Teaching an Old Dog New Tricks: Verifiable FHE Using Commodity Hardware

Jules Drean
MIT CSAIL

Fisher Jepsen
MIT CSAIL

G. Edward Suh
NVIDIA / Cornell University

Srinivas Devadas
MIT CSAIL

Aamer Jaleel
NVIDIA

Gururaj Saileshwar
University of Toronto

Abstract

We present Argos, a simple approach for adding verifiability to fully homomorphic encryption (FHE) schemes using trusted hardware. Traditional approaches to verifiable FHE require expensive cryptographic proofs, which incur an overhead of up to seven orders of magnitude *on top of FHE*, making them impractical.

With Argos, we show that trusted hardware can be securely used to provide verifiability for FHE computations, with minimal overhead relative to the baseline FHE computation. An important contribution of Argos is showing that the major security pitfall associated with trusted hardware, *microarchitectural* side channels, can be completely mitigated by excluding any secrets from the CPU and the memory hierarchy. This is made possible by focusing on building a platform that only enforces program and data *integrity* and *not* confidentiality (which is sufficient for verifiable FHE, since all data remain encrypted at all times). All secrets related to the attestation mechanism are kept in a separate coprocessor (e.g., a TPM)—inaccessible to any software-based attacker. Relying on a discrete TPM typically incurs significant performance overhead, which is why (insecure) software-based TPMs are used in practice. As a second contribution, we show that for FHE applications, the attestation protocol can be adapted to only incur a fixed cost.

Argos requires no dedicated hardware extensions and is supported on commodity processors from 2008 onward. Our prototype implementation introduces 3% overhead for FHE evaluation, and 8% for more complex protocols. In particular, we show that Argos can be used for real-world applications of FHE, such as private information retrieval (PIR) and private set intersection (PSI), where providing verifiability is imperative. By demonstrating how to combine cryptography with trusted hardware, Argos paves the way for widespread deployment of FHE-based protocols beyond the semi-honest setting, without the overhead of cryptographic proofs.

Keywords

Fully homomorphic encryption, trusted execution environment, transient execution attacks, microarchitectural side channels

1 Introduction

Fully homomorphic encryption (FHE) [35, 40, 53, 63] makes it possible to evaluate any logical circuit directly on encrypted data. In the client-server setting, it can be used to build other powerful cryptographic primitives such as private set intersection (PSI) [4, 42, 72, 95, 96], private information retrieval (PIR) [34, 36, 42, 77] or multi-party computation (MPC) [9, 48]. FHE also has many practical use cases including private contact discovery [99], private smart contracts [122], or private inference [102]. Reducing performance overhead (currently 3 to 7 orders of magnitudes over non-private computation) has been the main focus of the last decade of research, but other limitations of FHE are now becoming more relevant.

At the forefront of these issues, the fact that FHE-schemes are not secure when considering a malicious evaluator, or that existing FHE schemes lack notions of integrity, are significant limitations for real-world deployment [10, 31, 131]. First, if a malicious server is able to supply malformed ciphertext for a client to decrypt, it can mount key recovery attacks [38, 69] and break all security (if the attacker recovers the private key, it can now decrypt the client's private requests). Second, if FHE enables delegation of computation on confidential data to an untrusted party, existing constructions cannot help the user *verify* that the correct function was evaluated. In other words, FHE schemes always assume honest-but-curious attackers. In the case of smart contracts, for instance, a malicious evaluator can completely change the functionality of the contract, with the option to even reveal confidential inputs and violate privacy. A straightforward solution is to add a layer of verifiable computation on top of the FHE scheme. Unfortunately, existing solutions suffer from major limitations. Cryptographic proofs incur an overhead of 4 to 6 orders of magnitude on top of FHE [131]. Another solution is to replicate the evaluation across several non-colluding parties, but distributed-trust setups are often impractical in the real world [87].

Trusted execution environments (TEEs) or secure enclaves [39, 46, 110, 121] have also been considered as a potential solution [81, 102, 131]. These technologies implement remote attestation, providing processors the capability to attest the code they are running, even in the presence of a privileged software attacker. TEEs offer better performance than cryptographic proofs, but come with weaker security guarantees: an attacker does not need to break a cryptographic assumption to forge a proof. Instead, the attacker needs to hack the platform or steal the attestation key. Still, these platforms initially seemed to offer a reasonable compromise. Unfortunately, a long series of published attacks (the SoK of Li et al. [85] covers 43 of them) quickly revealed that they were extensively vulnerable to microarchitectural side channels and transient execution attacks [33, 65, 118, 127, 129, 137]. These powerful attacks can be mounted in software and exploit microarchitectural structures and speculative mechanisms present in modern processors to extract secrets from trusted environments, not only violating confidentiality

Proceedings on Privacy Enhancing Technologies 2025(3), 282–303

https://doi.org/10.56553/popets-2025-0099

of the protected programs but also extracting keys used for attestation. New microarchitectural side channels are discovered every year [97, 133] and the current consensus is that it will take many years of research to build software and hardware defense mechanisms that can efficiently eliminate these threats from modern processors. On top of this existential threat, the availability of TEEs is also limited. Modern platforms require dedicated hardware and are only available on server-grade processors, limiting adoption.

This Work. We present Argos, the first enclave platform specifically designed to build maliciously-secure and verifiable FHE. As observed in previous work [81, 102, 125, 131], for use cases where all application data is encrypted, such as in FHE, the only secrets exposed on the server are those used for attestation. We go further and show how this is a unique opportunity to rethink TEE architecture and design *integrity-only* enclaves that focus on data and code integrity *without* offering any confidentiality guarantee. Lessening the requirements for enclaves gives us the opportunity to 1) completely eliminate microarchitectural side channels by secluding all secret key material in a physically-separated coprocessor (such as a trusted platform module or TPM), 2) use a hypervisor-based TEE architecture that offers better performance and hardware compatibility, and 3) build a simplified attestation scheme that makes it possible to easily prove the security of our construction and amortize the cost of relying on a slow coprocessor for key operations.

Eliminating Microarchitectural Side Channels. An important observation we make is that side-channel attacks are "read-only" gadgets and can be eliminated simply by excluding any secrets from the CPU and the memory hierarchy (see Figure 1). Software-mounted side channels require an attacker program to share hardware resources with the victim programs in order to extract secrets. If all secrets are secluded on a separate chip, such as a physical trusted platform module (TPM) or a secure coprocessor [6, 44, 75], no attacker program can ever share microarchitectural resources with the cryptographic algorithm and extract secrets. Leveraging this insight, we design Argos to store all attestation secrets inside a discrete TPM and delegate all sensitive operations to outside of the CPU. This simple principle makes Argos secure *by construction* against side-channels. The only (side) channel left between the secluded cryptographic keys and a potential attacker is completion time [98] which is addressed through using constant-time cryptography in the TPM (see our threat model in Section 2).

Rehabilitating Hypervisor-based TEEs. Argos relies on a hypervisor-based TEE design. A security monitor runs with hypervisor privilege and provides code integrity and isolation to enclave programs. Such TEEs were introduced by TrustVisor [93] and are still widely used by cloud vendors to implement confidential computing [7]. These architectures are usually considered less secure than dedicated hardware platforms such as Intel SGX or AMD-SEV as they do not encrypt main memory and offer no security against Coldboot [70], one of the most practical physical attacks (see Table 7). In our context, this is not an issue as no secrets are ever exposed to main memory. As a result, despite our strong threat model, Argos can adapt this architecture to bring compatibility with most commodity hardware. In terms of performance, existing platforms typically rely on virtual TPMs [115]—one per execution environment—that are endorsed by the root TPM but executed on CPU [12, 14]. While this approach avoids the performance bottleneck of a single physical TPM [93], it makes them vulnerable to microarchitectural side-channels. In contrast, Argos prefers a secure-by-design approach, centered on the discrete TPM.

Performance Optimization With a Discrete Chip. Relying on a single discrete TPM can have a significant impact on performance [94]. Nevertheless, we can still virtualize most TPM resources in the security monitor, trusted code that runs at hypervisor privilege level, as measurement (e.g., hashing) does not require any secret manipulation. Our use case also excludes the use of the TPM "sealed" secret storage. That means we only use the TPM as a "signing oracle". This can still be expensive, especially when considering interactive cryptographic protocols that require repeated and attested message exchanges.

Simple and Efficient Attestation Scheme. We show that for our FHE applications, we can build a simple attestation scheme that only ever requires one TPM signature. Unlike in other enclave platforms, the remote user does not need to see any attestation proof before sending its sensitive data, as it will always stay encrypted. This also applies to intermediate protocol messages that do not need to be attested but simply added to a transcript whose hash is extended in the security monitor. At the end of the FHE computation (or more precisely, before any FHE ciphertext needs to be decrypted), the transcript's hash will be signed using the TPM. We show how our simplified transcript-based attestation can be model as a proof system, and show that it is sufficient to achieve malicious security needed for verifiable FHE. We also show how Argos can be extended to support batched verifiable FHE. As a result, the overhead of using a physical TPM becomes a fixed cost, and performance becomes similar to that of a virtual TPM, while enforcing security-with-abort in the presence of a malicious attacker [64].

Extending Argos to FHE-Based Applications. Circuit-level security sometimes differs from application-level security. In many FHE-based applications, a malicious server can provide corrupted inputs and gain some information from how the client behaves following decryption. This can have devastating downstream security implications. We show how Argos can easily be extended to support complex protocols in the malicious setting, such as authenticated PIR or authenticated PSI with almost no overhead compared to the semi-honest schemes.

Implementation and Evaluation. We implement and evaluate Argos on real hardware[1]. Our prototype runs on Intel x86 platforms, but Argos is easily adaptable to other vendors and architectures (e.g., AMD X86, ARM or RISC-V) and compatible with most commodity machines past 2008. To guarantee the TCB integrity at boot-time, we use commodity hardware root-of-trust technologies. Our prototype uses a TPM, but our architecture is adaptable to other hardware roots of trust and secure coprocessors such as the Apple Secure Enclave [6] or Open Titan [75]. Our security monitor is an open source fork of Tyche [30], modified to support our attestation scheme and remove side-channel protections, but our approach is also compatible with other micro-hypervisors like seL4 [78]. We implement a custom minimal runtime for FHE and the SEAL library [35], emulating system calls and providing randomness through a hardware random number generator (i.e., RDRAND instructions) that is not

[1] https://github.com/mit-enclaves/argos

under OS control. For more complex applications that require a broader class of system calls, Gramine [126] can be used for better compatibility, at the price of a larger TCB and some performance overhead (5× slower startup time, for example). Argos has a limited attack surface with a minimal trusted code base (TCB) of less than 18KLOC, plus the target FHE application (≈ 50KLOC). Our evaluation shows that Argos is 80 times faster than previous work leveraging Intel SGX for FHE integrity [131] with a minimal average performance overhead of 3% for FHE evaluation. We show that Argos can be used to implement more complex protocols such as attested PIR and attested PSI with performance overheads under 8% and without the significant offline communication costs incurred by cryptographic solutions.

Contribution. Argos is the first TEE-based platform to enable maliciously-secure verifiable FHE while being secure against all known microarchitectural side channels, all at minimal overheads. Prior to this work, it was not obvious that TEEs in *commodity hardware* could achieve such strong security guarantees. Existing TEEs suffer from insecurity due to the fact that their remote attestation mechanisms are vulnerable to microarchitectural side channels (see Section 9.7). As a result, TEEs have thus far remained undesirable for cryptographic applications such as verifiable FHE. Argos is carefully designed such that only the TPM contains unencrypted secrets. The TPM uses a simple microarchitecture and is microarchitecturally isolated from the CPU, blocking all known microarchitectural side channels. This means that we can leverage the TPM as a "signing oracle" to design a simple, efficient, and *secure* remote attestation scheme. Furthermore, this simple solution addresses an important problem: the efficient deployment of FHE in real-world scenarios with malicious security.

To summarize, our main contributions are:

- Argos, the first *integrity-only* enclave platform designed to build maliciously-secure verifiable FHE;
- Argos can be used to build fully malicious and authenticated PSI and PIR schemes;
- Argos is secure by construction against microarchitectural side channels and transient execution attacks;
- Argos requires no specialized hardware and is compatible with commodity processors from 2008 onward; and
- Argos only incurs 3% average performance overhead for FHE computation, less than 8% performance overhead for complex protocols, and virtually no communication overhead;

2 Threat model

Our goal is to ensure integrity of the data and computation running inside our enclave environment. Most importantly, we do not protect confidentiality of our programs, nor do we protect them against denial of service. We assume a strong adversary collocated on the server that has compromised the majority of the software stack including the OS, and can mount any microarchitectural side-channel and transient execution attacks. Because Argos is secure against all software-mounted side channels that underlie known transient execution attacks, we will refer to all these attacks as “microarchitectural side channels” throughout the rest of the paper. Our trusted computing base (TCB) consists only of a security monitor (18 KLOC) and our application, both assumed to be bug-free. We assume that the hardware is *functionally* correct and that the TPM’s cryptography is properly implemented with constant-time programming. Rowhammer and fault injection attacks are not as practical and are considered out of scope. We also protect against the majority of physical attacks (cold boot, BIOS tampering, physical side channels) on the main processor and DRAM, but for the rest of this paper, we will focus on software-mounted attacks and consider physical attacks out of scope. A primer on microarchitectural and physical attacks can be found in Appendix B. A detailed discussion on the remaining attack surface can be found in Section 9.7.

3 Background & Motivation

3.1 FHE Schemes And Client-Server Setup

We consider a client-server setup where a server evaluates a logical circuit on some client ciphertext obtained using an FHE scheme. An FHE scheme is usually defined as a tuple of algorithms ($\mathcal{E}$.Gen, $\mathcal{E}$.Enc, $\mathcal{E}$.Eval, $\mathcal{E}$.Dec). The client generates a key pair using $\mathcal{E}$.Gen and encrypts some input x using $\mathcal{E}$.Enc. The server then uses $\mathcal{E}$.Eval to evaluate a circuit f on the ciphertext c_x and sends the resulting ciphertext c_y back to the client to decrypt using $\mathcal{E}$.Dec. Concretely:

- $\mathcal{E}.\text{Gen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$
- $\mathcal{E}.\text{Enc}_{key}(x) \rightarrow c_x$
- $\mathcal{E}.\text{Eval}_{pk}(c_x, f) \rightarrow c_y$ where $y = f(x)$
- $\mathcal{E}.\text{Dec}_{sk}(c_y) \rightarrow y$

FHE schemes must enforce the following properties (see Appendix C.1 for formal definitions):

Correctness. A scheme is *correct* if any honest computation will decrypt to the expected result (i.e., $Dec_{sk}(c_y) = f(x)$ with high probability).

Security. All FHE schemes are secure against chosen plaintext attacks, which means that an attacker with the public key cannot distinguish between the encryption of two different messages of its choice.

3.2 Semi-Honest vs. Malicious Security

In a honest-but-curious or semi-honest setting, an attacker is allowed to observe intermediate messages and passively infer information in order to break security, but cannot deviate from the agreed-upon protocol. In a malicious setting, the attacker is much more powerful and allowed to deviate arbitrarily from the protocol, e.g., to lie to the victim or tamper with messages. The semi-honest setting is often used in theory, but has limited viability in a deployment context where assuming a malicious attacker is more realistic.

3.3 Why Does FHE Need Verifiability?

Most state-of-the-art FHE schemes are *insecure* to use in real-world settings [35, 40, 53, 63] where malicious security is generally required. Because these schemes are not CCA-secure [18], if an attacker is able to supply malformed ciphertexts and observe decryption outcomes (for example, by observing client behavior), it can mount key recovery attacks and completely compromise privacy [32, 38, 57, 69]. Beyond the “circuit-level” security, the absence of integrity for FHE schemes also has implications for FHE-based applications. A malicious server can arbitrarily modify the circuit

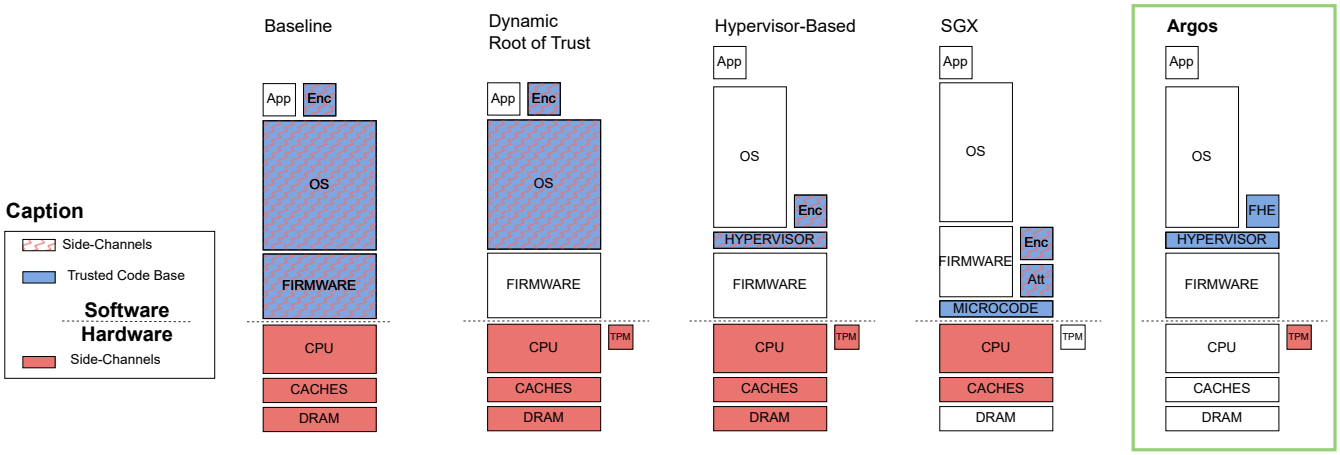

Figure 1: Evolution of the attack surface on TEE platforms. Enc: enclave program, Att: attestation enclave.

they evaluate on the encrypted data. Attacks on correctness are an obvious issue, but these can also translate to attacks on privacy at the application level. Let us take the example of private contact discovery. Alice wants to discover who among her contacts is using a private messaging service [99]. Alice encrypts her address book, which contains all of her contacts' private information, and sends it to the server. The server now has access to Alice's encrypted set of contacts. If malicious, it could simply return the entire set to Alice. This could mislead Alice into thinking that one of her friends (Bob) uses the messaging service. Consequently, Alice contacts Bob via the service, leaking his phone number. This would violate Bob's privacy and defeat the use of private contact discovery.

3.4 Why Do We Need a New TEE Platform?

Trusted execution environments (TEEs) provide hardware-based isolation for confidential programs while minimizing the trusted code base (see Figure 1). The existing TEE landscape spans from platforms that protect small "enclave" programs [11, 46, 110] to those that implement confidential computing for entire virtual machines [7, 39, 86, 121]. However, the discovery of numerous microarchitectural side channel and transient execution attacks has severely compromised their confidentiality guarantees. Side channel attacks exploit shared microarchitectural structures such as memory caches [25, 65], translation look-aside buffers (TLB) [66], branch predictors [1, 56], and DRAM controllers [132] to enable information leakage between security domains.

TEEs also provide integrity guarantees through remote attestation, allowing a remote client to verify the authenticity of the hardware and the initial state of the execution environment. However, even these attestation mechanisms have proven vulnerable to side-channel attacks that can extract secret keys used by platforms to sign attestation reports [33, 118, 129], leading the cryptographic community to lose faith in TEEs. On the other hand, discrete trusted platform modules (TPMs) and associated dynamic root of trust (DRoT) have long served as dedicated hardware for platform integrity and secure storage against software-based attackers. However, their discrete nature also creates a severe performance bottleneck when attesting multiple security domains (Flicker [94] incurs 2-3 orders of magnitude overhead). Although TPM virtualization [12, 14, 93, 115] can address these performance issues, in turn, they expose sensitive key material to microarchitectural side channels (see Section 10 for a more detailed comparison of Argos and other platforms).

To rebuild trust in hardware security primitives, we must develop new systems that are both inherently secure against microarchitectural side channels and capable of outperforming functionally equivalent cryptographic solutions. We address this challenge by focusing specifically on FHE applications and building a TEE platform that *only enforces integrity* of the enclave program, which is sufficient for verifiable FHE since data remains end-to-end encrypted throughout the computation.

4 Insights & Design Principles

4.1 FHE Applications Do Not Expose Secrets

In FHE applications, no sensitive data is ever manipulated in clear. This has several implications that can help simplify the design of a TEE platform and the attestation protocol.

No Secrets Exposed. In a standard TEE platform, a central feature is to enforce the confidentiality of the execution environment. In FHE, all data is encrypted. As a result, the contents of our enclave program (i.e., the state of $\mathcal{E}$.Eval) is public. This means our application is not vulnerable to side channels, and we do not need to harden the execution environment against the usual threats.

No Long-Term Secret Storage. In typical TEE platforms, an important feature is the ability for an enclave program to *seal* secrets and recover them later. This property is usually tied to the identity (or the measurement) of the enclave program, and a secret can only be unsealed by a program that matches the correct measurement. This is, for instance, what is used in Bitlocker to only ever release the disk encryption key to a correctly booted system. TPMs and TEE platforms such as SGX all offer a seal operation. FHE evaluation does not require long-term secret storage, which significantly reduces the features required by the TEE.

No Need To Pre-Establish Trust Before Sending Inputs. In usual remote attestation protocols, the remote client first needs to verify the TEE attestation to establish trust. Only once it trusts the TEE to have been correctly setup will it send its encrypted private input. This is because the inputs will then be decrypted inside of the TEE. In our case, inputs are never decrypted, which means that the remote client does not need to establish trust before sending the encrypted input. This makes it possible to amortize the cost of attestation by only signing the final transcript of program inputs and outputs (see Section 5).

4.2 Eliminating Microarchitectural Side Channels Using A Physical TPM

Because our FHE-applications do not expose secrets, the only secret ever present on the server is the private signing key used in the attestation scheme. This is a much smaller attack surface than in usual TEE systems. We can take advantage of that opportunity to completely eliminate microarchitectural side channels by placing the signing key in a microarchitecturally-isolated physical TPM. Because no secrets are left on the CPU or in main memory, this effectively makes our platform completely secure against microarchitectural side channels.

Side Channels Are Read-Only. One important element to keep in mind is that side channels are "read-only" gadgets and can only extract secrets from a victim program. Specifically, a side channel cannot modify the state of a victim's program memory or change its control flow.

Side Channels Require Shared Resources. To mount a microarchitectural side channel, an attacker needs to trigger a transmitter that will access the secret and modulate a channel to transmit information to a receiver under the attacker's control. That means the transmitter and receiver programs need to share some microarchitectural state in order for the transmission to be possible.

Physical TPMs Are Microarchitecturally Isolated. TPMs are commonly implemented on a discrete chip or as firmware on a secure co-processor (we cannot secure software TPM running on the CPU). They communicate with the main CPU using a hardware bus. They do not share cache, nor any processor resources, with the CPU. They use their own private resources, like private memory and registers, that are only accessible to the TPM. As a result, if the secret key is kept in a physical TPM, it is microarchitecturally-isolated from the attacker and there is no possibility for the secret to be extracted through a CPU microarchitectural side channel. In addition, the microarchitecture of TPMs and secure co-processors is intentionally kept simple, avoiding features like caches or out-of-order execution. This prevents the existence of indirect microarchitectural side channels such as NetSpectre [120]. Intrusive physical attacks might still be possible but are outside of our threat model (see Section 9.7).

4.3 System Overview

Our system is composed of several hardware and software elements (see Figure 1). The security monitor is a small (≈18 KLOC) trusted piece of code running at the hypervisor privilege level. It is in charge of enforcing isolation between the different security domains and integrity of the enclave programs. Isolation is enforced by hardware mechanisms (see Section 8.2) and integrity is enforced through our attestation scheme (Section 5). The security monitor does not manipulate any secret key material and delegates all cryptographic operations that require a key to the physical TPM. Integrity of the security monitor is enforced by the dynamic root-of-trust mechanism (Section 8.1). Running on top of the security monitor, a large untrusted operating system is in charge of all resource allocations. It contains all the complex logic, for instance, for memory allocation or scheduling. It also provides many services for (enclave) applications such as networking, or I/O.

Running an Enclave Program. To setup a trusted execution environment and securely run an enclave program, the untrusted OS interacts with the security monitor through vmcalls to the hypervisor. On behalf of the OS, the security monitor will reclaim some resources (memory and cores) and allocate them to the enclave. Thanks to the isolation mechanisms under the control of the security monitor, the OS is now unable to modify the state of the enclave memory. The security monitor will also enforce some invariants (setting the page table correctly and zeroing regions where no content is loaded), ensure the correct binary is loaded, and attest the enclave initial state (see Section 5.4). Once loaded, the enclave can be started. The remote client sends inputs over the network. They are received by the OS and placed in shared memory so that the enclave can access them. Inputs are systematically copied to enclave's private memory before being used. Any output is also copied to shared memory and sent back to the client by the OS.

5 The Argos Attestation Scheme

5.1 Existing Attestation Protocols

Attestation schemes are generally interactive protocols that involve several rounds of communication between a client and a server. First, before any sensitive data is sent, trust needs to be established between the execution environment running on the server side and the remote client. The client will send a challenge and receive an attestation report, endorsed by the manufacturer and attesting that a genuine piece of hardware has been correctly set up to instantiate a trusted execution environment. Once trust has been established, the enclave program and the remote client can perform key exchange and establish a trusted communication channel. They can then start communicating and running the expected application. Any message coming out of the enclave needs to be encrypted and attested using the private key generated inside of the enclave to prevent a malicious attacker from observing or tampering with the content of the messages. Similarly, every message from the remote client needs to be decrypted and its integrity verified.

5.2 Overhead of Using a Discrete TPM

One of Argos's design principles dictates that no secret key material should ever be present on the main CPU. This means that each enclave would need to delegate all key operations to a single physical TPM. Even if only considering integrity protection (signature of messages and attestation protocol), this implies that the TPM becomes a serious bottleneck for the platform. Performing a signature using a discrete TPM takes approximately 196 ms, and the numbers are similar for integrated TPMs. This can potentially be a significant overhead for any application that requires several rounds

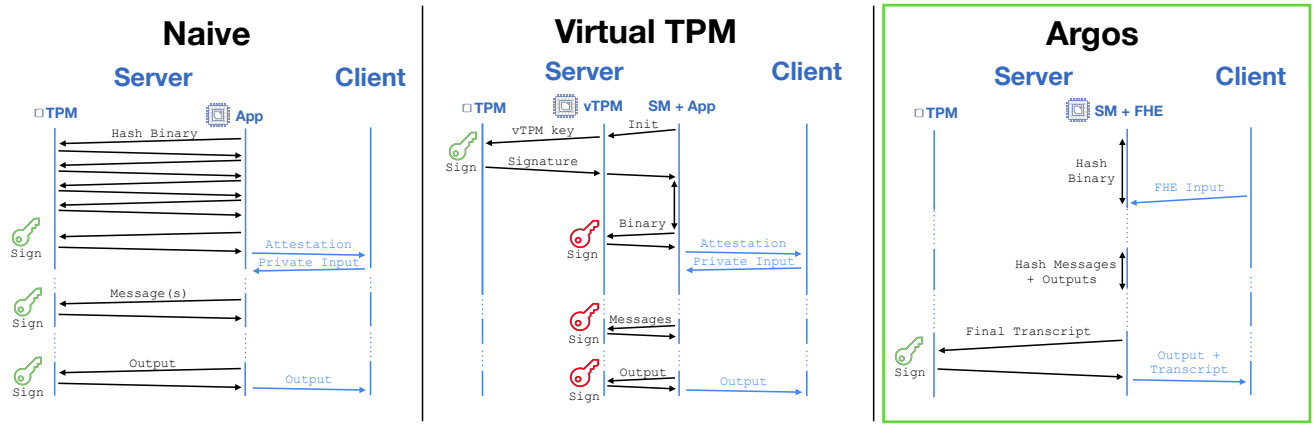

Figure 2: Evolution of remote attestation protocols. The naive approach with a secure co-processor is inefficient as it requires several back and forth communications with the discrete chip. The virtual TPM approach is insecure as it manipulates sensitive keys in the CPU, exposing them to microarchitectural side channels. The Argos protocol is both secure and efficient.

of communication. However, Argos's focus on FHE applications allows us to minimize much of this overhead.

5.3 Attested Transcript for FHE Applications

Trust does not need to be established before sending encrypted data. Because intermediate results are also encrypted, privacy cannot be compromised until a ciphertext is decrypted. That means integrity only needs to be checked *just before* decrypting an output (see Section 6 for our security analysis). In other words, each message of the FHE application does not need to be individually attested. Instead, the integrity-protected security monitor can keep a transcript of all relevant data (i.e., all application inputs, intermediate results, and final output), and, only perform a single TPM signature over that transcript. This simplifies the attestation scheme and drastically reduces the cost of using a physical TPM as remote attestation is now a simple fixed cost (see Figure 2).

5.4 Argos Attestation Scheme

Measured Boot. Upon boot, the hardware dynamic root of trust (DRoT) is used to derive a measurement of the Argos security monitor (see Section 8.1). This measurement is securely stored in the TPM registers (PCRs).

Enclave Setup. When a new enclave is initialized, the security monitor will allocate some of its private memory to the enclave transcript. During enclave setup, the security monitor measures the enclave binary and stores it in the transcript. In our specific case, the binary contains the FHE circuit, small public inputs (e.g., FHE parameters), alongside the logic to evaluate the circuit (the SEAL library in our case). It also contains the rest of the application logic. All the values of the relevant architectural registers that represent the initial state of the execution environment are also added to this measurement. For instance, the enclave initial program counter, page tables, the stack pointer and the addresses of shared memory used to communicate with the OS. The measurement of the binary and initial state unambiguously represent a function that takes inputs from shared memory and outputs results to it.

Enclave Lifetime. During the lifetime of the enclave, the transcript is appended with hashes of the data received and sent by the enclave. This includes all inputs and outputs.

Attesting the Transcript. Once the final output has been generated and the transcript extended with it, the application performs a special vmcall to the security monitor to request the attested transcript. The security monitor then leverages the TPM to obtain a signature over the transcript. The application will then be able to transmit the attested transcript to the remote client, along with the final output.

Client Verification. Upon receiving the result(s) and the attested transcript, the client first verifies the attestation. The verification of the attestation goes as follows: First, the client verifies that the key used to sign the transcript belongs to a genuine TPM (see Section A for more details). It then checks that the measurement for the security monitor contained in the transcript signature matches a reference measurement that is known to be correct (see Section 9.7 on how such a reference measurement can be obtained). Finally, it checks that the binary reference in the transcript corresponds to the correct FHE application and that it was evaluated on the input it provided. If all checks are satisfied, the verification is successful. In case of a verification failure, the server is behaving maliciously and the client aborts.

Output Decryption. If and only if verification succeeds, the client decrypts the FHE output.

5.5 Remote Attestation as a Proof System

Our simplified attestation scheme is now functionally equivalent to a proof system. We can formalize it as a tuple (Π.Gen, Π.Prove, Π.Verify). The server runs Π.Gen to generate a key pair. It can then use the private key to run Π.Prove and generate an attested transcript (π_y) over some client input x, an application g (formally a function, in practice a binary and an initial state for the execution environment), and the resulting output y. Given the public key, the client can then run Π.Verify to verify that the attested transcript π_y is correct, that is, $y = g(x)$. In summary, we have:

- $\Pi.\text{Gen}(1^\lambda) \rightarrow (\text{pk}, \text{sk})$
- $\Pi.\text{Prove}_{\text{sk}}(x, g) \rightarrow (y, \pi_y)$ where $y = g(x)$
- $\Pi.\text{Verify}_{\text{pk}}(y, x, \pi_y) \rightarrow$ `true/false` where the client accept the output is `true`

Additionally, our scheme satisfies these two properties (see Appendix C.2 for formal definitions):

Completeness. The system is *complete* if Π.Verify will always accept a honestly computed result, i.e., a correctly formed transcript where $y = g(x)$.

Soundness. The system is *sound* if an adversary cannot make Π.Verify accept an incorrect answer i.e., a transcript where $y \neq g(x)$.

Modeling Several Rounds of Communication. For applications that require several rounds of communication, we only generate one attested transcript at the end (see Section 5.3). That means x is the concatenation of all client inputs/messages. Our specific implementation simply shares intermediate results (the server messages) with the client. It also caches some internal state for Π.Prove (such as the measurement of g) so it can efficiently generate the final attested transcript once provided with all client inputs.

Security Analysis. Completeness of our attestation scheme reduces to the correctness of the underlying signature scheme used in the TPM. Soundness is more complex to establish, as it requires informally laying down all of our system security. First, as we explained earlier, an attacker (as described in our threat model) is not able to extract the private key from the TPM. Second, the security of our DRoT guarantees that the security monitor was correctly loaded and its integrity cannot be compromised. Third, our assumption of the TCB tells us that the security monitor is correctly implemented and bug-free. As a result, attested transcripts will only be generated for enclaves that are correctly set up, loaded, and executed. Our assumption on the underlying hardware isolation mechanisms (execution modes and nested page tables) guarantees that the integrity of the enclave code and data is maintained over the course of its lifetime. Finally, the functional correctness of the hardware guarantees that when an enclave is run that implements functionality f on input x, the resulting output $y = f(x)$. With all of this in place, the soundness of our attestation scheme reduces to the soundness of the TPM signature scheme.

6 Circuit-Level Verifiable FHE

We first show how, using Argos, we can take a semi-honest FHE scheme and build a circuit-level verifiable FHE scheme that achieves malicious security. In Section 7, we show how we can extend Argos to satisfy application-level security.

In our client-server setup, all state-of-the-art FHE schemes [35, 40, 53] are *insecure* when considering a malicious server. Let us take a simple example where a client wants to outsource some sensitive computation to a server using FHE. Here, the server might be able to infer information regarding the decryption of a ciphertext by observing the client reaction. For instance, in FHE, if the noise in the ciphertext overflows a given threshold, client-side decryption might fail, which could lead to the client emitting a new request to the server. This is not a problem if assuming an honest-but-curious server. However, if the server is malicious, it can tweak the response to the client in arbitrary ways and observe its reaction. This is equivalent to providing our attacker with access to a (limited) decryption oracle that outputs "success" or "failure" for ciphertexts under the attacker's control. This is already enough to mount key-recovery attacks [38, 69] and break security of the FHE scheme (if the attacker recovers the private key, it can now decrypt the client's private requests). In fact, this is much more problematic than a simple "correctness" issue where the client would just get a wrong output. In this simple outsourcing scenario, enforcing integrity at the *FHE-circuit-level* is enough, as it prevents the attacker from providing malformed ciphertexts to be decoded. When the client receives the response from the Argos server, it will first verify the attested transcript to ensure that the correct circuit was evaluated on the correct inputs. If verification fails, the client aborts, does *not* decrypt the ciphertext, and detects malicious behavior from the server. If and only if verification succeeds, the client may decrypt the ciphertext with the guarantee that it was correctly formed.

6.1 Definition

Inspired by the formalism of Viand et. al. [131], we define circuit-level verifiable FHE (vFHE) as follow. A vFHE scheme is defined as a tuple of algorithms (Gen, Enc, Eval, Verify, Dec). The client is equipped with a key pair $(\text{pk}_c, \text{sk}_c)$ and the server with $(\text{pk}_s, \text{sk}_s)$. The client uses Enc to encrypt some input x and obtains a ciphertext c_x. Using Eval, the server evaluates a circuit f over c_x and generates an encrypted output c_y, along with an attested transcript π_y. When the client receives the output, it will first *verify* the transcript is correct ($c_y = f(c_x)$) and if and only if it is, use Dec to decrypt c_y and recover y. In summary:

- $\text{Gen}(1^\lambda) \rightarrow ((\text{pk}_c, \text{sk}_c), (\text{pk}_s, \text{sk}_s))$
- $\text{Enc}_{\text{pk}_c}(x) \rightarrow c_x$
- $\text{Eval}_{\text{pk}_c, \text{sk}_s}(c_x, f) \rightarrow (c_y, \pi_y)$ where $y = f(x)$
- $\text{Verify}_{\text{pk}_s}(c_y, c_x, \pi_y) \rightarrow\!\!\rightarrow$ `true/false` where the client accept if `true`, aborts otherwise.
- $\text{Dec}_{\text{sk}_c}(c_y) \rightarrow y$

Formal properties can be found in Appendix C.3. Correctness is defined identically to a semi-honest FHE scheme (see Section 3.1). Security differs as our attacker is now malicious and can send arbitrary c_y values. We also add two properties compared to the vanilla FHE-scheme.

Completeness. A vFHE scheme is *complete* if Verify will always accept an honestly computed result.

Soundness. Finally, a scheme is *sound* if an adversary cannot make Verify accept an incorrect answer.

6.2 Putting Everything Together

Let's consider an FHE scheme ($\mathcal{E}$.Gen, $\mathcal{E}$.Enc, $\mathcal{E}$.Eval, $\mathcal{E}$.Dec) and our remote attestation scheme (Π.Gen, Π.Prove, Π.Verify). Argos' construction is as follows:

- $\text{Gen}(1^\lambda) \rightarrow ((\text{pk}_\mathcal{E}, \text{sk}_\mathcal{E}), (\text{pk}_\Pi, \text{sk}_\Pi))$ with $(\text{pk}_\mathcal{E}, \text{sk}_\mathcal{E}) = \mathcal{E}.\text{Gen}(1^\lambda)$ and $(\text{pk}_\Pi, \text{sk}_\Pi) = \Pi.\text{Gen}(1^\lambda)$
- $\text{Enc}(x) \rightarrow c_x$ with $c_x = \mathcal{E}.\text{Enc}(x)$
- $\text{Eval}(c_x, f) \rightarrow (c_y, \pi_y)$ with $(c_y, \pi_y) = \Pi.\text{Prove}(c_x, \mathcal{E}.\text{Eval}(\,.\,, f))$
- $\text{Verify}(c_y, c_x, \pi_y) \rightarrow b$ with $b = \Pi.\text{Verify}(c_y, c_x, \pi_y)$.

- $\text{Dec}(c_y) \rightarrow y$ with $y = \mathcal{E}.\text{Dec}(c_y)$

The key here is to use the attestation mechanism to attest the correct execution of $\mathcal{E}.\text{Eval}(\,.\,, f)$ represented by the binary and initial state of the execution environment.

Security Proof — Sketch. Correctness of our construction reduces to the correctness of the initial FHE scheme. Similarly, completeness and soundness directly derive from the completeness and soundness of the attestation scheme. Security is more subtle. Remember that our initial FHE scheme is not secure if provided with malformed ciphertexts. That means we can only rely on the security of the underlying FHE scheme if our attacker cannot gain control over the ciphertexts. This is enforced by the verification step. If the client runs Dec on an output received from the server, that means the output and the transcript had to successfully pass Verify. If this is true and the attacker was able to supply a ciphertext of its choice, i.e., $c_y \neq \mathcal{E}.\text{Eval}(c_x, f)$, that means the attacker was able to break the soundness of the underlying attestation system, which is a contradiction. As a result, our construction is secure, and we have successfully built a circuit-level verifiable FHE scheme. The full proof can be found in Appendix D.

Batching FHE Evaluations. Running Π.Prove is expensive (it requires a TPM sign operation). Because evaluations of different FHE circuits are independent, we can batch them for performance optimization and only produce one attested transcript for all these evaluations. Enc, Verify, and Dec are the same, and security still holds as long as Dec is never run if verification was not successful.

7 Extending Argos to FHE-Based Applications

Circuit-level security sometimes differs from application-level security. For instance, attacks on correctness can escalate to attacks on privacy (Section 3.3). Additionally, the security of our FHE schemes breaks if a malicious server is able to supply malformed ciphertexts for decryption (Section 3.3). This is potentially dangerous when, in many FHE-applications, the server itself can provide public or private inputs and act as a party in the computation. We explore two such applications, 1) private information retrieval, and 2) private set intersection. For each application, we discuss the challenges in deploying FHE-based schemes in a real-world setting and show how Argos can be extended to enforce malicious security.

7.1 Malicious Private Information Retrieval

Private information retrieval (PIR) allows a client to retrieve an element from a public database hosted on a remote server, without the server learning which element was requested by the client. PIR has many applications such as public-key directory [41], certificate transparency logs [72, 119], private web search [71] or private analytics [67]. PIR can be instantiated in a multi-server or single-server setting using a range of cryptographic primitives, but in this paper we focus on the state-of-the-art for single-server PIR which are FHE-based schemes [4, 72, 95, 96].

Most schemes assume a honest-but-curious server; however, this is not a realistic setup for real-world deployment. Furthermore, circuit-level integrity alone is not enough to enforce malicious security. A malicious server can always modify the values in the database, which could break not only the correctness (send the wrong answer for a given query), but most importantly security (i.e., privacy of client input) by mounting *selective failure attacks* [82]. We show how Argos can be extended to secure semi-honest FHE-based PIR scheme in a malicious setting.

Defending Against Selective Failure Attacks. In these attacks, a malicious server chooses an entry in the database and replaces the entry with a value that it knows will cause a decryption failure. This is possible because in FHE, using an input that is not in the correct plaintext space (for instance, an integer that is too large) can lead to a malformed ciphertext, which will not decrypt properly. Because our attacker might be able to observe the client reaction to decrypting the output (e.g., the client will send the request again), if decryption fails, the attacker can infer with high probability that the client queried the malformed entry.

Circuit integrity is not enough here. Argos also needs to preprocess the public database and check that the server's public inputs belong to the plaintext space. If a check fails, Argos should abort the server execution and not serve any client request. Formally, this is equivalent to allowing server inputs w for f, and in our construction, wrapping $\mathcal{E}.\text{Eval}$ in a function that evaluates $\mathcal{E}.\text{Eval}$ if and only if w is well formed and aborts otherwise.

Authenticated PIR. So far, our malicious PIR scheme is only able to guarantee that the server performed the protocol using a well-formed database, but not a specific one. Adding authenticity to a PIR scheme is a desirable property and might be essential for some real-world deployments. For instance, in the case of a transparency log or a public-key directory, it might be essential to guarantee the correct database was queried. We take inspiration from previous work and solve this problem by having the server initially commit to the database. This assumes a reference commitment is available for the client to fetch out-of-band.[2] Argos can easily be extended to support this functionality by adding a fresh commitment to the database (i.e., a hash or the root of a Merkle tree) to the attestation transcript. When receiving the response to its query, a client should first verify the attestation transcript, including that the database commitment matches the reference one. If the verification fails, it should abort. If (and only if) it succeeds, it should then decrypt the response and proceed. In our formalism, this is equivalent to extending Verify to take a reference commitment $H(w)$ as an input, and tweaking soundness to ensure an adversary cannot make Verify accept a transcript π_y that does not contain the correct commitment. A detailed security *argument* can be found in Appendix E.

7.2 Malicious Private Set Intersection

Private set intersection (PSI) allows two distrusting parties with some private sets to compute the intersection of these sets without learning more information than the intersection itself. PSI has many applications, such as private contact discovery [99] or compromised credential checking [106].

Most efficient PSI protocols do not achieve (full) malicious security [34, 36, 42, 77]. Some constructions do [50, 103, 109, 114, 116,

[2] This assumption is made by Authenticated PIR [41] but recent works [49, 52] have also shown how using a limited number of "validation queries", the client can verify with high probability that the committed database is correct.

117, 123] but at the price of high offline communication cost (see Section 9.5). Using a state-of-the-art semi-honest PSI [42] scheme based on FHE, we show how Argos can be extended to make the scheme fully secure against a malicious server. Here, the main difference from the PIR setup is that the server inputs are not public, but private. State-of-the-art FHE schemes are not *circuit private*. That means that the client might be able to learn information regarding the server's private input by looking at the noise contained in the output. FHE-based PSI schemes mask the server's input using oblivious pseudo random functions (OPRFs) to provide privacy for the server even in the presence of malicious clients. Moreover, the security of the FHE scheme is argued to also provide privacy for the client, even in the presence of a "malicious" server. This is because the ideal PSI functionality considered in these works assumes a limited attacker with no observability of the client reactions [34]. However, this might not hold in a real-world deployment, as we explained before.

Verifiable PSI. Once deployed in a real-world setting, the attacker might be able to observe some of the client's reactions. Attacks on correctness might have serious security implications, sometimes translating to attacks on privacy (see example of private contact discovery in Section 3.3). Fortunately, circuit-level integrity—as provided by Argos—can guarantee *verifiable* PSI, which means that verification of the attestation transcript succeeds if and only if the server correctly executed the PSI circuit on a private set. Nevertheless, for full malicious security, we still need to ensure that the server evaluates PSI over the *correct* set.

Authenticated PSI. If the server's input set is not authenticated, a malicious server can arbitrarily modify the dataset without detection. This can have serious security implications in deployment. For instance, in the case of compromised credential checking, Alice reaches out to Mozilla's server to ensure that her passwords were not compromised and are safe to use. If Alice indeed uses some compromised credential and the server is malicious, it could evaluate the PSI circuit on her private set, but using a malicious input set, crafted server side, ensuring no intersection is found. This might lead Alice to use her compromised credentials.

Similarly to Authenticated PIR, Argos can be extended to include a commitment to the dataset (e.g., a Merkle tree or hash-based commitment scheme) in the attestation transcript. Here too, a public commitment needs to be made available out-of-band to the client, through a transparency log, for instance. Upon receiving a response to its request, the client first verifies the transcript and aborts if it does not match the correct commitment. Formally, the extensions we made for PIR are almost sufficient, we just need to tweak the wrapper function over $\mathcal{E}$.Eval to match our PSI functionality. Instead of only checking that w is well formed, the wrapping function should compute the corresponding OPRF values for w. The server's privacy is guaranteed by the OPRF, the soundness of the attestation scheme, and the fact that commitment schemes are hiding i.e., they do not leak information about the server's set. The security *argument* can be found in Appendix F.

8 Implementation Details

8.1 Security Monitor

Our security monitor is based on the X86 version of Tyche [30], a Rust micro-hypervisor that exposes hardware resources and isolation primitives to better protect trust domains. Note that other "small" hypervisors such as XEN [17] are at least one order of magnitude larger or closed source. We implement our attestation scheme on top of Tyche, including support for the hardware TPM and the interface to manage attested transcripts. We are also able to simplify some aspects of the platform, for instance, by removing side channel protections from Tyche (e.g., cache and CPU flush). One of the implementation challenges was to adapt Tyche and the Linux driver to support large (i.e., several GBs) enclaves with our custom runtime and manage memory allocation accordingly. To enforce the integrity of the security monitor, we use the TPM and a dynamic root of trust (e.g., Intel TXT or AMD DRMT [60, 104]) to implement standard measure boot (see primer in Section A). At the end of this process, two TPM registers (PCRs) are set with hash values that uniquely identify the security monitor binary. Later, when the security monitor needs to attest an application transcript, it will load the transcript hash in a third PCR and request a TPM Quote (a signature over select PCRs).

8.2 Memory Isolation Primitives

We use hardware extensions for virtualization to enforce isolation between security domains [128]. Extended or nested page tables [20], as implemented by Intel VT-x or AMD-V, make it possible for a guest domain to have an entire address space available and create a level of indirection for memory management that is exclusively under the control of the hypervisor. That means no execution domain can access other execution domains' address space nor modify its content. Other virtualization technologies such as AMD-Vi and Intel VT-d make it possible to protect domains from arbitrary DMA accesses. We use these technologies in Argos to enforce memory isolation for our enclave programs.

8.3 Custom Runtime

The execution environments provided by Argos are similar to bare-metal environments with no OS support. That means no syscalls, no program loader, no memory allocation, nor access to shared libraries. A runtime is required to support programs and modern libraries like SEAL that leverage many services from the OS. Gramine [126] is such a runtime and is already supported by Tyche. It was conceived to securely run off-the-shelf applications on Intel SGX. As a result, it implements all Linux syscalls and also has extensive support for shared library and memory management. However, because of its versatility, it is rather large (20KLOC) and adds some non-negligible overhead [112]. In order to reduce the TCB and improve performance, we implement our own custom runtime. It is minimal (870LOC) and consists of a simple memory allocator [108] and a handful of syscall handlers required by our application. For instance, we implement our own `open` and `read` syscalls to supply randomness from `RDRAND` when an application attempts to read `/dev/urandom`. We write our runtime to interface with MUSL [101] by interposing on system calls and providing our own handlers.

Component	LOC
BIOS	1.5M
Linux	28M
Security Monitor	18K
Runtime (Custom / Gramine)	870 / 20K
SEAL Library [35]	20K
Application	1K—20K

Table 1: Software Components and TCB Breakdown. Red columns are excluded from the TCB.

Platform	Time
Software	42
vTPM	136
dTPM	195752

Table 2: Signing operation (μs).

	Setup	Attest	Term
SGX2	953	7	274
Nitro	3759	1	363
Argos+G	466	196	5
Argos	85	196	10

Table 3: Fixed-cost operations on a 1GB enclave (ms).

That means all applications and libraries (including SEAL [35]) need to be compiled statically and linked using our MUSL library, which requires some engineering effort to port applications but makes it possible to obtain high performance *and* a very light runtime.

9 Evaluation

9.1 Testbench

We prototype Argos on a local Dell Optiplex machine from 2017 equipped with an Intel Core i7-7700 processor clocked at 3.6GHz, a discrete TPM 2.0, and 8GB of RAM. We run our security monitor (adapted from Tyche [30] see Section 8.1) directly on the hardware and evaluate our benchmarks using our custom SEAL runtime or Gramine [126]. We run Ubuntu 22.04 with the Linux kernel v6.2 and SEAL 4.1 with the BFV scheme [23]. All benchmarks are single-threaded, compiled using clang++ with the O3 optimization level.

We compare Argos with two other TEE platforms that have the following specs. SGXv2 running on an Azure server equipped with an Intel Xeon Platinum 8370C CPU with a frequency of 2.80GHz. AWS Nitro Enclaves [11] running on an AWS server equipped with an Intel Xeon Platinum 8259CL CPU with a frequency of 2.50GHz. The SGXv2 setup uses Gramine as a runtime while Nitro Enclave uses a docker container environment that runs its own Linux kernel. Note that under our threat model, both SGXv2 and AWS Nitro Enclaves are *insecure* and are only used as comparison points for performance. See Table 7 for a more complete comparison. All numbers are obtained by averaging values over 10 runs.

9.2 TCB Evaluation

Detailed count for lines of code (LOCs) can be found in Table 1. Our TCB is small with a minimum of 40KLOC for our PIR application and a total of 60KLOC for our PSI application. Note that we did not account for SMM code, the microcode used in the DRoT, or the TPM as it is not public information, and we consider these elements as part of the hardware. This should be compared to other TEEs and system projects that try to keep a small footprint. For instance, Nitro Enclaves include an entire Linux kernel which blows up their TCB in the range of MLOC, several orders of magnitude bigger than our TCB. On the other hand, SGX would not include the SMM code or the security monitor (18KLOC), but relies on a significant amount of microcode. As a point of comparison, XEN [17] (a popular mini-hypervisor), and Coreboot [21] (a minimal BIOS) each total 200KLOC. On the other hand, micro-kernels such as SeL4 [78] are in a similar size range with tens of thousands of lines of code.

9.3 Microbenchmarks

TPM Sign Operation. We evaluate the cost of signing a 32-byte message (i.e., a hash) using a physical TPM. We also compare the incurred overhead with two insecure baselines: 1) software signature in the application and 2) software signature performed by the security monitor (vTPM). Results can be found in Table 2. The vTPM signature requires a vmcall and a context switch to the security monitor, which incurs a small overhead (about 100μs) compared to the application-level signature. Operations with the physical TPMs are comparatively slow: we measure 196ms to perform a signature (which in our case boils down to loading the hash value in a PCR and requesting a TPM quote). This is 3 orders of magnitude slower than a vTPM. Some modern chipset-integrated hardware TPMs are faster than discrete TPMs and are still microarchitecturally isolated from the CPU. However, they are only 5× faster than the discrete TPM at best. These results hint at the importance of Argos' optimized attestation scheme, which only requires at most one signature per FHE evaluation and even less when using batching.

Enclave Fixed Costs. We evaluate the different fixed costs of different operations over the lifetime of an enclave. To obtain measurements, we instrument a dummy enclave for which we allocate 1GB of memory. Setup time is measured between the time a command line program requests the operating system to start an enclave and when execution reaches the main function of the enclave application. This means it includes the allocation of enclave resources, loading of the binary, page table setup, binary measurement, and runtime initialization. We can see that Nitro Enclaves have the highest setup time mostly due to the fact that they contain an entire Linux kernel that needs to be booted. SGXv2 also presents some large startup time. This is mostly due to all the special instructions required to start an enclave and the time it takes Gramine to initialize. As indicated in Gramine's documentation, initialization can initiate more than 200 OCALLs, which all incur expensive context switches with architectural flushes (CPU caches and state) that cost 8,000 — 12,000 cycles each [112]. This overhead can also be observed when running Argos with Gramine. In comparison, our custom runtime is initialized with minimal overhead, which might be essential for latency-sensitive applications.

Attestation time is the time it takes for an enclave to request an attestation quote. For SGXv2, support for remote attestation service by Intel is deprecated [45] so we use the analogue Microsoft Azure Attestation [111] scheme. Once provisioned, signing of the attestation quote is performed locally by the quoting enclave, hence the high performance similar to using a vTPM. For Nitro Enclaves, the attestation is performed by the hypervisor using a vTPM [12]

which also explains the high performance. Due to Argos's reliance on a physical TPM, we have a comparably slow attestation time.

Termination time is measured between when the enclave main function exits and when the control returns to the command-line program that launched the enclave. It includes all the time required to deallocate, clear resources, and return them to the OS. For SGX, that also implies a lot of cleanup at the hardware level. For Nitro Enclaves, it requires to wait for Linux to shutdown, which also adds delay. In Argos, since all sensitive data is encrypted, memory does not need to be sanitized before being reallocated to the OS.

Transcript Size. Our transcript is quite small and composed of 2 PCR values used by the DRoT, the initial value of some key registers for the execution environment, the hash of our binary, the hash of messages exchanged by the application, the hash of some server input if needed. It also contains the signature and the TPM public key along with the TPM certificates endorsed by the manufacturer. Concretely, we measure a small transcript size of only 1,407 bits, negligible in front of the size of an FHE ciphertext (≈500KB).

9.4 FHE Evaluation

We evaluate the overhead of circuit-level vFHE in Argos compared to semi-honest FHE. Results can be seen in Table 4. We use the same benchmarks as previous work on vFHE [131] and compile and evaluate the different circuits using the SEAL library [35]. The results from Viand. et al. highlight the impractical performance overheads when using cryptographic proofs such as Bulletproofs [29], Aurora [19], Groth16 [68] or Rinocchio [61], to build vFHE: between 6 and 7 orders of magnitude, with the exception of Rinocchio for the Tiny benchmark. Their semi-honest baseline is also relatively slow, hinting that they might have disabled hardware acceleration. Their implementation of FHE-in-TEE also shows poor performance probably due to the limitation of SGXv1 (limited 128MB EPC memory and unreliable timers). For a fair comparison, we run the benchmarks of SGXv2 and Nitro Enclaves. We compiled the benchmarks on one machine and used the same binary across or different setups. Here, the performance looks much better with single-digit overheads and even small speedups for Nitro Enclaves. Indeed, once setup, running a program in an enclave is equivalent to running it with bare-metal performance. We tried to keep the runtime as similar as possible between the different configurations, but small differences might explain these results. For Argos, we evaluate two different configurations. One using the Gramine runtime (Argos+G) and one using our custom runtime. Our custom runtime performs better than Gramine. This might be for a couple reasons including that the MUSL library tends to be faster than Glibc, our custom runtime simply ignores interrupts sent by the OS scheduler (the enclave is not de-scheduled), and our custom syscall handlers handle less corner cases and are more efficient (for instance, to allocate memory). In conclusion, Argos shows performance comparable to commercial TEEs while offering better security. Our custom runtime makes it possible to reduce the TCB and improve performance at the cost of some engineering effort to port applications.

Platform	Tiny	Small	Medium
Baseline [131]	2ms	11ms	14ms
Bulletproofs [29]	7569s	3957s	8697s
Aurora [19]	1554s	3750s	5028s
Groth16 [68]	196s	473s	634s
Rinocchio [61]	320ms	305s	443s
SGXv1 [131]	154ms	1100ms	1260ms
Baseline Azure	283us	1727us	3170us
SGXv2 Azure	290us (+3%)	1840us (+7%)	3638us (+15%)
Baseline AWS	324us	1889us	3456us
Nitro Enclave	317us (-2%)	1827us (-3%)	3450us (-0%)
Baseline Local	351us	2341us	4376us
Argos+G	392us (+12%)	2702us (+16%)	5202us (+19%)
Argos	352us (+0%)	2480us (+6%)	4447us (+2%)

Table 4: Time for FHE evaluation for three circuits. Baseline is always semi-honest FHE *without* integrity. Greyed values are reported from [131]. We indicate the overhead compared to the baseline on the same machine.

	Per Enclave			Per Query		Per Batch
	Setup	Pre-Proc.	Hash	Proc.	Hash	Attest.
Baseline	3	2835	N/A	1593	N/A	N/A
Argos+G	1559	3192	465	1812	1	196
Argos	158	2821	465	1610	1	196

Table 5: Breakdown of end-to-end execution times for authenticated private information retrieval (ms).

		Enclave			Query		Batch
	#C	Setup	Pre-Proc.	Hash	Proc.	Hash	Attest.
Baseline	1	4	40351	N/A	3231	N/A	N/A
Argos+G	1	1576	44530	132	3732	187	196
Argos	1	179	37054	105	3354	148	196
Baseline	3K	4	40348	N/A	3276	N/A	N/A
Argos+G	3K	1530	44467	131	3740	187	196
Argos	3K	175	36933	104	3403	148	196

Table 6: Breakdown of end-to-end execution times for private set intersection (ms). #C: size of the client set.

9.5 Authenticated Private Information Retrieval

We evaluate the overhead of authenticated PIR using Argos compared to semi-honest unauthenticated PSI. We use the SealPIR library from Angel et. al. [4]. We use the BFV scheme with polynomials of degree 4096. The plaintext modulus is set to 20. We evaluate performance for a database of size $2^{20} \approx 1$M where each element measures 128 bytes. We allocate 4GB of static memory to our enclave. We instrument the code to hash the database, queries, and responses and to attest the final transcript once all queries have been processed. Results can be found in Table 5. We compare two configurations for Argos. One using the Gramine runtime (Argos+G) and one using our custom runtime. For Argos+G, the server setup –which is a fixed cost– shows an overhead of +83% mostly due to overhead setting up the enclave (+55%) and hashing the database (+16%). For query processing, we measure an overhead of 13% for fetching one element. Here, hashing the response before sending it to the client takes a negligible amount of time. Taking

into account the signing of the final transcript, the overhead goes up to 26% for retrieving one element. Batching queries here would amortize costs down to a minimum of 13% for numerous queries. For Argos using our custom runtime, overhead goes down to 21% for setup (mostly due to hashing of the batabase) and 1% for query processing. This is due to our more efficient handling of system calls. The only cost left is attestation that can be amortized by batching requests (see Section 6.2). Once again, our custom runtime offers better performance than Gramine. For all setups, Argos's transcript incurs negligible (>1%) communication overhead.

Recent work on single-server authenticated PIR using cryptographic constructions by Colombo. et al., while providing better security guarantees, show a performance overhead of 30-100× [41]. Other recent work, VeriSimplePIR [49], shows more reasonable online overheads (13%-20%) but incurs significant offline communication of the same order of magnitude as the database itself (2GiB communication for a 4GiB dataset). This also makes updating the database extremely difficult, while our scheme only requires updating a Merkle tree and communicating the commitment.

9.6 Authenticated Private Set Intersection

We evaluate the overhead of authenticated PSI using Argos compared to semi-honest non-attested PSI. We use the APSI library compatible with SEAL that implements the state-of-the-art FHE-based scheme from Cong et. al. [42]. We use the BFV scheme with parameters 8192 for the polynomial degree modulus and the coefficient modulus. The plaintext modulus is set to 65537. We evaluated the performance of the unlabeled scheme with a server set size of $2^{20} \approx 1M$. We allocate 4GB of static memory to our enclave. We instrument the code to hash the database, queries, and responses and to attest the final transcript once all queries have been processed. Results can be found in Table 6. For Argos+Gramine, the server setup phase, we measure an overhead of 15% mainly due to preprocessing operations, but with our custom runtime, we see a *speedup* of 8% due to our fast handling of memory allocation, and other syscalls. Hashing the database is negligible here. For query time, we see +16% and +8% overhead for Gramine and our custom runtime, respectively. For all setups, Argos incurs negligible communication (>1%). In comparison, the state-of-the-art for malicious PSI using a cryptographic construction [114] shows a 19% performance overhead and a 67% communication overhead compared to their own semi-honest construction for a dataset of 2^{20} elements. However, they cannot guarantee that the server uses the same dataset across clients and require to compute and transfer a large commitment for each client.

Attested Transcript. Our remote attestation scheme makes it possible to amortize the cost of attestation over messages and queries. In the PSI scheme we use, each query requires two rounds of communication (one round for OPRF, and one for the query itself), plus one initial attestation, which would bring the overhead for one query to +27%. Instead, we only need to pay the cost of one attestation operation per batch. The PSI scheme we use already supports batching, but only up to a certain client set size (this is a limitation of FHE-based PSI schemes). For our given server set size, we empirically measure a non-negligible failure rate for client set size above 3000 elements. A client who would want to compute the intersection of a bigger set would have to do so using several queries. That means that the maximum overhead of 15% for a client set size of 1 can be amortized for larger client set sizes. For instance, for a client set size of 30,000, computing the complete intersection would take 10 queries and overhead would be reduced back to 8%. This phenomenon would be heightened for more efficient PSI schemes as the cost of TPM signing is constant.

9.7 Analysis of the Remaining Attack Surface

Argos primarily defends against microarchitectural side channels, which constitute the vast majority of TEE attacks so far (43 published according to [85]). While Argos is vulnerable to fault injection [76, 100, 113] and Rowhammer attacks (less studied on TEEs), these threats are not fatal – software can be hardened against fault injections [26], and no Rowhammer attacks have demonstrated the capabilities needed to compromise Argos (gaining hypervisor privilege on a hardware VM). Regarding physical attacks, Argos protects against ColdBoot [70] and physical side channels on the CPU [83], but remains vulnerable to physical fault injection attacks [27, 37]. The TPM only exposes a minimal attack surface for physical side channels, and modern TPMs are now directly integrated into the SoC, significantly raising the bar for these attacks [74]. Indeed, integrated TPMs do not implement modern features such as DVFS or RAPL, making all known software attacks such as Hertzbleed [133] or PLATYPUS [89] and others [140] impossible. Other physical attacks are out-of-scope, but would still be significantly less practical: 1) CPU-noise most likely masks signals from small integrated TPM, 2) TPM cryptography is assumed *constant-time*, thwarting simple power and EM attacks [62, 92], 3) for *constant-time* code, side channels are sometimes due to advanced circuit optimizations [88], unlikely in a small integrated co-processor. Finally, our TCB might still contain bugs, and attacks that exploit implementation mistakes have been demonstrated on the Intel DRoT and the associated secure coprocessor [28, 43, 55, 135, 136]. However, these implementation errors can be *patched when discovered* —unlike vulnerabilities to microarchitectural side channel attacks.

10 Related Work

TPMs and Hypervisor-based TEEs. Trusted platform modules (TPMs) combined with dynamic root-of-trust technologies have long served as dedicated hardware to enforce platform integrity. Flicker [94] proposes to use these two technologies combined to isolate and attest small pieces of application logic, or PALs while reducing the software TCB to a handful of lines of code. No keys ever leave the TPM in Flicker, which gives it the same resistance to microarchitectural side channels as Argos (even if not discussed in the paper at the time). However, their complete reliance on hardware mechanisms creates significant overheads, i.e., 2-3 orders of magnitude. Virtualization technologies (making their appearance in processors around 2008) aim at isolating different trusted domains in the form of virtual machines (VMs). They create a new privileged execution mode dedicated to the hypervisor and provide hardware primitives such as extended (or nested) page tables to enforce memory isolation between different security domains. These technologies identified the opportunity to virtualize the TPM [107] and provide confidentiality and integrity for trusted domains with

TEE Platform	Security						Usability			Performance		
	TCB	μ-arch SC	Cold Boot	Phys. SC CPU	Fault Inject.	Phys. SC TPM	Availability	Dedic. HW	Implementation	Setup	Attest.	Comp.
ZK Proofs	Null	P	P	P	P	P	SW	SW	Open Source	– – – –	– – – –	– – – –
Nitro Enclaves	Large	V	V	V	V	V	AWS	Yes	Closed Source	– – –	+ +	+ +
Arm TrustZone	Small	V	V	V	V	V	High	Yes	HW Closed Source	+	+	–
Intel SGX V1	Small	V	P	V	V	V	Deprecated	Yes	HW Closed-Source	– –	+	– –
Intel SGX V2	Small	V	P	V	V	V	Cloud	Yes	HW Closed-Source	– –	+	+
AMD SEV	Large	V	P	V	V	V	Cloud	Yes	HW Closed-Source	– – –	+	+ +
Intel TDX	Large	V	P	V	V	V	Coming	Yes	HW Closed-Source	– – –	+	+ +
ARM CCA	Large	V	P	V	V	V	Coming	Yes	HW Closed-Source	– – –	+	+ +
TrustVisor[93]	Small	V	V	V	V	V	Deprecated	No	Open Source	–	+ +	+ +
Flicker [94]	Tiny	P	P	P	V	V	Deprecated	No	Open Source	– – –	– – –	– – –
Argos	Small	P	P	P	V	V	High	No	Open Source	+	–	+ +

Table 7: Comparison of TEE Platforms across Security, Usability and Performance.
SC: Side Channel, P: Protected, V: Vulnerable. +'s and -'s represent relative (and subjective) measure of performance.

minimal performance overhead. TrustVisor [93] is the first work to introduce this hypervisor-based architecture, which was eventually adopted by cloud providers such as AWS for their Nitro Enclaves [11]. Virtual TPMs are also the standard for trusted computing in modern clouds [12, 14, 115]. As a result, all these architectures are insecure under our threat model as they expose sensitive key material to microarchitectural side channels.

Hardware TEE Platforms. All commercial hardware TEE platforms are vulnerable to microarchitectural side channels. This includes, but is not limited to, Intel SGX [46], Intel TDX [39], AMD SEV-SNP [121], ARM TrustZone [110] and ARM CCA [86]. Secrets and key material are sometimes stored on secure co-processors, such as Intel ME [44], AMD PSP [28], or the Apple secure enclave [6]. However, to our knowledge, all existing designs expose some secret material on the CPU such as platform-level secrets for Intel SGX/TDX [46], vTPM key material for AMD SEV-SNP [5] or application-level keys for Apple Private Cloud Compute [8]. Beyond their vulnerability to side channels, commercial platforms also suffer from availability issues as they require dedicated hardware and are not available on consumer-grade processors. For example, Intel announced that SGX would be deprecated on non-server-grade processors [73] and also deprecated its remote attestation service [45]. A comprehensive comparison of existing platforms with Argos can be found in Table 7. Academic platforms [13, 16, 24, 47, 51, 58, 58, 59, 84, 130, 134] are limited in terms of how much hardware customization they can perform to defend against microarchitectural side channels. As a result, the vast majority simply consider side channels out of scope, with the exception of Sanctum [47] and MI6 [22] which are built on top of an open-source RISCV processor.

Combining Cryptography and TEEs. Previous work has identified that protecting remote attestation mechanisms alone against microarchitectural side channels was easier than enforcing confidentiality of entire enclave programs [125]. Some even suggest to combine trusted hardware with cryptography to build hybrid mechanisms, but all show important performance overheads, use (insecure) SGX and are vulnerable to microarchitectural side channels. CrypTFlow [81] looks at secure inference and proposes to build maliciously-secure MPC from semi-honest MPC schemes using SGX. They show a 3× performance overhead over the semi-honest scheme. Chex-Mix [102] leverages SGX with FHE to build secure inference, but considers microarchitectural side channels out of scope and even relies on SGX confidentiality for the privacy of the model weights. They show a 142% performance overhead compared to semi-honest FHE and consider key recovery attacks out of scope. Viand et. al. [131] introduces many useful formalisms that inspired Argos. Their paper primarily focuses on combining FHE with cryptographic proofs but they also consider an FHE-in-TEE approach and design a special protocol to optimize its performance. Nevertheless, they use SGX, do not adapt their formalism to the FHE-in-TEE approach, and their results show an 80× performance overhead when compared to Argos. In contrast, Argos presents a new TEE design for malicious and verifiable FHE with a dedicated remote attestation scheme. Argos is secure *by design* against microarchitectural side channels and shows minimal performance overhead compared to semi-honest baselines.

11 Conclusion

We present Argos, the first *integrity-only* enclave platform designed to build maliciously-secure verifiable FHE. Argos is secure *by design* against microarchitectural side channels and transient execution attacks and can be used to build fully malicious and authenticated PSI and PIR schemes. It requires no specialized hardware and is compatible with commodity processors from 2008 onward. Argos only incurs 3% average performance overhead for FHE computation, less than 8% performance overhead for complex protocols, and no communication overhead. By demonstrating how to combine cryptography with trusted hardware, Argos paves the way for widespread deployment of FHE-based protocols beyond the semi-honest setting, without the overhead of cryptographic proofs.

Acknowledgments

The authors thank Charly Castes, Adrien Ghosn, Neelu S. Kalani, and the authors of Tyche [30] for providing early access to their code and for their invaluable assistance in setting up Tyche on our hardware and adapting the platform to our needs. We thank Sacha Servan-Schreiber for helpful discussions on fully homomorphic encryption and useful feedback. The authors used Claude to revise the text in most of this paper to correct for typos, grammatical errors, and awkward phrasing. This research was supported in part by NSF contracts 2330065, 2115587 and 1955270, and Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference number RGPIN-2023-04796, and an NSERC-CSE Research Communities Grant under funding reference number ALLRP-588144-23. Any research, opinions, or positions expressed in this work are solely those of the authors and do not represent the official views of the Communications Security Establishment Canada or the Government of Canada.

References

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007. Proceedings*. Springer, 225–242.

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2003. The EM side—channel (s). In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*. Springer, 29–45.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying {Constant-Time} Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 962–979.

[5] Pedro Antonino, Ante Derek, and Wojciech Aleksander Wołoszyn. 2023. Flexible remote attestation of pre-SNP SEV VMs using SGX enclaves. *IEEE access* 11 (2023), 90839–90856.

[6] Apple. 2022. Apple Platform Security. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. Accessed on 28.04.2023.

[7] Apple. 2024. Private Cloud Compute: A new frontier for AI privacy in the cloud. https://security.apple.com/blog/private-cloud-compute/.

[8] Apple. 2024. Private Cloud Compute Security Guide. https://security.apple.com/documentation/private-cloud-compute.

[9] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer, 483–501.

[10] Shahla Atapoor, Karim Baghery, Hilder VL Pereira, and Jannik Spiessens. 2024. Verifiable FHE via Lattice-based SNARKs. *Cryptology ePrint Archive* (2024).

[11] AWS. 2024. AWS Nitro Enclaves Documentation. https://docs.aws.amazon.com/enclaves/latest/user/nitro-enclave.html.

[12] AWS. 2024. AWS Nitro TPM Documentation. https://aws.amazon.com/blogs/compute/deep-dive-into-nitrotpm-and-uefi-secure-boot-support-in-amazon-ec2/.

[13] Ahmed M Azab, Peng Ning, and Xiaolan Zhang. 2011. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*. 375–388.

[14] Azure. 2024. Virtual TPMs in Azure confidential VMs. https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-tpms-in-azure-confidential-vm.

[15] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, Caroline Sporleder, et al. 2010. Acoustic {Side-Channel} attacks on printers. In *19th USENIX Security Symposium (USENIX Security 10)*.

[16] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves.. In *USENIX Security Symposium*. 1073–1090.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[18] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. 1998. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*. Springer, 26–45.

[19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. 2019. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer, 103–128.

[20] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 26–35.

[21] Anton Borisov. 2009. Coreboot at your service! *Linux Journal* 2009, 186 (2009), 1.

[22] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.

[23] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[24] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[25] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th USENIX workshop on offensive technologies (WOOT 17)*.

[26] Jakub Breier and Xiaolu Hou. 2022. How practical are fault injection attacks, really? *IEEE Access* 10 (2022), 113122–113130.

[27] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2875–2889.

[28] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. 2019. Insecure until proven updated: analyzing AMD SEV's remote attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1087–1099.

[29] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 315–334.

[30] Charly Castes, Adrien Ghosn, Neelu S Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. 2023. Creating Trust by Abolishing Hierarchies. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 231–238.

[31] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, et al. 2017. Security of homomorphic encryption. *HomomorphicEncryption. org, Redmond WA, Tech. Rep* (2017).

[32] Bhuvnesh Chaturvedi, Anirban Chakraborty, Ayantika Chatterjee, and Debdeep Mukhopadhyay. 2022. A practical full key recovery attack on tfhe and fhew by inducing decryption errors. *Cryptology ePrint Archive* (2022).

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

[34] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1223–1237.

[35] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple encrypted arithmetic library-SEAL v2. 1. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*. Springer, 3–18.

[36] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1243–1255.

[37] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. {VoltPillager}: Hardware-based fault injection attacks against Intel {SGX} Enclaves using the {SVID} voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*. 699–716.

[38] Massimo Chenal and Qiang Tang. 2015. On key recovery attacks against existing somewhat homomorphic encryption schemes. In *Progress in Cryptology-LATINCRYPT 2014: Third International Conference on Cryptology and Information Security in Latin America Florianópolis, Brazil, September 17–19, 2014 Revised Selected Papers 3*. Springer, 239–258.

[39] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel tdx demystified: A top-down approach. *Comput. Surveys* 56, 9 (2024), 1–33.

[40] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[41] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J Wu, and Bryan Ford. 2023. Authenticated private information retrieval. In *32nd USENIX security symposium (USENIX Security 23)*. 3835–3851.

[42] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from homomorphic encryption with reduced computation and communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1135–1150.

[43] Intel Corporation. 2019. Intel CSME, Intel SPS, Intel TXE, Intel ‘DAL, and Intel AMT 2019.1 QSR advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00213.html.

[44] Intel Corporation. 2022. Intel® Converged Security and Management Engine (Intel® CSME) Security. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf.

[45] Intel Corporation. 2024. Intel® Software Guard Extensions Attestation Service Using Intel® Enhanced Privacy Identification End-of-Life Timeline. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/sgx-ias-using-epid-eol-timeline.html.

[46] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[47] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[48] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P Smart, Samuel Tap, and Michael Walter. 2023. Noah’s Ark: Efficient Threshold-FHE Using Noise Flooding. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 35–46.

[49] Leo de Castro and Keewoo Lee. 2024. VeriSimplePIR: verifiability in simplePIR at no online cost for honest servers. *Cryptology ePrint Archive* (2024).

[50] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. 2010. Linear-complexity private set intersection protocols secure in malicious model. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 213–231.

[51] Liang Deng, Qingkai Zeng, Weiguang Wang, and Yao Liu. 2014. EqualVisor: Providing memory protection in an untrusted commodity hypervisor. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 300–309.

[52] Marian Dietz and Stefano Tessaro. 2024. Fully malicious authenticated PIR. In *Annual International Cryptology Conference*. Springer, 113–147.

[53] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[54] Alexander Ermolov. 2016. Safeguarding rootkits: Intel BootGuard. *Remote access (01 Sep 2021): https://2016. zeronights. ru/wp-content/uploads/2017/03/Intel-BootGuard. pdf* (2016).

[55] Mark Ermolov and Maxim Goryachy. 2017. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe* (2017).

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

[57] Prastudy Fauzi, Martha Norberg Hovd, and Håvard Raddum. 2022. On the IND-CCA1 security of FHE schemes. *Cryptography* 6, 1 (2022), 13.

[58] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the {PENGLAI} Enclave. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 275–294.

[59] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 287–305.

[60] William Futral and James Greene. 2013. *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*. Springer Nature.

[61] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. 2023. Rinocchio: SNARKs for ring arithmetic. *Journal of Cryptology* 36, 4 (2023), 41.

[62] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: physical side-channel key-extraction attacks on pcs: Extended version. *Journal of Cryptographic Engineering* 5 (2015), 95–112.

[63] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[64] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 307–328.

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

[66] Ben Gras, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida, et al. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks.. In *USENIX Security Symposium*, Vol. 216.

[67] Matthew Green, Watson Ladd, and Ian Miers. 2016. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 1591–1601.

[68] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[69] Qian Guo, Denis Nabokov, Elias Suvanto, and Thomas Johansson. 2024. Key Recovery Attacks on Approximate Homomorphic Encryption with Non-Worst-Case Noise Flooding Countermeasures. In *Usenix Security*.

[70] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[71] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. 2023. Private web search with Tiptoe. In *Proceedings of the 29th symposium on operating systems principles*. 396–416.

[72] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast {Single-Server} Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3889–3905.

[73] Intel. 2024. Core Processors. Deprecated Technologies. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/004/deprecated-technologies/.

[74] Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert. 2023. faulTPM: Exposing AMD fTPMs’ Deepest Secrets. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1128–1142.

[75] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. 2018. Titan: enabling a transparent silicon root of trust for cloud. In *Hot Chips: A Symposium on High Performance Chips*, Vol. 194. 10.

[76] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. {V0LTpwn}: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*. 1445–1461.

[77] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. 2017. Private set intersection for unequal set sizes with mobile applications. In *Privacy Enhancing Technologies Symposium*. De Gruyter, 177–197.

[78] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[79] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[80] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael B Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer.. In *WOOT@ USENIX Security Symposium*.

[81] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 336–353.

[82] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings 38th annual symposium on foundations of computer science*. IEEE, 364–373.

[83] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An {Off-Chip} attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*.

[84] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[85] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. 2024. SoK: Understanding Design Choices and Pitfalls of Trusted Execution

Environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 1600–1616.
[86] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.
[87] Yehuda Lindell, David Cook, Tim Geoghegan, Sarah Gran, Rolfe Schmidt, Ehren Kret, Darya Kaviani, and Raluca Ada Popa. 2023. The deployment dilemma: Merits & challenges of deploying MPC. https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html.
[88] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. {AMD} prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*. 643–660.
[89] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[90] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.
[91] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. 2023. Siloz: Leveraging DRAM isolation domains to prevent inter-VM rowhammer. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 417–433.
[92] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. 2018. How secure is green IT? The case of software-based energy side channels. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23*. Springer, 218–239.
[93] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 143–158.
[94] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 315–328.
[95] Samir Jordan Menon and David J Wu. 2022. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 930–947.
[96] Samir Jordan Menon and David J Wu. 2024. YPIR: High-Throughput Single-Server PIR with Silent Preprocessing. *Cryptology ePrint Archive* (2024).
[97] Daniel Moghimi. 2023. Downfall: Exploiting speculative data gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7179–7193.
[98] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL:TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 2057–2073.
[99] moxie0. 2017. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/. Accessed on 28.04.2023.
[100] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.
[101] musl. 2024. Musl Libc. https://musl.libc.org.
[102] Deepika Natarajan, Andrew Loveless, Wei Dai, and Ronald Dreslinski. 2021. Chex-mix: Combining homomorphic encryption with trusted execution environments for two-party oblivious inference in the cloud. *Cryptology ePrint Archive* (2021).
[103] Ofri Nevo, Ni Trieu, and Avishay Yanai. 2021. Simple, fast malicious multiparty private set intersection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1165.
[104] Cong Nie. 2007. Dynamic root of trust in trusted computing. In *TKK T1105290 Seminar on Network Security*. Citeseer.
[105] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 1–20.
[106] Bijeeta Pal, Mazharul Islam, Marina Sanusi Bohuk, Nick Sullivan, Luke Valenta, Tara Whalen, Christopher Wood, Thomas Ristenpart, and Rahul Chatterjee. 2022. Might I get pwned: A second generation compromised credential checking service. In *31st USENIX Security Symposium (USENIX Security 22)*. 1831–1848.
[107] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*. 305–320.
[108] James L Peterson and Theodore A Norman. 1977. Buddy systems. *Commun. ACM* 20, 6 (1977), 421–431.
[109] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: fast, malicious private set intersection. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 739–767.
[110] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
[111] Gramine Project. 2024. Gramine Documentation. Microsoft Azure Attestation (MAA) Integration. https://github.com/gramineproject/contrib/tree/master/Integrations/azure/ra_tls_maa.
[112] Gramine Project. 2024. Gramine Documentation. Performance tuning and analysis. https://gramine.readthedocs.io/en/stable/performance.html.
[113] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 1–6.
[114] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing fast PSI from improved OKVS and subfield VOLE. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2505–2517.
[115] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. 2016. fTPM: A Software-Only Implementation of a TPM Chip.. In *USENIX Security Symposium*, Vol. 16. 841–856.
[116] Peter Rindal and Phillipp Schoppmann. 2021. VOLE-PSI: fast OPRF and circuit-PSI from vector-OLE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 901–930.
[117] Mike Rosulek and Ni Trieu. 2021. Compact and malicious private set intersection for small sets. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1166–1181.
[118] Keegan Ryan. 2019. Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 181–194.
[119] Mark D Ryan. 2013. Enhanced certificate transparency and end-to-end encrypted mail. *Cryptology ePrint Archive* (2013).
[120] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 279–299.
[121] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).
[122] Nigel P Smart. 2023. Practical and Efficient FHE-based MPC. In *IMA International Conference on Cryptography and Coding*. Springer, 263–283.
[123] Yunqing Sun, Jonathan Katz, Mariana Raykova, Phillipp Schoppmann, and Xiao Wang. 2024. Actively Secure Private Set Intersection in the Client-Server Setting. Cryptology ePrint Archive, Paper 2024/570. https://eprint.iacr.org/2024/570
[124] Cheng Tan, Lijun Zhang, and Liang Bao. 2020. A Deep Exploration of BitLocker Encryption and Security Analysis. In *2020 IEEE 20th International Conference on Communication Technology (ICCT)*. IEEE, 1070–1074.
[125] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 19–34.
[126] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.
[127] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.
[128] Leendert Van Doorn. 2006. Hardware virtualization trends. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments*, Vol. 14. 45–45.
[129] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX fails in practice.
[130] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. 2023. CHERI-TrEE: Flexible enclaves on capability machines. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 1143–1159.
[131] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. 2023. Verifiable fully homomorphic encryption. *arXiv preprint arXiv:2301.07041* (2023).
[132] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 225–236.
[133] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. 679–697.
[134] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v.. In *NDSS*.
[135] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking intel trusted execution technology. *Black Hat DC* 2009 (2009), 1–6.

[136] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. 2009. Another way to circumvent Intel trusted execution technology. *Invisible Things Lab* (2009), 1–8.
[137] Tianhong Xu, Aidong Adam Ding, and Yunsi Fei. 2024. TrustZoneTunnel: A Cross-World Pattern History Table-Based Microarchitectural Side-Channel Attack. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 01–11.
[138] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.
[139] Zhi Zhang, Decheng Chen, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, et al. 2024. SoK: Rowhammer on Commodity Operating Systems. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 436–452.
[140] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. 2021. Red alert for power leakage: Exploiting intel rapl-induced side channels. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 162–175.
[141] Vincent Zimmer and Michael Krau. 2016. Establishing the root of trust. *UEFI. org document dated August* (2016).

A Primer on Measured Boot

Measured boot is obtained using a trust chain that binds the measurement of the operating system or hypervisor to a component that is inherently trusted by the system (the root). Starting from the root, every element of the chain measures (hashes) and signs the binary of the next element, ensuring its integrity. This process is sometimes referred to as secure boot, as, historically, the chain of trust was composed of the entire boot chain. The root of trust (RoT) is the base case of this recursive security argument, often a hardware mechanism (e.g. read-only memory) that enforces integrity for the first instructions executed at boot and which integrity is supported by a certificate signed by the manufacturer [54, 141].

When elements of the boot chain are measured (for instance, before any memory can be trusted), these measurements need to be stored in a secure location where they cannot be tampered with, for instance, by a compromised BIOS. To provide that functionality, a small chip or coprocessor, called a trusted platform module (TPM) will be in charge of securely storing hash-based measurements in special registers (PCRs). The interface exposed by the TPM is quite limited. For example, measurements can only be hash-extended (never reset). TPMs also have other capabilities, like long-term secrets storage referred to as *secret sealing* (e.g., a key for disk encryption [124]) that can only be accessed if the PCRs are in a specific state, i.e., the correct system has been booted. The TPM also has attestation capabilities, which will use a private key tied to the TPM to sign or *attest* the values of the PCRs and the state of the system. However, the TPM is merely a subordinate to the RoT and the software in the chain of trust, as it needs to be instructed and fed inputs to extend its internal measurements.

Any code measured through that process, up to the final domain, is considered part of the trusted code base (TCB). Code in the TCB is critical for the security of the system (it enforces the integrity of the next step in the chain of trust), and as a result, code in the TCB is expected to be bug-free. This assumption is practically impossible to enforce for a TCB that contains millions of lines of code (e.g., the Linux kernel with 28MLOC), hence the incentive to exclude as much code as possible from the TCB. A dynamic root of trust (DRoT) makes it possible to remove elements of the boot chain from the chain of trust, hence reducing the TCB. DRoT is implemented as a hardware extension (e.g., Intel TXT or AMD DRMT [60, 104]) that can instruct and control the TPM independently of any firmware or software. However, DRoT and TPM were originally conceived to attest to one trusted domain (the OS). That means 1) there is no mechanism to isolate more than one domain at a time 2) TPM hardware resources (PCRs, key storage, bandwidth) are sized for one domain and extremely constrained.

B Primer on Microarchitectural and Physical Attacks

B.1 Side Channel Attacks

In side-channel attacks, an attacker leverages some shared state (initially not meant for communication) to learn information about the execution of a victim program and extract victim secrets. There exist many types of side channels, and they can first be categorized by the medium or shared state that is exploited by the attacker to extract information. For example, classic *timing* side channels exploit the fact that the program execution time might vary depending on the value of some secret inputs. It was shown early on [3] that cryptographic algorithms such as RSA using *non-constant-time* implementation (such as square and multiply) were vulnerable to these types of side channels. Beyond simple timing, other types of side channel are

- *microarchitectural* side channels, which exploit shared *microarchitectural* state. These include memory caches [25, 65], translation look-aside buffers (TLBs) [66], branch predictors [1, 56], DRAM controllers [132], or any other shared microarchitecture.
- *physical* side channels, which exploit *physical* mediums such as power consumption [89, 133], electromagnetic emissions [2], or acoustic noise [15].

Finally, another way to classify *physical* side channels is (1) if the attack can be mounted in *software* or (2) if the attacker requires *physical* access to the machine running the victim program. Argos defend against all types of side channels targeting the CPU and is vulnerable to *physical* side-channels targeting the TPM. However, to our knowledge, no such attack has been published (see Section 9.7).

B.2 Cache Side Channel Attacks

Cache side channels are an example of *microarchitectural* side channels. Here, the shared channel is shared caches. The caches are between the CPU and main memory (see fig. 1) and ensure that the performance penalty of accessing a location in memory (100 cycles) can be amortized if the same memory value has been recently accessed. We usually talk about a cache hierarchy, as a CPU usually contains a core-private fast (1 cycle) (but small) "L1" cache which is connected to other larger (but slower, 10 - 100 cycles) caches ("L2" or "L3") closer to memory. In principle, only the last level cache (LLC) is shared among cores, but this is machine-dependent. That means that caches can be shared between a victim and an attacker program or *temporarily* (between context switches on the same core [138]) or *spatially* (across cores if the cache is shared [90]). To exploit caches as side channels, the attacker usually exploits the timing difference of loading a cache and non-cached value from memory. We will describe (a simplified version) or Prime&Probe [105], a powerful cache side channel attacks:

(1) **PRIME:** The attacker will access a large array to fill up the cache with its own data.
(2) **WAIT:** The attacker will wait for the victim to run and access the shared cache.
(3) **PROBE:** The attacker will access the large array again, but time each memory access.

If a memory access takes longer, that means that the corresponding value has been evicted from the cache by the victim when running, and the attacker can infer information regarding the *addresses* accessed by the victim in memory. In fact, to remove a piece of data from a cache entry, the new memory value must have the same *set index*, hence sharing some common address bits with the original entry. These attacks are efficient, but require the victim program to access memory locations whose addresses depend on a secret. This is exactly why constant-time programming for cryptographic algorithms does not perform any memory accesses on secret-dependent addresses.

B.3 Spectre Attacks

Miroarchitectural side channel attacks are limited by the presence of gadgets in the victim code that can be exploited (e.g., secret dependent memory accesses). These gadgets might exist in the code but may never be executed under "normal" sequential program execution. Unfortunately, attacks from the Spectre family [33, 79, 80] have demonstrated that such gadgets could be accessed *speculatively* and weaponized into *universal read gadgets*.

Speculative execution (sometimes called transient execution) is due to a family of speculative mechanisms for performance optimizations present on modern processors. These include the pattern history table, the branch target buffer, and the return stack buffer, which all have their corresponding Spectre variants. They make it possible to *speculatively* execute code or load data before knowing if it corresponds to the correct control / data flow for the program. If the microarchitecture guesses right, a lot of precious CPU cycles have been saved. If the microarchitecture guesses wrong, some performance penalty needs to be paid to rollback the *architectural* state (but not the *microarchitectural* state) before correct execution can restart. These optimizations are in large part responsible for the great performance of modern CPUs. Disabling them altogether would have disastrous performance effects (200-300% slowdown [22]). However, they make reasoning about the security of programs especially tricky as there is no clear *speculative* execution flow for a given program. In addition, when a speculative execution path is deemed wrong, the different pieces of microarchitectural states that were modified (e.g., cache) are usually not rolled back. This has significant security implications, as this potentially secret-dependent state can now be observed by an attacker using adequate microarchitectural side channels. Argos defend against these attacks by eliminating all microarchitectural side channels.

B.4 Spectre V1

As explained above, a large number of Spectre variants can be generated by combining a mechanism that triggers speculation with microarchitectural side channels. We will now describe the original Spectre attack (V1 [79]) that exploits the branch predictor (pattern history table) and a cache side channel. The branch predictor is one of the most common microarchitectural mechanisms for speculation. When the CPU encounters a branch (i.e., `if`, `while`, or `for` statement), and before the result of the branch condition is known, the branch predictor guesses the direction of the branch (taken vs. non-taken) and speculatively executes the corresponding execution path. A branch predictor is *trained* using the history of correct directions for the branch to make its guesses more efficient. That means that for a simple branch predictor, if a branch has been taken a dozen times in the same execution context, it will be predicted as taken in the next run.

Let us now describe the classic Spectre V1 attack. Let us consider the following pseudo-code located in the victim program:

```
if(i > 0 && i < 10) {
    s = array1[i];
    b = array2[s];
}
```

Here, we assume that the attacker can call this code snippet by interacting through the victim program API. Note that double-memory access is a perfect *transmitter* for a cache side channel, as it accesses a memory value `s` and then leaks its content by accessing an address that depends on it (`array2[s]`). However, thanks to the bound check on index `i`, the attacker should not be able to access arbitrary data location in victim memory and load it into `s`. An attacker can exploit the branch predictor by first calling the code snippet repeatedly with inbound indices, training it to predict the branch on line 1 as non-taken. Subsequently calling the snippet with an out-of-bounds index causes the branch predictor to *incorrectly* speculate, executing both the arbitrary memory access on line 2 and the cache side channel transmission on line 3. This creates a *universal read gadget*, which enables arbitrary memory access within the victim's address space.

B.5 Cold Boot

Cold boot is a physical attack that requires physical access to the target machine running the victim program. DRAM is volatile, but data values may persist for a few seconds / minutes even after a power switch-off. If an attacker acts swiftly, it has the opportunity to *cold boot* the machine with an attacker-controlled operating system (on a USB disk, for instance) and dump all the memory. Argos defends against these attacks by never storing sensitive data in main memory.

B.6 Fault Injection Attacks

Fault injections are a different family of attacks than side channels, as they do not directly aim at extracting victim secrets, but rather at *modifying* the victim control flow or its state by introducing faults into the system. Typically, such attacks require physical access to the system. Voltage glitching attacks are an example of such attacks that consist of lowering the power supply of a system in order to introduce glitches or faults into a system. When properly synchronized and timed, changes in the power supply can affect the propagation of electric currents in the processor and affect the control or data flow of a program. For instance, instructions can be skipped and the result of instructions can be affected, resulting in faulty outputs. Other classic attacks include "voltage/clock glitching, electromagnetic pulses, and laser-based attacks" [26]. Argos does

not protect against such attacks. However, in some cases, they can be mitigated in software by adding extra checks on the control flow.

B.7 Rowhammer

Rowhammer [139] is a *software-mounted* fault injection attack that aims to modify the contents of the victim's memory. It exploits undesirable side effects in modern DRAM due to electromagnetic interference between closely located memory cells. When repetitively activating a memory row (hammering the row), adjacent rows might leak charge from their cells due to the interference and some of their bit values might flip, even if never directly accessed by the program. However, these attacks require the attacker to be able to access a memory row adjacent to the victim. Moreover, such attacks can be mitigated by Argos by ensuring that different security domains use distinct memory partitions [91].

C Formal Definitions

We use the formal definitions from Viand et. al. [131].

C.1 FHE Properties

Correctness A scheme is correct if any honest computation will decrypt to the expected result. More formally, a scheme is correct if for all functions f, and for all x in the domain of f:

$$\Pr\left[\mathcal{E}.\mathsf{Dec}_{\mathsf{sk}}(c_y) = f(x) \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)\\ c_x \leftarrow \mathcal{E}.\mathsf{Enc}_{\mathsf{key}}(x)\\ c_y \leftarrow \mathcal{E}.\mathsf{Eval}_{\mathsf{key}}(c_x, f)\end{array}\right] = 1.$$

Security We extend the definition of IND-CPA to our notion. Note that we do not require an evaluation oracle since $\mathsf{Eval}_{\mathsf{pk}}$ is public and can be executed by the adversary directly. Formally, a scheme is secure if for any PPT adversary $\mathcal{A}$ and any function f the advantage $\mathsf{Adv}^{\text{IND-CPA}}[\mathcal{A}](\lambda) = 2\left|\Pr[b = \hat{b}] - \frac{1}{2}\right|$ of the attacker in the following game is negligible in the security parameter λ:

$$\begin{array}{l}\underline{\text{IND-CPA } \textit{for}\text{ vFHE}}\\ (\mathsf{pk},\mathsf{sk}) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)\\ (\mathbf{m}_0, \mathbf{m}_1) \leftarrow \mathcal{A}^{O_{\mathcal{E}.\mathsf{Enc}}}(1^\lambda, f, \mathsf{pk})\\ (c^*, \pi^*) \leftarrow \mathcal{E}.\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_b)\\ \hat{b} \leftarrow \mathcal{A}^{O_{\mathcal{E}.\mathsf{Enc}}}(c^*).\end{array}$$

Extension for ϵ-approximate FHE schemes Approximate FHE schemes can be captured by changing the correctness property to the following: A scheme is correct if for all functions f, and for all x in the domain of f:

$$\Pr\left[\left\|\mathsf{Dec}_{\mathsf{sk}}(c_y) - f(x)\right\| \le \varepsilon \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)\\ c_x \leftarrow \mathcal{E}.\mathsf{Enc}_{\mathsf{key}}(x)\\ c_y \leftarrow \mathcal{E}.\mathsf{Eval}_{\mathsf{key}}(c_x, f)\end{array}\right] = 1$$

where $||\cdot||$ is a scheme-specific norm and ϵ is a scheme-specific upper bound on the decoding error (which may depend on f, pk, or other quantities of the scheme). We leave out details on how to adapt our other definitions and proofs to this setup.

C.2 Remote Attestation Properties

Completeness A scheme is complete if Verify will always accept an honestly computed result. More formally, a scheme is complete if for all functions g, and for all x in the domain of g:

$$\Pr\left[\Pi.\mathsf{Verify}_{\mathsf{sk}}(y, x, \pi_y) = \mathtt{true} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \Pi.\mathsf{Gen}(1^\lambda)\\ (y, \pi_y) \leftarrow \Pi.\mathsf{Prove}_{sk}(x, g)\end{array}\right] = 1.$$

Soundness A scheme is sound if the adversary cannot make Verify accept an incorrect answer. Formally, a scheme is sound if for any PPT adversary and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\Pi.\mathsf{Verify}_{\mathsf{sk}}(y, x, \pi_y) = 1\\ \wedge\\ y \neq g(x)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \Pi.\mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}(\mathsf{pk})\\ \text{x is in the domain of g}\\ (y, \pi_y) \leftarrow \mathcal{A}(\mathsf{pk}, x)\end{array}\right].$$

C.3 vFHE Properties

Correctness A scheme is correct if any honest computation will decrypt to the expected result. More formally, a scheme is correct if for all functions f, and for all x in the domain of f:

$$\Pr\left[\mathsf{Dec}_{\mathsf{sk}}(c_y) = f(x) \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y, \pi_y) \leftarrow \mathsf{Eval}_{\mathsf{key}}(c_x, f)\end{array}\right] = 1.$$

Completeness A scheme is complete if Verify will always accept an honestly computed result. More formally, a scheme is complete if for all functions f, and for all x in the domain of f:

$$\Pr\left[\mathsf{Verify}_{\mathsf{sk}}(c_y, c_x, \pi_y) = 1 \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y, \pi_y) \leftarrow \mathsf{Eval}_{\mathsf{key}}(c_x, f)\end{array}\right] = 1.$$

Soundness A scheme is sound if the adversary cannot make Verify accept an incorrect answer. Formally, a scheme is sound if for any PPT adversary $\mathcal{A}$ and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\mathsf{Verify}_{\mathsf{sk}}(c_y, c_x, \pi_y) = 1\\ \wedge\\ c_y \neq \mathcal{E}.\mathsf{Eval}(c_x, f)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}^{O_{\mathsf{Enc}}, O_{\mathsf{Dec}}}(\mathsf{pk})\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y, \pi_y) \leftarrow \mathcal{A}^{O_{\mathsf{Enc}}, O_{\mathsf{Dec}}}(c_x)\end{array}\right].$$

Note that in the literature [10, 131], soundness is sometimes defined using the conditional probability of

$$\mathsf{Verify}_{\mathsf{sk}}(c_y, c_x, \pi_y) = 1 \quad \wedge \quad \mathcal{E}.\mathsf{Dec}(c_y) \neq f(x).$$

This definition is impractical as it allows an attacker to compute a different circuit that represents the same function. We use a more restrictive definition instead, even if it requires the $\mathcal{E}$.Eval algorithm to be deterministic (which it is in our case).

Security We extend the definition of IND-CCA1 to our notion. Note that we do not require an evaluation oracle since $\mathsf{Eval}_{\mathsf{pk}}$ is public and can be executed by the adversary directly. Formally, a scheme is secure if for any PPT adversary $\mathcal{A}$ and any function f the advantage $\mathsf{Adv}^{\text{IND-CCA1}}[\mathcal{A}](\lambda) = 2\left|\Pr[b = \hat{b}] - \frac{1}{2}\right|$ of the attacker

in the following game is negligible in the security parameter λ:

$$\begin{array}{l}\underline{\text{IND-CCA1 } \textit{for} \text{ vFHE}}\\ (\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ (\mathbf{m}_0,\mathbf{m}_1) \leftarrow \mathcal{A}^{O_{\mathsf{Enc}},O_{\mathsf{Dec}}}(1^\lambda, f, \mathsf{pk})\\ (c^*,\pi^*) \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}_b)\\ \hat{b} \leftarrow \mathcal{A}^{O_{\mathsf{Enc}}}(c^*)\end{array}$$

D Extended Proof: Circuit-Level vFHE

Construction Let us consider the vFHE scheme (Gen, Enc, Eval, Verify, Dec) described in Section 6.1. The construction is as follows: we take an FHE scheme ($\mathcal{E}$.Gen, $\mathcal{E}$.Enc, $\mathcal{E}$.Eval, $\mathcal{E}$.Dec) and a remote attestation scheme (Π.Gen, Π.Prove, Π.Verify).
We then have

- $\mathsf{Gen}(1^\lambda) \to ((\mathsf{pk}_\mathcal{E},\mathsf{sk}_\mathcal{E}),(\mathsf{pk}_\Pi,\mathsf{sk}_\Pi))$ with $(\mathsf{pk}_\mathcal{E},\mathsf{sk}_\mathcal{E}) = \mathcal{E}.\mathsf{Gen}(1^\lambda)$ and $(\mathsf{pk}_\Pi,\mathsf{sk}_\Pi) = \Pi.\mathsf{Gen}(1^\lambda)$
- $\mathsf{Enc}(x) \to c_x$ with $c_x = \mathcal{E}.\mathsf{Enc}(x)$
- $\mathsf{Eval}(c_x, f) \to (c_y,\pi_y)$ with $(c_y,\pi_y) = \Pi.\mathsf{Prove}(c_x, \mathcal{E}.\mathsf{Eval}(\,.\,,f))$
- $\mathsf{Verify}(c_y,c_x,\pi_y) \to b$ with $b = \Pi.\mathsf{Verify}(c_y,c_x,\pi_y)$.
- $\mathsf{Dec}(c_y) \to y$ with $y = \mathcal{E}.\mathsf{Dec}(c_y)$

Correctness Correctness of Dec_{sk} follows immediately from the correctness of the underlying FHE scheme. Specifically, we have that

$$\Pr\left[\mathcal{E}.\mathsf{Dec}_{\mathsf{sk}}(c_y) = f(x) \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)\\ c_x \leftarrow \mathcal{E}.\mathsf{Enc}_{\mathsf{key}}(x)\\ c_y \leftarrow \mathcal{E}.\mathsf{Eval}_{\mathsf{key}}(c_x,f)\end{array}\right] = 1.$$

and so, rewriting the terms, it follows that

$$\mathsf{Dec}_{sk}(c_y) = \mathcal{E}.\mathsf{Dec}_{sk}(c_y) = f(x)$$

with probability 1.

Completeness Completeness of Verify_{sk} follows immediately from the completeness of the underlying remote attestation scheme. Specifically, we have that

$$\Pr\left[\Pi.\mathsf{Verify}_{\mathsf{sk}}(y,x,\pi_y) = \mathtt{true} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \Pi.\mathsf{Gen}(1^\lambda)\\ (y,\pi_y) \leftarrow \Pi.\mathsf{Prove}_{sk}(x,g)\end{array}\right] = 1$$

and so, rewriting the terms, it follows that $\mathsf{Verify}_{\mathsf{sk}}(c_y,c_x,\pi_y) = \Pi.\mathsf{Verify}_{\mathsf{sk}}(c_y,c_x,\pi_y) = \mathtt{true}$ with probability 1.

Security We prove the security of the scheme via a reduction to the security of the underlying FHE scheme. Consider a probabilistic polynomial time (PPT) attacker $\mathcal{A}$ with oracle access to Enc and Dec that breaks the semantic security of our vFHE scheme. Formally, this means that $\mathcal{A}$ can produce two messages m_0 and m_1 such that it can distinguish with non-negligible advantage between $\mathsf{Enc}(m_0)$ and $\mathsf{Enc}(m_1)$. We show that we can build a PPT attacker $\mathcal{B}$ with only oracle access to $\mathcal{E}$.Enc that runs $\mathcal{A}$ to contradict the semantic security of the FHE scheme with the same advantage.

It is sufficient to show that $\mathcal{B}$ can emulate a Dec oracle for the vFHE scheme without having access to a $\mathcal{E}$.Dec oracle for the underlying FHE scheme. $\mathcal{B}$ emulates the Dec oracle by executing the following sequence of steps:

(1) For each query to Enc issued by $\mathcal{A}$, $\mathcal{B}$ makes an oracle request to its $\mathcal{E}$.Enc oracle and stores the latest x used and the c_x obtained. It response to $\mathcal{A}$ with c_x.
(2) Then, when $\mathcal{A}$ sends an oracle request to O_{Dec} with a ciphertext c_y (along with π_y), $\mathcal{B}$ runs $\Pi.\mathsf{Verify}(c_y,c_x,\pi_y)$ (which is a polynomial algorithm) by using the latest c_x it stored in the previous step.
   - If $\Pi.\mathsf{Verify}(c_y,c_x,\pi_y) = \mathtt{true}$, then $\mathcal{B}$ uses the stored value of x to compute $f(x)$ (f is a finite circuit) and sends it to $\mathcal{A}$.
   - Otherwise, $\mathcal{B}$ responds with $\perp$.
(3) When $\mathcal{A}$ sends the challenge messages (m_0,m_1), $\mathcal{B}$ forwards them to its own challenger as the challenge tuple.
(4) After the challenge, $\mathcal{B}$ continues to respond to the Enc oracle as before.
(5) Eventually, $\mathcal{B}$ outputs as $\mathcal{A}$ does.

Because the underlying remote attestation scheme is sound, $\Pi.\mathsf{Verify}$ will only return $\mathtt{true}$ (with overwhelming probability) if $c_y = \mathcal{E}.\mathsf{Eval}(c_x,f)$, and because the underlying FHE scheme is correct, $\mathsf{Dec}(c_y) = f(x)$ with high probability. As a result, O_{Dec} is exactly a Dec oracle for the vFHE scheme. That means $\mathcal{B}$ can successfully emulate the required oracles and use $\mathcal{A}$ to distinguish with non-negligible advantage between $\mathsf{Enc}(m_0)$ and $\mathsf{Enc}(m_1)$.

This contradicts the security assumption of the underlying FHE scheme, concluding the proof.

Soundness We now turn to analyzing the soundness property. We prove the soundness of the scheme via a reduction to the soundness of the underlying remote attestation scheme. We make use of a hybrid argument (changes are highlighted).

- **H0:** For any PPT adversary $\mathcal{A}$ and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\mathsf{Verify}_{\mathsf{sk}}(c_y,c_x,\pi_y) = 1\\ \wedge\\ c_y \neq \mathcal{E}.\mathsf{Eval}(c_x,f)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}^{O_{\mathsf{Enc}},O_{\mathsf{Dec}}(\mathsf{pk})}\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y,\pi_y) \leftarrow \mathcal{A}^{O_{\mathsf{Enc}},O_{\mathsf{Dec}}}(c_x)\end{array}\right].$$

- **H1:** For any PPT adversary $\mathcal{A}$ and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\mathsf{Verify}_{\mathsf{sk}}(c_y,c_x,\pi_y) = 1\\ \wedge\\ c_y \neq \mathcal{E}.\mathit{Eval}(c_x,f)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}^{O_{\mathsf{Dec}}}(\mathsf{pk})\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y,\pi_y) \leftarrow \mathcal{A}^{O_{\mathsf{Dec}}}(c_x)\end{array}\right].$$

- **H2:** For any PPT adversary $\mathcal{A}$ and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\mathsf{Verify}_{\mathsf{sk}}(c_y,c_x,\pi_y) = 1\\ \wedge\\ c_y \neq \mathcal{E}.\mathit{Eval}(c_x,f)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}(\mathsf{pk})\\ c_x \leftarrow \mathsf{Enc}_{\mathsf{key}}(x)\\ (c_y,\pi_y) \leftarrow \mathcal{A}(c_x)\end{array}\right].$$

- **H3:** For any PPT adversary $\mathcal{A}$ and any function g the following probability is negligible in the security parameter λ:

$$\Pr\left[\begin{array}{c}\Pi.\mathsf{Verify}_{\mathsf{sk}}(y,x,\pi_y) = 1\\ \wedge\\ y \neq g(x)\end{array} \;\middle|\; \begin{array}{l}(\mathsf{pk},\mathsf{sk}) \leftarrow \Pi.\mathsf{Gen}(1^\lambda)\\ x \leftarrow \mathcal{A}(\mathsf{pk})\\ \text{x is in the domain of g}\\ (y,\pi_y) \leftarrow \mathcal{A}(\mathsf{pk},x)\end{array}\right].$$

H0 → H1: Let us assume there exists an attacker $\mathcal{A}$ with oracle access to, Enc, and Dec that can win the challenge described in H1 with non-negligible probability. Because the attacker in H1 has access to the FHE scheme public key, it can emulate O_{Enc} and leverage $\mathcal{A}$ to win H2 with non-negligible probability.

H1 → H2: Let us assume there exists an attacker $\mathcal{A}$ with oracle access to Enc and Dec that can win the challenge described in H1 with non-negligible probability. Similarly to how we proved the security for our vFHE scheme, the attacker in H2 can emulate O_{Dec} and leverage $\mathcal{A}$ to win H2 with non-negligible probability.

H2 → H3: H3 is exactly H2 after rewriting the terms using the notation from the remote attestation scheme, and choosing a specific family of functions for g (functions in the form $\mathcal{E}.Eval(., f)$ for any functions f). Because H3 should hold for any function g, a PPT adversary that can win H2 with non-negligible probability can also win H3 with non-negligible probability.

Because H3 is exactly the soundness property of our underlying remote attestation scheme, this concludes our proof by reduction.

E Security Argument: Auth. PIR

In this section, we sketch how Argos can be used to instantiate authenticated PIR. The first requirement towards building authenticated PIR is extending Argos to support public inputs.

E.1 Extending vFHE for Public Server Inputs

Accepting public inputs. We extend our vFHE scheme to allow public server input w for f as follows. We define a function $h(\cdot)$ that takes w as input and outputs w if it is "well formed" or halts otherwise. In practice, this is equivalent to type-checking w. We now define $\mathcal{E}.\text{Eval}'(c_x, w, h, f)$ that evaluates $\mathcal{E}.\text{Eval}(c_x, f(\cdot, h(w)))$ and return the output.

Authenticating server inputs. To authenticate server input, we assume that the server commits to its public database ahead of time. As in prior work on authenticated PIR, we assume that the client obtains this commitment *out-of-band*. We now define a new family of vFHE schemes defined as (Commit, Gen, Enc, Eval, Verify, Dec), which is augmented with the new Commit algorithm. The server uses Commit to commit to its public input w by computing $Commit(w) \leftarrow C(w)$. Here, C is a commitment scheme (e.g., a Merkle tree) that is *computationally binding* (informally, the probability to find $x \neq x'$ so that $C(x) = C(x')$ is negligible) and because the input is public, we do not require the commitment scheme to be hiding. The server then transmits this commitment to the client (e.g., by posting it to a public bulletin board).

Putting everything together. We define $\Pi.\text{Prove}'$ which emulates $\Pi.\text{Prove}$, but ensures that on top of a commitment to $\mathcal{E}.\text{Eval}(c_x, f(\cdot, \cdot))$, the transcript π_y also contains a commitment to $h(x)$ and $C(w)$. We use $\Pi.\text{Prove}'$ to define our new Eval' function.

- $\text{Eval}'(c_x, f) \rightarrow \Pi.\text{Prove}'(c_x, \mathcal{E}.\text{Eval}'(\ .\ , w, h, f))$.

We also define Verify′ which

(1) takes $C(w)$ as input;
(2) checks that $C(w)$ matches the commitment included in π_y;
(3)
- *if* commitments match, evaluate the original Verify and return the output.
- *else* return `false`.

In summary, our new vFHE scheme is defined by (Commit, Gen, Enc, Eval', Verify', Dec). Given these changes, adapting the vFHE properties is straightforward. Using the binding property of the Commit algorithm, all the security games can be extended to include the initial execution of Commit by the adversary. We omit providing the formal properties and proof, since this extension is straightforward to derive and verify.

E.2 Semi-Honest PIR from FHE

We now consider a simple FHE-based PIR scheme. We note that this approach can be extended to specific PIR schemes (e.g., [4]) in natural ways, but for simplicity we restrict ourselves to the FHE-based approach. More concretely, consider an FHE scheme ($\mathcal{E}$.Gen, $\mathcal{E}$.Enc, $\mathcal{E}$.Eval, $\mathcal{E}$.Dec). We construct our PIR scheme by evaluating (using FHE) the query function $f(\cdot, w)$ that takes an input index x and returns $w[x]$, i.e., the xth value in the server database. We now sketch the two required properties for PIR schemes: correctness and privacy. These will help define authenticated PIR in the next section.

Correctness. Informally, a PIR scheme is *correct* if, when an honest client interacts with honest servers, the client always retrieves $w[x]$ (i.e., the xth value in the database w).

Privacy. A PIR scheme satisfies *privacy* if an honest-but-curious server does not learn information regarding x.

Security Argument (folklore). It is trivial to check that the correctness and security of the PIR scheme constructed by having the server evaluate the circuit that computes $w[x]$ given an encryption of x under FHE, reduces to the correctness and security of the underlying FHE scheme.

E.3 Authenticated PIR from vFHE

We now turn to constructing authenticated PIR from verifiable FHE.

Construction. We build an authenticated PIR scheme by using the same construction of PIR described above, but swapping out the FHE scheme with our vFHE scheme that supports authenticated public server input (see Section E.1).

Authenticated PIR properties For authenticated PIR, we sketch the properties from Colombo et. al. [41]. In the case of authenticated PIR, the security property is extended to support a malicious server, and a new integrity property is added. Correctness is the same.

Integrity. An authenticated-PIR scheme preserves *integrity* if, when an honest client interacts with a malicious server, the client either: outputs $w[x]$ if the server followed the protocol or outputs the error symbol $\perp$ if the server deviated from the protocol.

Privacy (against malicious servers). An authenticated PIR scheme satisfies *privacy* if a malicious server does not learn information

regarding x, even if the servers learn whether the client's output was the error symbol $\perp$ during reconstruction.

Security Argument. We now briefly argue why the scheme using vFHE with authenticated public server input satisfies these properties. Correctness follows from the correctness of the PIR scheme. Integrity reduces to the soundness of the underlying vFHE scheme, while security reduces to the underlying CCA1 security of the vFHE scheme. We leave out the detailed proofs.

E.4 Extensions to More Complex PIR Schemes

Most FHE-based PIR schemes are not implemented as a simple FHE circuit and utilize other techniques to optimize performance (e.g., cuckoo hashing). These schemes fit into our formalism by extending $h(x)$ to also contain any server input reprocessing. Because h is also part of the attested transcript, the resulting PIR scheme run using Argos is verifiable (and authenticated).

F Security Argument: Authenticated PSI

We now turn to our second application: authenticated PSI. Similarly to FHE-based PIR schemes, FHE-based PSI schemes can also be formalized as the evaluation of a specific circuit under FHE, albeit with a more complex pre-processing of the server input (in particular to enforce *server privacy*). This fits our formalism by extending $h(x)$ to contain this extra pre-processing of server input. Similarly to authenticated PIR, we could show that our vFHE scheme is sufficient to enforce analogous properties for authenticated PSI (correctness, integrity, client privacy, even if PIR security is often defined using an ideal functionality). We omit these proofs in the interest of space. Instead, we focus on the main difference with PIR, which is "server privacy."

Server Privacy. Informally, for a given x, for two server inputs w and w' that result in the same output y, there should be negligible advantage for a PPT adversary to distinguish between (c_y, π_y) and (c'_y, π'_y).

Because most FHE schemes are not circuit-private, the output c_y can leak some information about the server's input to the secret key holder. To achieve server privacy with regard to c_y, FHE-based PSI schemes pre-process and mask server input using an oblivious pseudo-random function (OPRF) [34]. As explained earlier, this pre-processing is captured by $h(\cdot)$. This construction was shown by previous work to be *server private* with regard to c_y. Because our vFHE scheme does not modify the underlying FHE scheme or PSI scheme, we inherit this property and only need to ensure that π_y does not breach the privacy of the server. Given our construction, it is sufficient for us to choose a commitment scheme C that is *hiding*. Informally, π_y contains a commitment to $\mathcal{E}.\text{Eval}(c_x, f(\cdot,\cdot))$ and $h(x)$, which are independent of w, and $C(w)$, which is hiding. As a result, our construction is *server private* with regard to (c_y, π_y) and combining a PSI scheme with Argos successfully instantiates an authenticated PSI scheme.