

Registering Validator Nodes. Validators are operators on the SECRET network which execute transactions and validate a new consensus state after each transaction. To execute a transaction, validators must be able to decrypt transactions and thus receive the consensus seed. 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. The attestation report combines the DCAP quote and collateral, where the first 32 bytes of the DCAP quote's report data field 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 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.

Setup. We use the hardware setup from Section 3 to set up a SECRET validator joining their Pulsar-3 testnet, running an unmodified SECRET binary and enclave (version 1.17.1).

9.2. Attacking ECDH Directly

We observe that, rather than using an extracted ECDSA key to sign our own DCAP quotes, our memory interposition setup actually allows us to reconstruct the node's ECDH private key from an unmodified enclave, highlighting that these issues affect all enclave developers.

The ECDH algorithm performs a scalar multiplication on the consensus seed exchange public key and the node's private key. SECRET's enclave performs this multiplication using the Montgomery Ladder algorithm, which slides over the private key bits using a 2-bit window (previous and current) with every iteration. See Listing 2. The code computes `choice`, an XOR of these two bits (Line 4), to determine if it should swap two bits (`x0` and `x1`) or not (Line 5). Thus, if we can observe whether `x0` is the same or not before and after executing this conditional swap, we can recover the value of `choice` in each iteration, allowing us to recover the private key. See Appendix Cfor a detailed explanation.

```
1  let mut bits = scalar.bits_le().rev();
2  let mut prev_bit = bits.next().unwrap();
3  for cur_bit in bits {
4    let choice = prev_bit ^ cur_bit;
5    conditional_swap(x0, x1, choice);
6    differential_add_and_double(x0, x1, aff_u);
7    prev_bit = cur_bit;
8  }
```

Listing 2: SECRET's simplified Montgomery Ladder implementation.

Controlling the Code Execution. As we want to observe the state of `x0` before and after the conditional swap (Line 5), we have to be able to halt the enclave's execution at the right locations. We perform a controlled-channel attack [40] by unmapping enclave pages before and after the conditional swap, executing the enclave until a page fault occurs and then remapping the page to reach the correct locations. Appendix C describes this in more detail.

Locating x0. To obtain the exact address of `x0` we sign Secret's enclave to run in debug mode and run it with `sgx-gdb` attached. Additionally, to ensure that our virtual address space is consistent between runs, we disable ASLR.

Collecting Traces. Next, using our techniques from Section 5, we observe the DRAM traffic to obtain the ciphertext before and after the conditional swap in each loop iteration. For each loop iteration, we infer that `choice` is 0 if the ciphertexts are the same, and 1 otherwise. We ultimately recover a stream of 252 bits, the state of `choice`.

Recovering the Private Key. We execute our attack and find it takes approximately 90 minutes to collect all necessary traces. Given a trace of `choice`'s state, we then proceed to reconstruct the private key. Note that we need the previous bit in each loop iteration to determine the value of the current bit, but that we are missing the last three bits and the first bit. Thus, we simply try all possibilities 0 and 1, yielding $2^4 = 16$ possible candidate private keys.

Verifying the Private Key. To identify the correct private key, we use each candidate key to forge an encrypted consensus seed to pass to `init_node`, which only decrypts our encrypted consensus seed if the candidate private key is correct, as AES-SIV is an authentication (AEAD) scheme.

Decrypting the Consensus Seed. We now broadcast a registration transaction onto SECRET's blockchain with our

node's public key to obtain the encrypted consensus seed. Finally, we decrypt it with the ECDH-derived shared secret.

Decrypting Transactions. One of the main applications of the SECRET network is to preserve the privacy of transactions, allowing parties to privately transfer control over digital assets. As previously demonstrated by [20], access to the initial and current consensus seeds lets us directly decrypt any transaction on SECRET's network, thus allowing us to completely breach SECRET's confidentiality guarantees without ever even seeing an attestation key.

10. Attacking Confidential SEV-SNP VMs

Moving beyond Intel's TEE implementation, in this section we demonstrate how our attack affects AMD's SEV-SNP (Secure Encrypted Virtualization-Secure Nested Paging) [2] running on EPYC servers based on the Zen 5 architecture. We note that these incorporate AMD's ciphertext hiding features, aiming to mitigate prior software-based ciphertext attacks.

More specifically, following [24] we attack a SEV-SNP protected confidential virtual machine (CVM) running OpenSSL's ECDSA signature operations. However, as ciphertext hiding is enabled, we are unable to recover the secret key using any software-based ciphertext attacks. After using our bus interposition setup to confirm that SEV-SNP's memory encryption exhibits the same deterministic behavior found in Section 6.1, we proceed to key extraction using our logic analyzer setup.

10.1. AMD SEV-SNP Overview

AMD SEV-SNP. In 2016, AMD introduced SEV as a first-level offering for CVMs and memory encryption, with additional features being added over time, culminating in the release of SEV-SNP in 2020 [2]. SEV-SNP provides memory confidentiality and integrity protection, encryption of VM state, and multi-level paging. Unlike TDX, SEV-SNP allows the hypervisor to observe the contents of encrypted memory, allowing software exploitation of deterministic encryption [7]. To address various attacks resulting in part from this ability to view ciphertexts from software, AMD has introduced a feature known as Ciphertext Hiding [6] starting with 5th generation EPYC server processors, which prevents the hypervisor from observing encrypted memory contents. A VM can set a policy ensuring it was launched with Ciphertext Hiding enabled.

AMD SME. The memory encryption engine powering SEV-SNP is known as AMD Secure Memory Encryption (SME) [81]. Similarly to Intel's use of AES-XTS, AMD SME utilizes the XEX or XTS mode on a block size of 16 or 32 bytes [82]. To verify SME results in deterministic encryption, we replicated the findings in Section 6.1, observing deterministic encryption based on the physical address and plaintext data.

Target Machine. We targeted a server with an AMD EPYC 9015 CPU installed on an ASRockRack BERGAMOD8-2L2T motherboard, running BIOS version 10.02 with CPU microcode `0xb002116`. The machine has a single 16GB RDIMM of DDR5 memory, capable of running at 4800 MT/s. For software, our target machine runs Ubuntu 24.04 with a custom Linux kernel based on version 6.14.6 containing patches based on SEV-Step [83] and to enable Ciphertext Hiding [84]. The machine has SEV-SNP firmware version 1.55.54. Furthermore, we ensured that the SEV firmware reports that Ciphertext Hiding is fully enabled.

Enclave Memory and Control Channel. Unlike with Intel SGX and TDX, AMD SEV-SNP does not require all memory controller channels to be filled, and thus we do not need to control page allocation as done in Section 5.2. In order to obtain an equivalent control channel to the SGX one described in Section 5.3, we utilize the SEV-Step [83] framework, merged on top of a more recent Linux kernel. This framework provides a page-granular control channel primitive identical to the one on SGX. Finally, unlike SGX and TDX, AMD SEV-SNP allows caching for a specific memory page to be disabled via mTRRs [85], so cache thrashing is unnecessary.

10.2. An OpenSSL ECDSA Key Recovery Attack

To show the impact of interposition attacks on CVMs protected with SEV-SNP, we recover the ECDSA private key from a single ECDSA signing operation in the OpenSSL cryptographic library. As the cryptographic code is constant-time, we are only able to recover the private key due to leaking information about the plaintext contents of memory via deterministic encryption.

Setup. As a target, we set up an SEV-SNP protected CVM running Ubuntu 24.04 with OpenSSL version `3.0.13` as packaged in Ubuntu. We created a simple test application which creates a random ECDSA key for the curve `secp256k1`, used in Bitcoin wallets, and signs a message with the OpenSSL `EVP_DigestSign` API.

OpenSSL Montgomery Ladder. As part of an ECDSA signature, OpenSSL performs a scalar-by-point multiplication with a Montgomery Ladder algorithm [86]. A simplified version is shown in Listing 3. Similarly to the Montgomery Ladder implementation in Section 9, OpenSSL slides over windows of 2 bits, performing a conditional swap then an add-and-double. For each window, the code computes a bit `choice` as the XOR of the previous `choice` and current scalar bit (Line 2). Then, a constant time conditional swap is performed on two points `p0` and `p1` based on `choice` (Line 3), and an add and double ladder step is performed on both points (Line 4). If we can determine if `p0` and `p1` were swapped, we can recover the sequence of `choice` bits and thus the complete scalar, which in the case of ECDSA, is the secret nonce.

Key Extraction. Unlike in Cipherleaks' [24] attack on the VM register state, we instead observe the contents of the point `p0` in memory with our interposer setup. We note that if the contents of `p0` change between before and after executing Line 3, then the `choice` bit must have been 1, and vice-versa. To obtain the contents of `p0` at

```
1  for (i = scalar_bits - 1; i >= 0; i--) {
2      current_bit = bit(scalar, i) ^ previous;
3      conditional_swap(p0, p1, bit);
4      differential_add_and_double(p0, p1);
5      previous ^= choice;
6  }
```

Listing 3: OpenSSL's simplified Montgomery Ladder implementation.

these times, we utilize the page-granular control channel to interrupt the CVM once when `conditional_swap` reads `p0` in Line 3, and again during the execution of the `add_and_double` ladder step in Line 4. Each time, we arm our logic analyzer before resuming the CVM, recording the memory contents upon first read. We find our attack takes 1 hour to record 514 memory reads and recovers all the `choice` bits with 100% accuracy. With the nonce in hand, we recovered the ECDSA signing key via the same process as Section 6.3 within a few seconds.

11. Mitigations and Conclusions

Avoiding Deterministic Encryption. We note that one of the fundamental issues with server TEEs is the use of deterministic memory encryption. This is a step back from the prior SGX implementation on CPUs meant for the PC market, where the hardware used on-die Merkle trees to provide confidentiality and integrity guarantees. Furthermore, to prevent attackers from observing the same ciphertexts when the same plaintext is stored at the same physical address, these implementations also provided freshness guarantees [8]. Unfortunately, the use of Merkle trees carries substantial overhead limiting the EPC size to 512 MB. Thus, reconciling these goals and obtaining scalable memory encryption without sacrificing these guarantees warrants further research. Performance issues aside, for current CPUs Intel has indicated that it is impossible to issue a microcode to update the inner workings of the memory encryption engine. Thus, we expect bus interposition to remain a viable attack vector in the foreseeable future.

Bus Speeds. The ability to lower the memory frequency from 4800 MT/s to 3200 MT/s makes our attacks somewhat cheaper. However, we warn against using bus speeds as attack mitigations as high speed interposition setups are likely to become readily-available in the near future.

Adding Entropy. One effective solution to solving issues resulting from deterministic encryption is to ensure that every 128-bit block has sufficient entropy to prevent ciphertext repetition. This can be done via a custom memory layout [7] where each 64-bit data block is followed by a 64-bit counter with a random initial value. While effective at mitigating our attack in principle, we note that this needs to be applied consistently throughout the entire software stack including the CVM's OS kernel and the Intel-controlled and signed PCE enclave. Moreover, this solution does not mitigate other issues such as memory integrity or relay attacks.

Permissioning Systems and Secure Multiparty Computations (MPC). We note that another common issue is that many TEE deployments are permissionless, i.e., a node can simply present a trusted attestation status without further context to be entrusted with security-critical roles. Unfortunately, this implies that secrets are often physically provisioned to attackers, where they can be extracted. Thus, TEE developers should either limit deployment to highly reputable cloud providers with strong physical security practices, or they must accept the possibility of a breach. To mitigate the latter scenario, MPC-based systems can distribute the trust among multiple parties, requiring a simultaneous breach to extract secrets [87]. However, as MPC-based constructions result in significant overhead, we leave the task of efficiently implementing these to future work.

Location Verification and CPU Whitelisting. A final countermeasure is to augment the attestation mechanism present in many server-grade TEEs with location or cloud verification primitives. These in turn will allow users to ascertain that TEE hardware is physically located in secure cloud environments as opposed to adversarial hands. An even more restrictive approach would be to allow users to whitelist specific CPU instances which are known to be in physically secure locations, preventing other CPUs from the same architecture or model from obtaining secrets. However we note that both approaches will require changes to the current Intel-controlled attestation protocol, with cloud vendors being major stakeholders aimed at safeguarding against any potential of infrastructure fingerprinting via user APIs.

Conclusion. In this paper we explored the security implications of mounting memory interposition attacks on DDR5-based TEE implementations, demonstrating how hobbyists can construct a memory interposition setup for under $1,000. Next, we used our setup to attack modern TEE hardware across Intel, AMD and NVIDIA, demonstrating for the first time significant TEE weakness in these machines. Finally, we use our ability to extract TDX and SGX attestation keys from machines in fully trusted status to breach the confidentiality and integrity guarantees of various real-world deployments as presented in our case studies.

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 and Zama.

References

[1] S. Johnson, R. Makaram, A. Santoni, and V. Scarlata, "Supporting Intel SGX on multi-package platforms," 2025. [Online]. Available: https://arxiv.org/abs/2507.08190

[2] AMD, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[3] A. Seto, O. K. Duran, S. Amer, J. Chuang, S. van Schaik, D. Genkin, and C. Garman, "Wiretap: Breaking server sgx via dram bus interposition," in *2025 SIGSAC Conference on Computer and Communications Security (CCS '25)*. Association for Computing Machinery, 2025. [Online]. Available: https://wiretap.fail

[4] J. De Meulemeester, D. Oswald, I. Verbauwhede, and J. Van Bulck, "Battering RAM: Low-cost interposer attacks on confidential computing via dynamic memory aliasing," in *47th IEEE Symposium on Security and Privacy (S&P)*, May 2026.

[5] J. De Meulemeester, L. Wilke, D. Oswald, T. Eisenbarth, I. Verbauwhede, and J. Van Bulck, "BadRAM: Practical memory aliasing attacks on Trusted Execution Environments," in *IEEE S&P*, 2024.

[6] AMD, "SEV ciphertext side channel attacks," 2025. [Online]. Available: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3021.html

[7] ——, "Technical guidance for mitigating effects of ciphertext visibility under AMD SEV," 2022. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/bulletin/technical-guidance-for-mitigating-effects-of-ciphertext-visibility-under-amd-sev.pdf

[8] V. Costan and S. Devadas, "Intel SGX explained," 2016.

[9] Intel, "Intel® Trust Domain Extensions," 2022. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/690419

[10] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX demystified: A top-down approach," *ACM Computing Surveys*, 2024.

[11] Intel, "Intel® Hardware Shield – Intel® Total Memory Encryption," 2022. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf

[12] I. C. Hormuzd Khosravi, "Runtime Encryption of Memory with Intel® Total Memory Encryption - Multi-Key," 2022. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-10/intel-total-memory-encryption-multi-key-whitepaper.pdf

[13] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Rogue in-flight data load," in *IEEE S&P*, 2019.

[14] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *ACM CCS*, 2019.

[15] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *ACM CCS*, 2019.

[16] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[17] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *IEEE S&P*, 2021.

[18] P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC leak: Architecturally leaking uninitialized data from the microarchitecture," in *USENIX Security*, 2022.

[19] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "CROSSTALK: Speculative data leaks across cores are real," in *IEEE S&P*, 2021.

[20] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, D. Genkin, A. Miller, E. Roonen, Y. Yarom, and C. Garman, "SoK: SGX.Fail: How stuff gets exposed," in *IEEE S&P*, 2021.

[21] L. Wilke, F. Sieck, and T. Eisenbarth, "TDXdown: Single-stepping and instruction counting attacks against Intel TDX," in *ACM CCS*, 2024.

[22] B. Schlüter, S. Sridhara, M. Kuhne, A. Bertschi, and S. Shinde, "HECKLER: Breaking confidential VMs with malicious interrupts," in *USENIX Security*, 2024.

[23] B. Schluter, S. Sridhara, A. Bertschi, and S. Shinde, "WeSee: Using malicious #vc interrupts to break amd sev-snp," in *IEEE S&P*, 2024.

[24] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel," in *USENIX Security*, 2021.

[25] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on AMD SEV-SNP," in *IEEE S&P*, 2022.

[26] Y. Yuan, Z. Liu, S. Deng, Y. Chen, S. Wang, Y. Zhang, and Z. Su, "CipherSteal: Stealing input data from TEE-shielded neural networks via ciphertext side channels," in *IEEE S&P*, 2025.

[27] ——, "HyperTheft: Thieving model weights from TEE-shielded neural networks via ciphertext side channels," in *ACM CCS*, 2024.

[28] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security*, T. Holz and S. Savage, Eds., 2016.

[29] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An Off-Chip attack on hardware enclaves via the memory bus," in *USENIX Security*, 2020.

[30] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against AMD's Secure Encrypted Virtualization," in *ACM SIGSAC*, 2021.

[31] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia, "VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface," in *USENIX Security*, 2021.

[32] Canonical, "index : ubuntu/+source/linux-intel," 2025.

[Online]. Available: https://git.launchpad.net/ubuntu/+source/linux-intel?h=ubuntu%2Fnoble-devel
[33] Intel, "An introduction to the Intel® QuickPath Interconnect," 2009. [Online]. Available: https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html
[34] ——, "Intel® xeon® processor scalable family technical overview," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html
[35] ——, "Hardware selection," 2025. [Online]. Available: https://cc-enabling.trustedservices.intel.com/intel-tdx-enabling-guide/03/hardware_selection/
[36] P. Jattke, M. Wipfli, F. Solt, M. Marazzi, M. Bölcskei, and K. Razavi, "ZenHammer: Rowhammer attacks on AMD Zen-based Platforms," in *USENIX Security*, 2024.
[37] Intel, "Memory address translation DSM interface," 2018. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/603354
[38] I. UEFI Forum, "Acpi-defined devices and device specific objects," 2025. [Online]. Available: https://uefi.org/specs/ACPI/6.6/09_ACPI_Defined_Devices_and_Device_Specific_Objects.html
[39] Intel, "linux/drivers/acpi/acpi_adxl.c," 2023. [Online]. Available: https://github.com/torvalds/linux/blob/v6.8/drivers/acpi/acpi_adxl.c
[40] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P*, 2015.
[41] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "TLB poisoning attacks on AMD secure encrypted virtualization," in *ACSAC*, 2021.
[42] Intel, "Intel® Trust Domain Extensions Data Center Attestation Primitives (Intel® TDX DCAP): Quote generation Library and Quote Verification Library," 2023. [Online]. Available: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_TDX_DCAP_Quoting_Library_API.pdf
[43] ——, "sources/ippcp/pcpgfpec_mul.c," 2024. [Online]. Available: https://github.com/intel/cryptography-primitives/blob/7d6ac3/sources/ippcp/pcpgfpec_mul.c#L35
[44] ——, "GitHub - intel/sgx-tdx-dcap-quoteverificationlibrary," 2025. [Online]. Available: https://github.com/intel/SGX-TDX-DCAP-QuoteVerificationLibrary
[45] E. D. Documentation, "Blocks," 2024. [Online]. Available: https://github.com/ethereum/ethereum-org-website/blob/bd4e95/public/content/developers/docs/blocks/index.md
[46] ——, "Maximal extractable value (MEV)," 2025. [Online]. Available: https://ethereum.org/en/developers/docs/mev/
[47] Flashbots, "Mev-boost overview," 2025. [Online]. Available: https://docs.flashbots.net/flashbots-mev-boost/introduction
[48] BuilderNet, "What is buildernet," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/9f82a4/docs/what-is-buildernet.mdx
[49] beaverbuild, "buildernet.txt," 2025. [Online]. Available: https://beaverbuild.org/buildernet.txt
[50] BuilderNet, "Verifiable system integrity," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/9f82a4/docs/verifiable-system-integrity.mdx
[51] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *IEEE S&P*, 2020.
[52] B. Liu, "Ethereum validator goes rogue, frontruns MEV bots for $25m," 2023. [Online]. Available: https://blockworks.co/news/validator-frontruns-mev-bots
[53] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2101.05511
[54] BuilderNet, "Architecture overview," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/9f82a4/docs/architecture.mdx
[55] E. Systems, "Attested TLS (aTLS)," 2023. [Online]. Available: https://github.com/edgelesssys/constellation/blob/8f21972aecfd44b1efc8d8e917479feda58aed5c/internal/atls/README.md
[56] flashbots, "GitHub - flashbots/cvm-reverse-proxy at v0.1.7," 2025. [Online]. Available: https://github.com/flashbots/cvm-reverse-proxy/tree/v0.1.7
[57] BuilderNet, "Operating a node," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/9624d6/docs/operating-a-node.mdx
[58] Microsoft, "Confidential VM Guest Attestation Design Detail," 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/guest-attestation-confidential-virtual-machines-design
[59] flashbots, "GitHub - flashbots/builder-hub at v0.2.1," 2025. [Online]. Available: https://github.com/flashbots/builder-hub/tree/v0.2.1
[60] BuilderNet, "Buildernet measurements," 2025. [Online]. Available: https://measurements.builder.flashbots.net/
[61] BuilderNet, "Orderflow encryption and Attestation," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/e8c66c/docs/encryption-attestations.mdx
[62] BuilderNet, "Public identification," 2025. [Online]. Available: https://github.com/BuilderNet/website/blob/9f82a4/docs/public-identity.mdx
[63] F. Phala Network, "GitHub - dstack-TEE/dstack," 2025. [Online]. Available: https://github.com/Dstack-TEE/dstack
[64] P. Network, "Dstack — phala network docs," 2025. [Online]. Available: https://docs.phala.network/overview/phala-network/dstack
[65] P. N. Docs, "Your first CVM deployment," 2025.

[Online]. Available: https://docs.phala.network/phala-cloud/getting-started/start-from-cloud-ui
[66] P. N. NEAR AI, "GitHub - nearai/private-ml-sdk," 2025. [Online]. Available: https://github.com/nearai/private-ml-sdk
[67] E. Plugins, "TEE core plugin for Eliza," 2025. [Online]. Available: http://github.com/elizaos-plugins/plugin-tee
[68] N. AI, "Decentralization," 2025. [Online]. Available: https://docs.near.ai/decentralization/
[69] S. Protocol, "Testnet limitations," 2025. [Online]. Available: https://docs.superprotocol.com/marketplace/limitations/
[70] Dstack-TEE, "DStack guest agent RPC API documentation," 2025. [Online]. Available: https://github.com/Dstack-TEE/dstack/blob/master/sdk/curl/api.md
[71] N. AI, "Building next-gen NEAR AI infrastructure with TEEs," 2025. [Online]. Available: https://near.ai/blog/building-next-gen-near-ai-infrastructure-with-tees
[72] NVIDIA, "Confidential Compute on NVIDIA Hopper H100," 2023. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf
[73] U. Nations, "Technology and innovation report 2025," 2025. [Online]. Available: https://unctad.org/system/files/official-document/tir2025_en.pdf
[74] Google, "Pricing — Spot VMs — Google Cloud," 2025. [Online]. Available: https://cloud.google.com/spot-vms/pricing
[75] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *EuroS&P*, 2019.
[76] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "FastKitten: Practical smart contracts on bitcoin," in *USENIX Security*, 2019.
[77] R. Sinha, S. Gaddam, and R. Kumaresan, "LucidiTEE: A TEE-blockchain system for policy-compliant multiparty computation with fairness," *Cryptology ePrint Archive*, 2019.
[78] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private data objects: an overview," *arXiv preprint arXiv:1807.05686*, 2018.
[79] G. Kaptchuk, I. Miers, and M. Green, "Giving state to the stateless: Augmenting trustworthy computation with ledgers," *Cryptology ePrint Archive*, 2017.
[80] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, "Obscuro: A bitcoin mixer using trusted execution environments," in *ACSAC*, 2018.
[81] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," 2021. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf
[82] AMD, "SEV Secure Nested Paging Firmware ABI Specification," 2025. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf
[83] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth, "SEV-step a single-stepping framework for AMD-SEV," in *TCHES*, 2023.
[84] A. Kalra, "[patch v7 0/7] add SEV-SNP CipherTextHiding feature support," 2025. [Online]. Available: https://lore.kernel.org/all/47061b1e0bf8ca10aa6a7ca76ba60ceb913653bb.1752869333.git.ashish.kalra@amd.com/
[85] R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, "CacheWarp: Software-based fault injection using selective state reset," in *USENIX Security*, 2024.
[86] T. O. P. Authors, "openssl/crypto/ec/ec_mult.c at openssl-3.0.13," 2024. [Online]. Available: https://github.com/openssl/openssl/blob/openssl-3.0.13/crypto/ec/ec_mult.c#L116
[87] P. Wu, J. Ning, J. Shen, H. Wang, and E.-C. Chang, "Hybrid trust multi-party computation with trusted execution environment," in *NDSS*, 2022.
[88] Intel, "Intel® SGX Data Center Attestation Primitives (Intel® SGX DCAP)," 2019. [Online]. Available: https://www.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-dcap-ecdsa-orientation.pf
[89] ——, "Remote Attestation for Multi-Package Platforms using Intel® SGX Datacenter Attestation Primitives (DCAP)," 2024. [Online]. Available: https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/Intel_SGX_DCAP_Multipackage_SW.pdf
[90] Automata, "Automata - Verifiable attestations with machines," 2025. [Online]. Available: https://www.ata.network/
[91] T. of Bits, "Automata DCAP attestation and onchain PCCS security assessment," 2025. [Online]. Available: https://github.com/trailofbits/publications/blob/master/reviews/2025-02-automata-dcap-attestation-onchain-pccs-securityreview.pdf
[92] A. Network, "GitHub - automata-network/automata-dcap-attestation," 2025. [Online]. Available: https://github.com/automata-network/automata-dcap-attestation
[93] R. Zero, "Risc zero," 2025. [Online]. Available: https://risczero.com/
[94] S. Labs, "GitHub - succinctlabs/sp1," 2025. [Online]. Available: https://github.com/succinctlabs/sp1
[95] ElizaLabs, "GitHub - elizaos-plugins/plugin-dcap," 2025. [Online]. Available: https://github.com/elizaos-plugins/plugin-dcap
[96] E. Systems, "Espresso SGX TEE verifier," 2025. [Online]. Available: https://eng-wiki.espressosys.com/mainch29.html
[97] Puffer, "Introducing UniFi Rollup — Puffer UniFi Docs," 2025. [Online]. Available: https://docs-unifi.puffer.fi/

Appendix A.
Reverse Engineered Physical Address Mapping

We provide full details of reverse engineered physical address mapping functions from Section 5.1. We denote a 64-bit physical address as A, with A_0 referring to the least significant bit of A and A_{63} the most significant bit of A. Similarly, we denote the the least significant bit of an n-bit location as $\texttt{Location}_0$, and $\texttt{Location}_{n-1}$ the most significant bit. $X_{[a...b]}$ refers to the set of bits from a to b inclusive. First, we list all bits with linear relationships and their formulas.

Location	Physical Address Bits
$\texttt{ChannelAddress}_{[0...7]}$	$A_{[0...7]}$
$\texttt{ChannelAddress}_8$	A_{10}
$\texttt{ChannelAddress}_{[9...27]}$	$A_{[12...30]}$
$\texttt{RankAddress}_{[0...5]}$	$A_{[0...5]}$
$\texttt{RankAddress}_6$	A_7
$\texttt{RankAddress}_7$	A_{10}
$\texttt{RankAddress}_{[8...26]}$	$A_{[12...30]}$
Location	**Physical Address Bits**
$\texttt{Column}_{[0...3]}$	$A_{[2...5]}$
$\texttt{Column}_4$	A_{18}
$\texttt{Column}_5$	A_{23}
$\texttt{Column}_6$	A_{28}
$\texttt{Column}_7$	A_{29}
$\texttt{Column}_8$	A_{30}
$\texttt{ChipSelect}$	A_6
$\texttt{BankGroup}_0$	$A_7 \oplus A_{30}$
$\texttt{BankGroup}_1$	$A_{10} \oplus A_{32} \oplus A_{33} \oplus A_{34} \oplus A_{35} \oplus A_{36} \oplus A_{37}$
$\texttt{BankGroup}_2$	$A_{12} \oplus A_{35} \oplus A_{36} \oplus A_{37}$
$\texttt{MCID}_0$	$A_8 \oplus A_{14} \oplus A_{22}$
$\texttt{MCID}_1$	$A_8 \oplus A_{11} \oplus A_{17} \oplus A_{25}$
$\texttt{ChannelId}$	$A_9 \oplus A_{15} \oplus A_{23}$
$\texttt{Bank}_0$	$A_{13} \oplus A_{33} \oplus A_{34} \oplus A_{35} \oplus A_{36} \oplus A_{37}$
$\texttt{Bank}_1$	$A_{14} \oplus A_{34} \oplus A_{35} \oplus A_{36} \oplus A_{37}$
$\texttt{Row}_0$	A_{21}
$\texttt{Row}_1$	A_{19}
$\texttt{Row}_2$	A_{20}
$\texttt{Row}_3$	A_{24}
$\texttt{Row}_4$	A_{25}
$\texttt{Row}_5$	A_{26}
$\texttt{Row}_6$	A_{27}
$\texttt{Row}_7$	A_{15}
$\texttt{Row}_8$	A_{16}
$\texttt{Row}_9$	A_{17}
$\texttt{Row}_{10}$	A_{22}

Next, we detail boolean formulas in conjunctive normal form for all non-linear bits. We note that the $\texttt{ChannelAddress}$, $\texttt{RankAddress}$, $\texttt{Column}$, and $\texttt{Row}$ bits all share functions, but the order of $\texttt{Row}$ bits 12 and 13 are different.

$(\texttt{ChannelAddress}_{28}, \texttt{RankAddress}_{27}, \texttt{Column}_9) \leftarrow A_{31} \wedge (A_{32} \vee \neg A_{37}) \wedge (\neg A_{32} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{36} \vee A_{37}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{33} \vee \neg A_{37})$

$(\texttt{ChannelAddress}_{29}, \texttt{RankAddress}_{28}, \texttt{Row}_{11}) \leftarrow (A_{31} \vee \neg A_{32}) \wedge (\neg A_{31} \vee A_{32}) \wedge (A_{33} \vee A_{34} \vee A_{35} \vee A_{36} \vee A_{37}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{33} \vee \neg A_{37}) \wedge (A_{31} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{36})$

$(\texttt{ChannelAddress}_{30}, \texttt{RankAddress}_{29}, \texttt{Row}_{13}) \leftarrow (A_{31} \vee A_{33}) \wedge (A_{32} \vee A_{33}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee \neg A_{33}) \wedge (\neg A_{33} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{34} \vee A_{35} \vee A_{36} \vee A_{37})$

$(\texttt{ChannelAddress}_{31}, \texttt{RankAddress}_{30}, \texttt{Row}_{12}) \leftarrow (A_{31} \vee A_{34}) \wedge (A_{32} \vee A_{34}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{33} \vee \neg A_{34}) \wedge (\neg A_{33} \vee A_{34}) \wedge (\neg A_{34} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{33} \vee A_{35} \vee A_{36} \vee A_{37})$

$(\texttt{ChannelAddress}_{32}, \texttt{RankAddress}_{31}, \texttt{Row}_{14}) \leftarrow (A_{31} \vee A_{35}) \wedge (A_{32} \vee A_{35}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{33} \vee A_{34} \vee \neg A_{35}) \wedge (\neg A_{34} \vee A_{35}) \wedge (\neg A_{33} \vee A_{35}) \wedge (\neg A_{35} \vee \neg A_{37}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{33} \vee A_{34} \vee A_{36} \vee A_{37})$

$(\texttt{ChannelAddress}_{33}, \texttt{RankAddress}_{32}, \texttt{Row}_{15}) \leftarrow (A_{31} \vee A_{36}) \wedge (\neg A_{36} \vee \neg A_{37}) \wedge (A_{32} \vee A_{36}) \wedge (\neg A_{31} \vee \neg A_{32} \vee A_{33} \vee A_{34} \vee A_{35} \vee A_{37}) \wedge (\neg A_{35} \vee A_{36}) \wedge (\neg A_{34} \vee A_{36}) \wedge (\neg A_{33} \vee A_{36})$

Appendix B.
Secure Attestation Extended Details

We now describe Intel SGX and TDX attestation in detail, which relies on the Intel Data Center Attestation Primitives (DCAP) libraries. We first describe the entities involved in attestation, how local attestation is done, then how remote attestation quotes are generated and verified.

Attestation Entities and Services. Before outlining the flow of Intel's attestation protocol, we first introduce the entities and services involved in performing attestation. More specifically, Intel provisions keys to CPUs and provides Intel-signed SGX enclaves for attestation. The Intel-signed Provisioning Certification Enclave (PCE) provisions other enclaves with attestation keys, while the Quoting Enclave (QE) ingests local attestation reports to generate remotely attestable quotes. While both SGX and TDX use the same PCE, the QEs differ in that they attest SGX or TDX reports respectively using the corresponding attestation keys.

Next, Intel also runs two services supporting attestation, the Registration Service (RS) for CPU registration and provisioning and the Provisioning Certification Service (PCS) to serve the current TCB information, signed Provisioning Certification Key (PCK) certificates, and the certificate chains to verify PCK certificates. The attesting user application, a SGX enclave or TDX VM, produces reports. The remote verifier uses the PCS to verify these reports as quotes. See Figure 7 for a diagram showing the interaction between these parties, which we now describe.

Key Generation. To generate remotely attestable quotes, the machine first goes through an initialization and registration process. During production, Intel fuses the Root Provisioning Key (RPK) and Root Seal Key (RSK) into the CPU. These keys are used to derive other keys using $\texttt{EGETKEY}$,

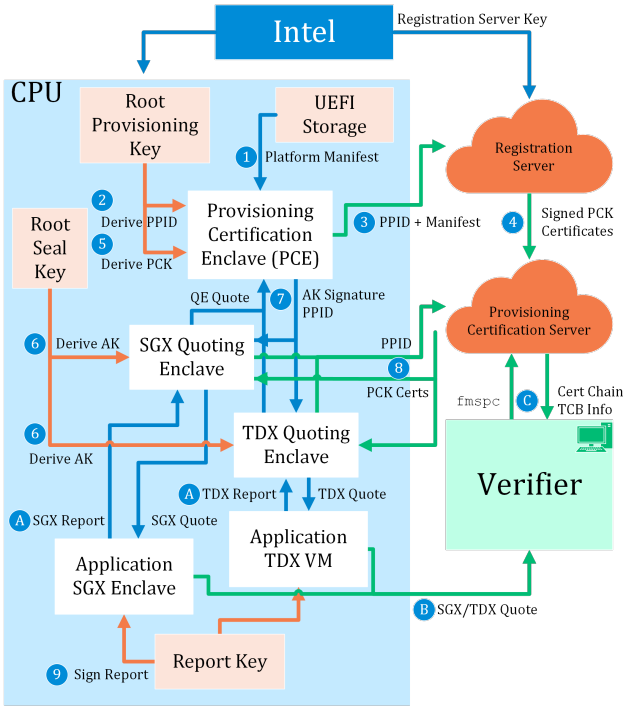


Figure 7: DCAP attestation flow in Intel SGX and TDX. Blue arrows represent the flow of raw data. Orange arrows represent key derivation or signing operations where the actual keys are never transferred. Green arrows represent data transfer over the internet.

and are inaccessible from software. To register the machine with the RS [88], it first retrieves the Platform Manifest from a UEFI variables (① in Figure 7), which contains root keys [89] and is encrypted by the microcode with the public key of Intel's RS. Next, the PCE derives a Platform Provisioning ID (PPID) from the RPK and its `MRSIGNER` measurement ②. The machine sends the Platform Manifest and the PPID to the RS ③ [89], which then confirms the PPID and derives and signs the Provisioning Certification Key (PCK) certificates available through the PCS ④.

The PCE and RS derive the PCK from the RPK, the PCE's `MRSIGNER` and the CPU microcode Software Version Numbers (SVNs) ⑤. Next, the QEs derive an attestation key from the RPK, its `MRSIGNER` and SVN ⑥. The QE generates a report with the QE's key in the report body, which is then verified by the PCE. Next, the PCE uses the PCK to sign the QE's key ⑦. Finally, the QE retrieves the PCK certificate chain from the PCS using the PPID ⑧, and uses the signature and this certificate chain to generate certification data to be included in future quotes. Upon initialization, the PCE has derived an Intel-signed PCK, and the QE has derived PCE-signed attestation keys, creating a signature chain tracing back to Intel's Root of Trust [42].

Local Attestation. Before TEE code can prove (or attest) to a remote verifier, it must first prove its own identity to the Quoting Enclave (QE) through *local attestation*. First, the CPU collects the environment's measurements in a report through the `EREPORT` instruction (SGX) and through the TD module's `TDG.MR.REPORT` instruction, which calls `SEAMREPORT` (TDX). Next, the CPU uses a report key to compute a cryptographic Message Authentication Code (MAC) over the report's contents (⑨ in Figure 7). These can then be verified using `EVERIFYREPORT` (SGX) or `EVERIFYREPORT2` (TDX) [8].

Remote Attestation. To create a remotely attestable quote, the untrusted host passes this report from the enclave or TD to the respective SGX or TDX quoting enclave (Ⓐ in Figure 7). The quote contains the QE's public key, the certification data from the PCE, and the CPU microcode's and QE's SVNs. Next, the TEE code receives the quote and sends it to the remote validator to verify the quote ⑧. The verifier first parses the `fmspc` value to identify the CPU and retrieve the TCB info from the PCS, which includes a set of SVNs, the corresponding mitigation status and Intel's certification chain Ⓒ [88]. For TDX, the PCS also returns information about the current TDX module. Finally, the verifier validates the signature chain in the report, verifying that the QE correctly signed the report, the PCE correctly signed the QE's report, all the way to the root of trust, as well as that the quote's measurements are as expected.

Appendix C.
SECRET Network Extended Details

For completeness, we provide a more detailed overview of our ECDH attack described in Section 9 here. More specifically, we delve deeper into cryptographic details and describe how we obtain required information for the attack.

Extended Cryptographic Details. The ECDH algorithm performs a scalar multiplication on the consensus seed exchange public key as a Montgomery point with the node's private key as the scalar. SECRET's enclave performs this multiplication using the Montgomery Ladder algorithm from `curve25519-dalek 4.0.0-rc.3` as shown in Listing 4. This implementation slides over the scalar bits in big endian using a 2-bit window with every iteration having access to `prev_bit` and `cur_bit`. Furthermore, depending on the value of `choice` = `prev_bit` ⊕ `cur_bit` (Line 4), `conditional_swap()` (Line 5) swaps `x0` and `x1`. Thus, if we can observe whether the state of `x0` is the same or not before and after executing this conditional swap, we can recover the value of `choice` in each iteration, allowing us to recover the original scalar, and thus the private key.

```
1 let mut bits = scalar.bits_le().rev();
2 let mut prev_bit = bits.next().unwrap();
3 for cur_bit in bits {
4   let choice = prev_bit ^ cur_bit;
5   conditional_swap(x0, x1, choice);
6   differential_add_and_double(x0, x1, aff_u);
7   prev_bit = cur_bit;
8 }
```

Listing 4: SECRET's simplified Montgomery Ladder implementation

Recovering the Private Key. With a trace of `choice`'s state, we proceed to reconstruct the scalar bits, as `choice`

$=$ prev_bit $\oplus$ cur_bit. However, note that we need the previous bit in each loop iteration to determine the value of the current bit, but as the Montgomery multiplication skips the MSB, we cannot recover the the scalar's initial bit. Thus, we recover two candidates for the choice bits.

Furthermore, the MSB and the last 3 LSBs are discarded from the 256-bit scalar, allowing us to only recover 252 bits. We simply try all possibilities for the missing 4 bits, yielding $2^4 = 16$ candidate private keys.

To identify the correct private key, we use each candidate key to forge an encrypted consensus seed to pass to init_node. init_node only successfully decrypts our encrypted consensus seed if the candidate private key is correct, as AES-SIV is an authentication (AEAD) scheme.

```
1  neg %eax
2  vmovdqu 0xe0(%rsp),%ymm0
3  vmovdqu 0x60(%rsp),%ymm2
4  vpxor %ymm0,%ymm2,%ymm3
5  vmovd %eax,%xmm4
6  vpbroadcastd %xmm4,%ymm4
7  vpand %ymm4,%ymm3,%ymm3
8  vpxor %ymm0,%ymm3,%ymm0
9  vmovdqu %ymm0,0xe0(%rsp)
```

Listing 5: Disassembly of the conditional swap in the Montgomery Ladder implementation at enclave addresses 0x29bff3 - 0x29c022.

Controlling the Code Execution. As we want to observe the state of x0 before and after the conditional swap, we need to halt the enclave's execution at the right locations. We use objdump to inspect the assembly code of the conditional swap as shown in Listing 5. In particular, conditionally swapping is implemented as ((x0 $\oplus$ x1) $\wedge$ mask) $\oplus$ x0 where mask is expanded from choice to either all zeros if choice is zero, or all ones otherwise. This expansion occurs through a combination of the neg (Line 1) and vpbroadcastd (Line 6) instructions, whereas conditionally swapping is implemented through vpand and vpxor instructions (Lines 4, 7 and 8).

We note that the Montgomery multiplication function itself is located at relative address 0x29b00, and that the loop spans from relative address 0x29bfb0 up to 0x29c91f, crossing a page boundary at 029xc000. Furthermore, 0x29c13d calls another page at address 0x42ea10.

Thus, we can perform a controlled-channel attack [40] by unmapping 0x29b000, executing the enclave until a page fault occurs and then remapping the page to reach the Montgomery multiplication function itself. Then we alternate between two steps until we reach the end of the function. In the first step, we simply target page 0x29c000, which is the second page of the loop, but is conveniently located before the conditional swap to reach the state before the conditional swap. In the second step, we target page 0x42e000 (the target of a function call) to reach the state after the conditional swap. After these two steps we target page 0x29c000 followed by page 0x29b000 to complete one iteration of the loop.

Locating x0. Now that we can control the code execution, we want to infer the state of x0 before and after the conditional swap. We find that x0 is located on the stack at 0xe0(%rsp) and has a size of 40 bytes. However, to perform our attack we need to know the exact stack address of 0xe0(%rsp). To achieve this, after launching the enclave, we read the enclave's memory maps from /proc/PID/maps and parse the stack area.

Appendix D. AUTOMATA

Overview. AUTOMATA is an attestation layer that integrates TEEs like TDX into decentralized systems [90]. They offer attestation SDKs and several fully-audited [91] means to verify TDX quotes on-chain for popular ecosystems such as Ethereum, Polygon, and Arbitrum [92]. AUTOMATA provides *on-chain attestation*, which performs the full verification entirely on-chain via a smart contract and their own Provisioning Certificate Caching Service (PCCS). Since smart contracts cannot interact with the internet, their PCCS stores the collaterals required to verify attestations on-chain.

Because of the compute-constrained nature of blockchains, AUTOMATA also provides *ZK proof attestation*, which verifies zero-knowledge proofs of successful attestation. This offloads the attestation verification and proof generation from the smart contracts, leaving a much lighter proof verification instead. For proof generation, users can choose between two zero-knowledge RISC-V VMs, RISC Zero [93] and SP1 [94]. Both are zero-knowledge virtual machines (zkVM) that provide proof of correct execution of an executed program.

Given that these two implementations are both just different instances of the DCAP quote verification process, they are susceptible to accepting our forged quotes (because our created attestation keys are actually valid). As such, attacking AUTOMATA requires massaging the raw quote into a format that is accepted by the smart contract, and submitting it to the network. Since these quotes pass network verification as valid, they are stored on-chain, thus subverting any projects relying on AUTOMATA as well.

On-Chain Attestation. First, using the techniques from Section 6.3, we forge a valid quote. With libraries provided by AUTOMATA, we then create a Rust program to take our quote, encode it, and submit it to their test network. We initiate on-chain verification, wherein our forged quote successfully verifies.

ZK Proof Attestation. Next, we set up an aforementioned zkVM, SP1, on a local machine. The proof generation process is very resource intensive, so AUTOMATA recommends using the SP1 decentralized proving network. Critically, this means that quote construction is separate from proof generation. We feed in a forged quote, and the SP1 VM verifies the quote, generating a corresponding proof. We use our program from earlier to submit this to the test network where it verifies the proof and accepts our quote.

Consequences. Recall Section 8, wherein vLLM-PROXY recommends the use of AUTOMATA for on-chain quote verification. Our attack demonstrates how it, and other

projects relying on AUTOMATA, lose the guarantees provided by TDX. Other projects relying on AUTOMATA's attestation layer are ElizaOS [95], an AI agent framework; Espresso [96], a cross-chain compatibility protocol; and Puffer's UniFi [97], an Ethereum layer 2 rollup.