Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy

Lauren Biernacki
University of Michigan

Meron Zerihun Demissie
University of Michigan

Kidus Birkayehu Workneh
University of Colorado Boulder

Fitsum Assamnew Andargie
Addis Ababa University

Todd Austin
University of Michigan

***Abstract*—Data breaches that penetrate web-facing servers and exfiltrate sensitive user data have become pervasive. Insulating these systems from attack is seemingly impossible due to the ubiquity of software vulnerabilities within cloud applications. It is simply insurmountable to adequately address all such vulnerabilities, and therefore imprudent to rely on software applications to protect user data. Rather, the ideal systems solution upholds data confidentiality, even in the presence of vulnerable or compromised software. Homomorphic encryption (HE) provides these capabilities, but its limited expressiveness and significant runtime overheads have inhibited its adoption. In this work, we explore how trusted hardware can be leveraged to provide data confidentiality in the presence of vulnerable software while achieving practical performance overheads. We present Sequestered Encryption (SE)—a hardware technique for data privacy that sequesters sensitive plaintext data into a small hardware root of trust and encrypts this data in all external microarchitectural structures, thereby rendering secret values inaccessible to software. With optimizations, SE achieves <2.5× performance slowdowns on average compared to native execution, demonstrating that architectural approaches can emerge as data privacy solutions that possess zero trust in software while being dynamic, expressive, and performant.**

I. Introduction

The task of securing private data is one of the great challenges of modern information systems. Security vulnerabilities are expected to be present in complex systems software, yet such systems are trusted to process sensitive user data. As a result, data breaches have become pervasive. In 2020, IBM estimated that public web servers have a 1-in-4 chance of being penetrated in any given year, with each breach costing an average US$3M to recover [33]. This, coupled with recent advances in data exfiltration (*e.g.*, Spectre [35]), has brought data privacy to the forefront of system design concerns.

Insulating complex systems and cloud applications from attack is intractable due to the pervasiveness of software vulnerabilities. It is simply insurmountable to adequately address all such vulnerabilities, and therefore imprudent to rely on software to protect user data. Rather, the ideal systems solution ***upholds data confidentiality, even in the presence of vulnerable or compromised software***. Homomorphic encryption (HE) is the only present technique that achieves this goal by imbuing systems with the ability to compute directly on ciphertext values. These cryptographic schemes behave as homomorphisms between the plaintext and ciphertext spaces, preserving operations performed on ciphertext values after decryption (*e.g.*, $\mathsf{ENC}(x)+\mathsf{ENC}(y)=\mathsf{ENC}(x+y)$). An application that leverages HE can compute on data without needing access to encryption keys or sensitive plaintext values. Thus, even if the application is compromised, its data will not be exposed, as plaintext values do not exist within the system. While HE has made breakthroughs in providing comprehensive data privacy, its limited multiplicative depth has remained the main practical limitation, bounding the scheme's expressiveness, exacerbating performance overheads, and ultimately inhibiting adoption.

The escalation of data breaches and shortcomings of HE warrant alternative solutions for providing comprehensive data privacy. Trusted hardware is uniquely positioned to administer data privacy, especially in the presence of vulnerable software, as it underlies all software abstractions and can be applied systematically, regardless of program-level semantics. Furthermore, hardware techniques have seen widespread adoption for providing secure key storage, isolated execution, and code integrity (*e.g.*, Intel SGX [22], [56]), demonstrating a willingness to place trust in hardware.[1]

In this work, we explore how trusted hardware can provide data confidentiality in the presence of vulnerable software while achieving practical performance overheads. We introduce ***Sequestered Encryption (SE)***—a hardware capability that computes directly on encrypted data in the microarchitecture while isolating related sensitive plaintext values from software. Specifically, SE ***sequesters*** sensitive plaintext data into a small trusted hardware unit (termed the SE unit) and, in turn, ***encrypts*** this sensitive data everywhere outside the SE unit, including in the register file and other microarchitecture structures. With this approach, ***no sensitive plaintext values exist in any software-accessible state***, significantly reducing the attack surface. Further, this approach drastically shrinks the trusted computing base compared to other hardware defenses since users only need to trust the small SE unit hardware, which lends itself to formal verification.

With our proposed architectural optimizations, we show that SE is able to achieve performance that is within 154% (2.5×) of native execution when using AES-128 cipher engines, with 73% of this overhead due to cryptographic latencies and the remainder (81%) due to software overheads (*e.g.*, data-oblivious programming, 128-bit data loads). Lightweight ciphers, like QARMA, have the potential to further reduce overheads to less than 99% (2×) of native execution. Comparisons to HE platforms underscore the value of hardware-based secure computation. We show SE to be faster than Microsoft SEAL [57] by three orders of magnitude over the same workloads.

Use Cases. SE is designed to provide confidentiality for data processed in an untrusted or potentially vulnerable environment. Thus, SE is suited for cloud services processing third-party data on a public web server or another untrusted cloud-based platform, or in an environment co-located with untrusted processes. In these settings,

[1]Trusted execution environments (TEEs) do not guarantee data privacy in the presence of vulnerable software, as they typically trust some software components and exclude timing side channels from their treat models. These choices leave them vulnerable to attack [11], [14], [37], [63].

third-party data can be encrypted in a trusted, client-side environment and offloaded to the server for computation.

Contributions. We believe that architectural approaches can emerge as comprehensive data privacy solutions that possess zero software trust while being dynamic, expressive, and performant. In support of this argument, we make the following contributions:

- We present Sequestered Encryption (SE)—a hardware capability that provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state.
- We design and evaluate two microarchitectural implementations of SE—a stateless variant (SE) and an optimized, stateful variant (SE-OPT)—and detail the trade-offs associated with these optimizations.
- We demonstrate that our solution is expressive by evaluating a diverse set of 21 workloads in our analysis.
- We provide a detailed evaluation of the performance overheads of our designs. We find that encryption latencies have a near-linear impact on the overheads of SE, whereas SE-OPT can tolerate strong ciphers like AES-128 with moderate overheads (<2.5× native execution).
- We compare the performance of SE to SEAL homomorphic encryption and find that SE offers a performance benefit by three orders of magnitude.
- We demonstrate that our proposed SE design is resilient to cryptanalysis and timing side channels.

In Section II, we introduce Sequestered Encryption (SE) and describe our security goals and threat model. In Section III and IV, we present the architecture and microarchitecture of SE and detail optimizations that enable performance. We then discuss the software interfaces in Section V before presenting our evaluation in Section VI. Finally, we discuss related works in Section VII and conclude in Section VIII.

II. SEQUESTERED ENCRYPTION (SE)

Sequestered Encryption (SE) is a hardware capability that computes directly on encrypted data in the microarchitecture while isolating related sensitive plaintext values from software. Specifically, SE *sequesters* sensitive plaintext data into a small trusted hardware unit (termed the SE unit) and, in turn, *encrypts* this sensitive data everywhere outside the SE unit, including in the register file and other microarchitecture structures. With this approach, no sensitive plaintext values exist in *any* software-accessible architectural state, significantly reducing the attack surface. We assume that the data encryption scheme ensures the confidentiality of ciphertext values stored outside of the SE unit.[2] Under this assumption, ciphertexts can persist in unverified or leaky hardware structures at no risk to data confidentiality. Decryption capabilities and keys are isolated within the SE unit, permitting the unit to process encrypted values while ensuring that the SE unit is the only hardware structure that can access sensitive plaintext values via decryption.

Our architecture design (§III) implements SE as a small hardware *functional unit* embedded within the pipeline. As a result of this design, operations can be dispatched to the SE unit on a per-instruction basis, enabling tight interleaving of secret and non-secret computation and minimizing overheads. Our microarchitecture design (§IV) ensures that *i)* no sensitive plaintexts within the SE unit are exposed outside of it, and *ii)* any operation (or sequence of operations) performed within the SE unit is free of timing side channels. Conceptually, the simplest design of the SE unit is an arithmetic logic unit (ALU) fitted with cryptographic engines, as shown in Figure 1. When instructions are dispatched to this unit, the input operands are decrypted, a single operation is computed (*e.g.,* addition), and the result is re-encrypted between two latches. While this design is uncomplicated, encryption latencies exist on the critical path, warranting performance optimizations. In this work, we present and evaluate two hardware implementations of SE: ***a stateless atomic implementation*** (Figure 1) and ***a stateful optimized implementation***. Below, we discuss the security goals that guide our design and optimization of SE before discussing our implementation in Sections III and IV.

[2]The data encryption scheme is parameterizable and can be chosen to match the performance-security needs of the system.

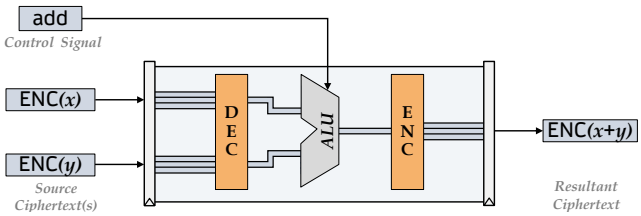


Fig. 1: **Simple Design of Sequestered Encryption Unit.** A simple implementation of SE is an arithmetic logic unit (ALU) fitted with cryptographic engines. Data decryption, computation, and re-encryption are done as one atomic operation between two latches. The computation is selected by a public control signal.

A. Security Goals

Our goal is to maintain the confidentiality of a user's secret data in the presence of untrusted software. For clarity, we refer to this user as the ***data owner*** in the remainder of this text. In our intended use case, this data owner is the client offloading their data to a server for computation. Secret data is the plaintext value of any program variable explicitly denoted as "Private" and encrypted by the data owner. All other program data is assumed to be non-secret, including code contents and ciphertext values, as we assume that the adversary can easily infiltrate the software processing the data (§II-C). Specifically, we adopt the following definitions:

> ***Secret Data:*** Plaintext values of program variables denoted as Private
>
> ***Non-Secret Data:*** Ciphertext values of program variables denoted as Private; Plaintext values of program variables not denoted as Private; Constant values and instruction immediates; Instructions and instruction properties (*e.g.*, data dependencies, instruction addresses)

Our security goal aims to maintain the confidentiality of secret data on the server under two potential threats: ***secret data disclosure*** and ***secret-data-dependent timing***. To insulate against the former, the plaintext values of secret data can never be exposed outside of the SE unit or downgraded to non-secret. Only the data owner can decrypt and release their data after receiving a response from the server. To protect against the latter, the timing of the SE unit must be independent of the plaintext values of secret data. Together, these goals prevent adversaries from directly learning secret data or inferring it via timing, control, or memory side channels.

B. Root of Trust

To be assured of the confidentiality of their secret data, the data owner must trust that the SE unit hardware is implemented in a secure

and bug-free manner. The data owner must also trust attestation and key exchange with the SE hardware located on the server. However, the data owner does not need to trust any hardware outside of the SE unit (*e.g.*, speculation logic, memory, *etc.*) or any software running on the SE device, including the operating system.

SE is unique in that its trusted computing base (TCB) is small and contains *no* software. Additionally, trust in the SE hardware can readily be established as SE's compact design lends itself to formal design validation.

C. Attacker Capability

We assume a software-based adversary that can exploit the vulnerabilities expected in systems software to gain full system access. In particular, we assume that an attacker can compromise any operating system or hypervisor present on the machine and launch malicious code in user or privileged mode. The attacker has complete knowledge of the machine's software, architecture, and microarchitecture, including the SE enclave, but does not have knowledge of secret key information. The attacker can passively analyze system data, directly access register and memory values, and mount active attacks (*e.g.,* install software to load and compute directly on secret data). The attacker can also exploit precise hardware timers, speculative state, and shared microarchitectural state that is accessible via software-based attacks (*e.g.,* Spectre [35]).

Within the context of our usage model, this threat model considers a cloud service that is actively malicious, is compromised by a remote attacker, or is hosted by a malicious administrator or service provider. The goal of this adversary is to steal secret data that is present on the device.

Non-Goals. We assume the adversary does not have physical access to the machine and therefore cannot leverage physical attacks, physical side channels (*e.g.*, power), or transistor-scale voltage measurement (*e.g.*, focused ion beam sensing). Additionally, we assume that the SE unit hardware is implemented correctly and is free of hardware Trojans. SE does not protect the integrity of code or memory since we assume our attacker has unlimited ability to manipulate computation. Finally, we exclude attacks that target data not denoted as private and not encrypted by the data owner.

D. Novelty of Proposed Approach

SE is a novel hardware technique characterized by its comprehensive use of encryption and minimal trusted computing base (TCB). Hardware defense techniques have mainly employed permissions checking (*e.g.*, Intel SGX [56]), physical isolation (*e.g.*, Ascend [27]), and isolation through the flushing and partitioning of shared structures (*e.g.*, MI6 [13]). Some hardware defenses do use encryption to protect program data, code, or pointers. However, these approaches lack comprehensiveness, as they decrypt values at the processor-cache boundary [24], [38], with many stopping further down the memory hierarchy [2], [28], [56]. For example, while Intel SGX does use memory encryption, it sits at the edge of the on-chip memory hierarchy and, therefore, is unable to protect enclave memory from software attacks [22].

SE's compact functional unit design results in a significant reduction in the overall TCB compared to other hardware defense mechanisms (§II-B). This minimal design lends itself to formal verification. As noted by prior work [22], formally reasoning about enclaves like Intel SGX is presumably infeasible as any proof would have to model all processor features that expose registers. Further, such work would be short-lived as any architectural modifications would invalidate this security proof. In contrast, SE's compact design is independent of many pipeline structures, avoiding these deficiencies.

While SE leverages similar defense mechanisms to the hardware techniques mentioned above, its security goals and threat model are significantly different. Namely, trusted execution environments (TEEs) and other secure enclaves are designed to maintain program integrity, often disregard side channels [14], [16], [22], and assume either a trusted operating system or application software [11], [37]. SE focuses solely on guaranteeing data confidentiality from both direct disclosures and timing side channels in a wholly untrusted software environment. For these reasons, the goals and threat model of SE are more closely related to homomorphic encryption (HE). HE has a smaller TCB than SE, since HE trusts only the cryptography itself. However, SE trades this trust in the SE hardware to build a solution that is more dynamic, expressive, and performant (§VI-A), addressing many of the core issues that have inhibited HE's adoption [1], [46].

III. Sequestered Encryption Architecture

Sequestered Encryption (SE) is a hardware capability that protects the confidentiality of secret data by sequestering all sensitive plaintexts into the trusted SE unit and encrypting these values in all other hardware structures. SE is architected as a small hardware functional unit embedded within the execute state of the pipeline. The interfaces to the SE unit are implemented as an instruction set extension such that all outputs are encrypted and all timing is independent of secret data, thereby thwarting attacks that attempt to leak or infer sensitive plaintext values. In this section, we discuss the architecture of the SE unit—its functionality and instruction-level interfaces. In Section IV, we detail the physical components that implement these architectural semantics.

A. Architectural Implementation

We implement SE as a hardware functional unit embedded within the execute stage of the x86 pipeline. The SE unit is commanded via an extension to the x86-64 instruction set architecture (ISA), summarized in Table I. When the processor decodes an instruction from the SE ISA extension, the instruction is dispatched to the SE unit for processing. The instruction opcode indicates the format of the source operands (*e.g.*, ciphertext-plaintext) and the operation for the SE unit to perform (*e.g.*, addition). This process mirrors how native x86-64 instructions are dispatched to functional units in the execute stage of the pipeline.

The processor and its software perceive the SE unit as a black box that receives control signals and data inputs and produces a result. Like other pipeline functional units (*e.g.*, the ALU), software is unable to access any microarchitectural state within the SE unit but can read the SE unit's inputs and outputs and time its overall latency. By designing the SE microarchitecture to encrypt all outputs and have deterministic timing (§IV), we ensure that any software-observable values do not compromise the confidentiality of the secret data processed within the SE unit.

B. Source Operands

The SE unit accepts source operands in multiple formats, as the SE ISA extension supports ciphertext-ciphertext, ciphertext-plaintext, and ciphertext-immediate inputs. Ciphertext inputs are expected to have the format illustrated in Figure 2. Namely, each ciphertext is constructed from an 8B secret data word appended with a unique 8B random cryptographic salt, which serves to diversify the ciphertext and thwart cryptanalysis attacks. The resultant 16B (128-bit) value is then encrypted under the symmetric encryption scheme supported by

the SE unit. Any secret data values that are shorter in length (*e.g.*, 4B integers) are padded accordingly.

When the SE unit receives a ciphertext source operand, the value is decrypted (inversely to Figure 2) and the random salt is discarded. The computation proceeds on the plaintext secret data word. The result is then formatted and encrypted using a fresh salt value before being emitted from the SE unit.

The data encryption scheme is parameterizable and can be chosen to match the performance-security needs of the system. In this work, we evaluate several encryption schemes that represent varying latencies and security: AES-128 [5], Simon-128/128 [6], and QARMA_{11}-128-σ_1 [5]. Data owners must prepare their sensitive inputs to match this data encryption scheme. Software support for writing programs within SE is discussed in Section V.

Instruction Class	Example	Semantics
Arithmetic	`enc_add x, y, z`	`x = enc(dec(y) + dec(z))`
Relational	`enc_sleq x, y, z`	`x = enc(dec(y) ≤ dec(z))`
Boolean	`enc_and x, y, z`	`x = enc(dec(y) & dec(z))`
Conditional Move	`enc_cmov x, y, z`	`x = enc(dec(y) ? dec(x) : dec(z))`
Memory Access	`enc_mov [x] → y`	`mov [x]→y; mov [x+0x8]→ y`

TABLE I: **Sequestered Encryption ISA Extension.** The x86-64 ISA extension facilitates encrypted computation of arithmetic, relational, and Boolean operators and conditional moves within the SE unit. Loads and stores of encrypted values (`enc_mov`) are decomposed into two 64-bit `mov` micro-ops, as discussed in Section III-C.

C. Control Interface (ISA Extension)

The processor explicitly commands the SE unit via its defined control interface—the x86-64 ISA extension summarized in Table I. The control signals provided to the SE unit are non-secret instruction opcodes that indicate the format of the source operands and the operation to compute. Below, we discuss the instruction classes that constitute the SE ISA extension.

Loading and Storing Encrypted Data. Encrypted data is loaded from memory for processing via an `enc_mov` instruction. This instruction is handled by the main processor's memory interface. This operation is modified slightly because the encrypted data is 128-bits long, whereas our modeled memory system accommodates 64-bit data paths. Rather than modifying the processor's data paths, we decode the `enc_mov` instruction into two 64-bit `mov` micro-ops to jointly load the entire 128-bit value into the register file. The register file is extended to 128-bits to accommodate these extended ciphertexts. Encrypted store instructions proceed equivalently. The operation of x86-64 load/store instructions is unaffected by these changes, as the lower 64-bits of the register file can be accessed independently and the data path is unmodified. These `enc_mov` commands do not decrypt data, as sensitive data in the register file always remains encrypted.

Operating on Encrypted Data. Once ciphertexts are loaded into registers, they are ready for processing by the SE instructions. The SE unit supports arithmetic (integer and floating-point), Boolean, and relational instructions, as listed in Table I. Each instruction must have deterministic (data-independent) timing to uphold our security goals. Thus, we prohibit hardware optimizations within the SE unit that accelerate instructions for specific inputs (*e.g.*, early termination for multiply-by-zero [32]), as these optimizations leak information about sensitive plaintext values via timing. Different SE instructions still have different latencies within the SE unit (*e.g.*, `enc_add` takes one cycle, whereas `enc_mul` takes three cycles), as these timing differences depend solely on non-secret instruction opcodes rather than secret data inputs.

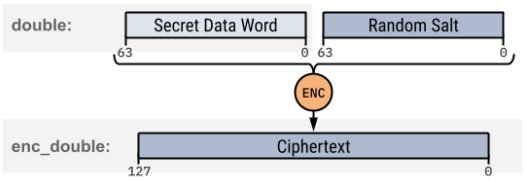


Fig. 2: **Encrypted Data Layout.** Primitive types (*e.g.*, `double`) are appended with an 8B random cryptographic salt and then encrypted to form a 128-bit encrypted data type (*e.g.*, `enc_double`).

Making Decisions on Encrypted Data. The SE architecture supports conditional move (`enc_cmov`) operations, akin to the x86-64 `CMOVcc` primitive. Conditional moves function as ternary operators, where a destination register is updated depending on the value of a secret condition. This primitive enables programs to make decisions on encrypted conditions without leaking information via timing or exposed state, as described further in Section V.

D. A Subtractive Approach to Enforcing our Security Goals

The SE ISA extension and its semantics are designed such that it only supports *oblivious computation*. Also called "constant time" or "data-oblivious" programming, this technique avoids side channels by ensuring that program execution is independent of secret data [26], [49], [55], [65], thereby enforcing our security goals (§II-A).

In SE, we achieve oblivious computation by *i)* isolating decryption capabilities and keys within the SE unit, and *ii)* only implementing oblivious operations within the SE unit. Traditionally, oblivious computation is difficult to achieve because many non-oblivious instructions exist in the native x86-64 instruction set (*e.g.*, conditional branches). However, the functional units that compute these native instructions are unable to access secret data via decryption since decryption capabilities are isolated within the SE unit. Thus, any native, non-oblivious instructions are performed on *ciphertext values*, producing garbled results and leaking no sensitive information. The SE unit, on the other hand, is capable of decrypting values and accessing sensitive plaintexts. Thus, to enforce oblivious computation, we *only* implement oblivious operations within the SE unit—arithmetic, relational, and Boolean instructions and conditional moves. The SE hardware simply *cannot* process conditional branches or instructions accessing encrypted memory addresses.[3] Together, this design forces oblivious computation on secret data at the hardware level.

We discuss the program transformations required to support oblivious computation in Section V. Programs can still perform non-oblivious operations on non-secret data, reducing the required program transformations and improving performance. Finally, we note that oblivious computation is required for *all* secret computation frameworks, including HE, as secret-data-independent control flow and program execution is fundamental to preserving confidentiality.

IV. Sequestered Encryption Microarchitecture

In Section III, we broadly discussed the Sequestered Encryption (SE) architecture, which is implemented as an extension to the x86-64 ISA. In this section, we present our implementation of the SE unit microarchitecture—the internals of the SE unit that implements the interfaces and semantics of the SE architecture. To uphold our

[3]Alternatively, one could permit encrypted address operands by using Oblivious RAM (ORAM) [58] to obfuscate memory access patterns.

security goals, we design our microarchitecture such that *i)* no secret plaintexts within the SE unit are exposed outside of it, and *ii)* all computation within the SE unit has deterministic timing.

Many different microarchitecture designs could implement the SE architecture and meet these goals. In this work, we present and evaluate two hardware implementations of SE: ***a stateless atomic implementation***, denoted **SE**, and ***a stateful optimized implementation***, denoted **SE-OPT**. SE-OPT is an iteration on the stateless design that serves to reduce performance overheads by masking the encryption and decryption latencies within the SE unit. Below, we present these two designs and discuss the tradeoffs of optimization (§IV-A-IV-C). Then, we discuss the surrounding infrastructure that is synonymous with both designs (§IV-D-IV-E).

A. Stateless Microarchitecture

As presented in Figure 1, our simplest design of the SE unit is a functional unit bookended with cryptographic engines. The cryptographic engines and computational hardware sit between two latches, which provide the entry and exit point for the SE unit data path. We outfit the SE functional unit with an arithmetic logic unit (ALU) and a floating-point unit (FPU) to implement the semantics of the SE ISA extension. Furthermore, we pipeline both the cryptographic engines and computational hardware to improve throughput and reduce performance impacts on overall runtime.

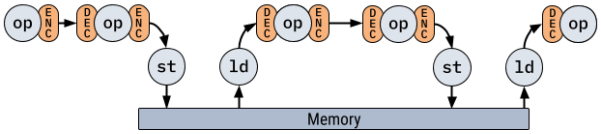
The SE unit can be extended to meet the computational requirements of the native processor. For example, if the processor supports single-instruction multiple-data (SIMD) operations, the SE unit could be built to include SIMD computational capabilities. Additionally, the SE unit could include multiple cipher engines or computational units to further increase throughput (*i.e.*, a superscalar implementation). In this paper, we evaluate an SE unit with a single instance of each component, which we regard as the most conservative design.

Walkthrough. The stateless miroarchitecture is shown in Figure 1. When a ciphertext value enters the SE unit, it is fed to the decryptor, which decrypts the source operands and discards the salt. Then, the plaintext values are forwarded to the computation hardware, which operates on them as indicated by the public control signal (*e.g.,* `enc_add`). When the computation completes, the encryptor re-encrypts the result with a new salt value. This final ciphertext is fed to the latch exiting the SE unit. This path through the encryptor and corresponding latch is the only exit point from the SE unit.

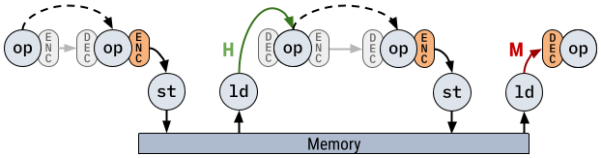
B. Optimized (Stateful) Microarchitecture

While the design of the stateless microarchitecture is uncomplicated, cryptographic operations on the critical path significantly increase the overhead of SE instructions. For example, with QARMA's 20-cycle cipher latency, an add instruction takes 41 cycles to compute in SE [5]. In this section, we present optimizations to the SE design that sufficiently mask these cipher latencies.

To guide our optimizations, we examine the data flow for a sequence of dependent SE instructions, shown in Figure 3. When data dependencies occur, the result of the first operation is encrypted then immediately decrypted for use in the second operation. These encrypt-decrypt operations exist on the critical path for neighboring instructions, bloating performance. Further, we expect these dependencies to occur frequently due to locality in the register file. In SE-OPT, we bypass these idempotent operations by forwarding previously-computed plaintext values to subsequent dependent instructions within the SE unit. Such forwarding will introduce timing differences in the SE design, but these differences do not violate our security goals if they are not dependent on secret data.



(a) **Data Flow for Stateless Design.**



(b) **Data Flow for Optimized (Stateful) Design.**

Fig. 3: **Data Flow for Sequestered Encryption.** In the unoptimized microarchitecture, cryptographic latencies are exposed on every operation. Our optimizations bypass idempotent encrypt-decrypt operations and reduce latencies on the load path.

Specifically, timing differences based on non-secret data—ciphertext values, instruction characteristics, *etc.*—are permitted.

Forwarding Around Decryption. To mask decryption latencies, we extend the SE unit with a small ***decryption cache*** that stores ciphertext-plaintext pairs. Specifically, when an instruction is dispatched to the SE unit, the ciphertext value of a source operand is used to index the decryption cache, which returns the corresponding decrypted plaintext if present. If the decrypted plaintext is not present, the source operand is decrypted in its entirety, as done previously. This design imbues the SE unit with short-term memory of recent decryption operations. Decryption cache latency is dependent on ciphertext values and their reuse, which is non-secret, upholding our security goals related to timing discrepancies.

The decryption cache is updated at two points: *i)* when there is a cache miss and *ii)* when a new value is encrypted by the SE unit. The former optimizes instructions that reuse the same input (`r`) without updating that input (*e.g.*, `enc_add x ←x,r; enc_add y←y,r`). The latter optimizes instructions with true data dependencies (*e.g.*, `enc_add x ←x,r; enc_add y←y,x`). To accelerate instructions that succeed a load, we dispatch the load to the SE unit to prime the decryption cache so that the ciphertext-plaintext pair is available when the subsequent usage occurs.

We implement the decryption cache as a two-way set-associative cache with eight sets and least-recently-used (LRU) replacement. The cache stores 125-bit ciphertext tags and 8B plaintext data blocks. This small <0.4 kB cache can be accessed in a single cycle and is flushed whenever the SE unit performs a key switch.

Forwarding Around Encryption. To further optimize performance, we add state to internally forward values around the encryptor to subsequent SE instructions. This forwarding occurs *within* the SE unit[4] and can save up to 40 cycles of latency between dependent operations (the latency of AES). We extend the SE unit with an additional decryption cache, termed the ***physical register file (PRF) decryption cache***, or register cache, for short. This new cache is indexed by the physical register identifier[5] of the source operand and

[4]All values that leave the SE unit pass through the encryptor, per our security goals.

[5]Physical registers are distinct from architectural registers, which are the register names programs and compilers use.

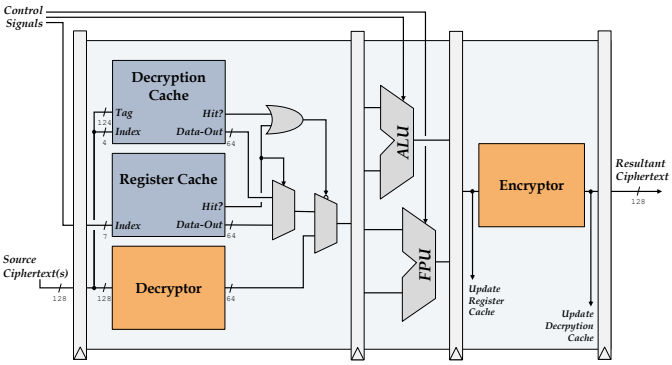


Fig. 4: **Optimized Sequestered Encryption Microarchitecture.** The decryption cache and register cache forward plaintext values to dependent SE instructions, bypassing idempotent cipher operations on the critical path. Since these caches are indexed with non-secret information (*i.e.*, ciphertext values and register IDs), any timing variation due to these optimizations will not reveal any secret data.

stores the corresponding plaintext. This enables forwarding between two instructions that write then read the same physical register while these instructions are both in-flight in the pipelined SE unit. Indexing the cache with the physical register identifier provides two advantages: *i)* it is easy to maintain coherence for the register cache, even during speculation, due to the native design of the PRF, and *ii)* physical register use is non-secret, upholding our security goals related to timing properties.

The register cache is updated whenever a new plaintext result is computed by the SE computation hardware (*i.e.*, ALU or FPU). For example, after the instruction `enc_add p3←p1,p2` is computed in SE's internal ALU, the register cache entry corresponding to physical register `p3` is updated with the plaintext result. On this cycle, the SE unit also broadcasts that the register `p3` is available so that any dependent instructions can be issued to the SE unit.

We implement the register cache as a direct mapped cache with 128 entries (the size of the PRF) and 8B plaintext data blocks. This 1kB cache can be accessed in one cycle and, as for the decryption cache, is flushed on context switches. The register cache has to be coherent with the PRF, as the PRF may be updated outside of the SE unit. To achieve this, the SE unit snoops the PRF and invalidates cache entries when writes occur outside of the SE-unit (*i.e.*, by the main pipeline).

Walkthrough. The optimized miroarchitecture is shown in Figure 4. When an encrypted source operand enters the SE unit, the decryption cache and the register cache are accessed in parallel. On a hit in either cache, the plaintext value is read directly from the corresponding cache, bypassing the decryptor. If both caches miss, the ciphertext is passed to the decryptor, as done in §IV-A. Then, the plaintext result is forwarded to the computation hardware. The caches are updated according to the descriptions above.

Revisiting Figure 3, the dashed lines represent forwarding facilitated by the register cache, which optimizes instructions with true data dependencies. The decryption cache primarily serves to bypass decryption operations for values that are stored to memory and reloaded later. The solid green line represents forwarding facilitated by the decryption cache during a cache hit (H), indicating that the ciphertext has been used recently. The solid red line represents a miss (M), which is likely if there is a long delay between ciphertext accesses.

C. Tradeoffs of Optimization

Our initial microarchitecture design has a minimal TCB, stores no state, and occupies a small hardware area. This uncomplicated design has the potential to simplify formal verification efforts. However, performance overheads of this design may be prohibitive, especially when using long-latency ciphers (*e.g.*, AES-128 has 36.6× slowdowns per Section VI).

Our optimizations hide long-latency ciphers, reducing performance overheads to only 2.5× native execution. However, this performance gain comes at the cost of an increased TCB, the storage of plaintext state, and added hardware area. Because this design stores plaintext state, it is paramount that the SE unit is fully isolated from the main processor and is only accessible via its defined interfaces. Formal verification can provide these guarantees and, while more complicated due to these optimizations, such efforts are still feasible as they only require validation for a subset of the overall pipeline. Finally, these optimizations rely on the assumption that ciphertexts and code characteristics like data dependencies are non-secret.

D. Key Infrastructure

Prior work has refined hardware-based key management to the point where we can integrate these techniques with few modifications (*e.g.*, [4], [62]). To facilitate communication between the data owner and the server-side SE architecture, the SE unit is extended to support a key exchange protocol that establishes a shared data encryption key (k). The SE unit is also defined to contain dedicated, isolated hardware registers to store this key (k) and those used for key exchange (pk_{se}, sk_{se}). These registers are not accessible to software-based addressing and are only read/written implicitly though the SE interface (*e.g.*, `expose_key`). The SE unit's private key (sk_{se}) can be generated in hardware using a memory physically unclonable function (PUF), as done in [30], [31]. Hardware-supported attestation [22], [56] and key exchange allow the data owner to authenticate the SE unit and agree on a shared data key (k). Namely, to perform key exchange using RSA:

1) The data owner acquires the public key (pk_{se}) of the SE hardware from a trusted certificate authority.
2) The data owner generates a symmetric key (k) that will be used as the shared data encryption key. They encrypt k with pk_{se} according to the RSA protocol.
3) The encrypted key packet and encrypted data is transported to the server containing the SE extension.
4) The server OS receives the packet and issues an `expose_key` instruction.
5) `expose_key` is dispatched to the SE unit, which decrypts the key packet using the SE unit's private key, sk_{se}, and stores the result in the isolated key register.
6) Now, protected computation using SE can proceed with the data owner's symmetric key, k.

The semantics of `expose_key` are implemented within the SE unit to uphold the SE security assumptions and have deterministic timing. For RSA, this requires that the exponentiation implementation does not optimize for zero key bits.

E. Process-OS Interface

Applications using SE must still run atop and interact with the operating system (OS) for routine tasks, like context switching and exception handling. SE's key mechanisms enable programs to make these requests without having to trust this complex (and likely vulnerable) systems software to protect data privacy. On a context switch, the OS must swap in the data encryption key of the new

```
void bubbleSort(enc_int *arr, int n){
  for(int i=0;  i<n-1;  i++)
    for(int j=0;  j<n-i-1;  j++) {
      enc_int  tmp  = arr[j];
      enc_bool gt   = arr[j] > arr[j+1];
      arr[j]   = enc_cmov(gt, arr[j+1], tmp);
      arr[j+1] = enc_cmov(gt, tmp, arr[j+1]);}
}
```

Fig. 5: **Sorting a List with Sequestered Encryption.** This program sorts a list of encrypted integers using SE operations, facilitated by our GNU C++ library. Due to the restricted SE environment (§V-B), a conditional move is used in place of an if-statement.

context. To stash the existing data encryption key (k), the OS issues a `conceal_key` instruction, which is the functional inverse of `expose_key`. Specifically, this instruction encrypts k with the SE unit's public key, pk_{se}, and writes the resultant packet to a register. This key packet can then be stashed into storage, permitting multiple contexts to reside at once. We note that this key packet does not pose an additional security risk, since it has already been transmitted during key-exchange. The OS, if uncompromised, will use the `expose_key` and `conceal_key` instructions to perform context switches appropriately. If the OS is malicious, data corruption will occur from key mismanagement, affecting correctness but not violating our confidentiality goals (§II-A).

To prevent exceptions from revealing secret data, exception indications must be embedded inside the encrypted result. For example, a divide-by-zero exception cannot be revealed; otherwise, it would be possible to determine that the confidential denominator is zero. As done in prior work [49], [65], the SE unit masks such exceptions by replacing the plaintext with a canonical value indicating the fault. Subsequent operations propagate this value, thereby allowing the fault to be discovered when the final result is decrypted by the data owner.

V. Programming with Sequestered Encryption

Sequestered Encryption (SE) extends the processor with the capability to recognize secret data and selectively protect the confidentiality of these values. Applications can engage the SE instruction interface to extend these privacy guarantees to their users. To provide access to these data security features, we extend programming environments to support SE semantics, as detailed in this section.

A. Abstractions for High-Level Languages

We abstract the SE interfaces in a C++ library, enabling programmers to easily engage SE's protections for particular program variables. Our library implements 128-bit encrypted data types (*e.g.*, `enc_int`) and uses overloaded operators to emit SE ISA instructions (*e.g.*, `enc_add`) directly via inline *asm()* statements. This implementation encapsulates the SE ISA extension, allowing programs to compute with SE as they compute with primitive types (*e.g.*, `int`), shown in Figure 5. We chose this approach because it enabled fast prototyping. However, one could integrate this support into the compiler using static data-flow analysis, which would be better suited for a high-performance commercial implementation.

SE uniquely allows for tight coupling of secret and non-secret computation. Our library supports sensitive variables for all primitive C++ types, from 1B (*e.g.*, `enc_char`) to 8B (*e.g.*, `enc_double`). These types can be combined with non-secret types in structs or classes. Additionally, since instructions are dispatched to the SE unit on a per-cycle basis, public and private data types and computation can be tightly interleaved. Thus, algorithms can opt to use sensitive types to protect the most critical portions of computation. This reduces overheads for real-world applications by reserving encrypted operations for a minimal subset of values. For example, the Wagner–Fischer algorithm is a well-known dynamic programming algorithm to calculate the edit distance between two substrings. Because the algorithm's looping structure and constants are public, only a small subset of instructions that interact directly with user data need to be encrypted (33%, as found in our *Edit-Distance* benchmark).

B. Programming Limitations

As discussed in Section III-D, the SE unit only supports oblivious instructions. Despite this limitation, SE's conditional move primitive enables programs to still express secret-dependent conditional logic and memory addressing by leveraging the following transformations, which concede performance. We view compilers as promising mechanisms to assist in these code transformations [40], [44], [49], [66].

Secret-Dependent Conditional Logic. Control decisions on secret data can only be expressed using SE conditional move instructions as predicates. As a result, programs cannot use sensitive-data-dependent heuristics, early-exit conditions, or recursion, causing algorithms like *Bubble-Sort* to have worst-case time complexity. The following code snippet demonstrates how programs can compute an if-statement that increments `x` or `y` depending on an encrypted condition `secret`.

```
x = enc_cmov( secret, x+1, x);
y = enc_cmov(!secret, y+1, y);
```

Secret-Dependent Memory Addressing. Secret-based memory addressing or array indexing can be achieved (at a cost) by accessing every element and using a conditional move to select the desired value. Alternatively, SE could be adapted to use techniques like Oblivious RAM (ORAM) [58] to enable secure secret-based addressing, thereby reducing overheads related to these code transformations [27], [65]. The following code snippet demonstrates how programs can compute the secret-dependent memory access `arr[secret]`.

```
for(int i=0;  i<len(arr);  i++;)
    ret = enc_cmov(i==secret, arr[i], ret);
```

VI. Experimental Evaluation

We implemented SE on a x86-64 out-of-order core in gem5 [10] in system call emulation mode with a 2.5 GHz clock frequency. We extend the core with: *i)* an SE functional unit in the execute stage (§IV-A,IV-B), *ii)* extensions to the x86-64 ISA that dispatch the SE ISA instructions to the SE unit (§III-C), and *iii)* an extended 128-bit register file (§III-C). We analyze two distinct SE unit microarchitectures: the stateless variant, denoted **SE**, and the optimized (stateful) variant, denoted **SE-Opt**. Below, we detail the encryption schemes and workloads used for our analysis.

Evaluated Data Encryption Schemes. The data encryption scheme used for SE is parameterizable and can be chosen to match the performance-security needs of the system. In this work, we evaluate three symmetric encryption schemes: *i)* AES-128 (40 cycles), *ii)* Simon-128/128 (20 cycles), and *iii)* QARMA$_{11}$-128-σ_1 (12 cycles). The cipher latencies were derived from hardware implementations reported by prior work [5], [6] and scaled to a 7 nm technology node and 2.5 GHz clock using the techniques in [59]. We use a Mersenne Twister pseudorandom number generator (PRNG) [21] to generate cryptographic salts, re-seeded with a true random source every 200 uses. We used existing open-source encryption libraries

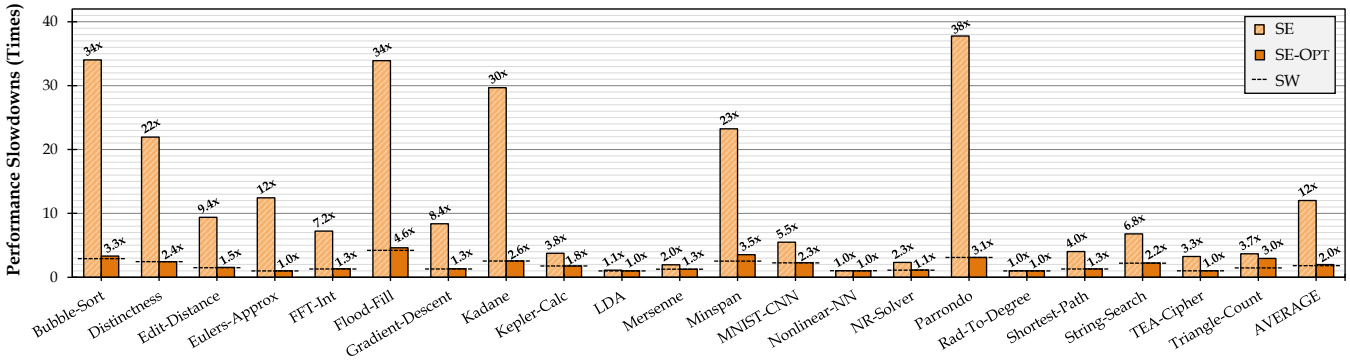


Fig. 6: **Performance Slowdowns for Sequestered Encryption with QARMA$_{11}$-128-σ_1.** This figure shows the overall performance slowdowns for the stateless SE microarchitecture (SE) and the optimized SE microarchitecture (SE-OPT) when parameterized with 12-cycle QARMA data encryption. The dashed lines represents slowdowns originating from software modifications alone (SW).

within our simulation [34], [42], [48] and parameterized gem5 with the above latencies.

Benchmarks. We evaluated SE on 21 benchmarks from the VIP-Bench privacy-focused benchmark suite compiled using GNU g++ with -O2 optimizations [9]. VIP-Bench annotates secret variables in its native workloads, enabling us to easily port all annotated VIP-Bench variables to use the encrypted types defined by SE's C++ library (§V). Furthermore, VIP-Bench includes both native and data-oblivious variants of each workload. This feature enables us to evaluate SE's overheads compared to a native (unprotected, non-data-oblivious) baseline, as well as analyze the software overheads associated with data-oblivious programming transformations.

A. Performance Analysis

We assess the performance overheads of SE and SE-OPT against a native baseline. The results for QARMA are shown in Figure 6. All results are enumerated in Table III in the Appendix. Below, we detail the instruction and performance overheads of our design.

Overheads of Program Modifications. Workloads running on SE may experience an increased dynamic instruction count due to required program transformations. We find that, in many cases, overheads related to data-oblivious modifications are minimal, as computational benchmarks like *Edit-Distance* are oblivious by default. A few benchmarks exhibit more than a 50% increase in dynamic instructions due to data-oblivious code transformations (*i.e.*, *Distinctness*, *Flood-Fill*, and *Kepler-Calc*). Dynamic instruction count also increases due to use of the SE C++ library, which emits additional loads/stores for encrypted data types. Workloads experienced an average 143% increase in loads due to the use of 128-bit encrypted data types for secret values. We believe that these loads, in addition cipher latencies, are the main source of overhead.

The proportion of dynamic instructions dispatched to the SE unit can provide intuition to the relative performance overheads caused by exposed cipher latencies. This analysis is shown in Figure 7. Across all workloads, 30% of committed instructions were dispatched to the SE unit. Some benchmarks only exhibit a small proportion of SE instructions (*i.e.*, *LDA*, *Nonlinear-NN*, *Rad-To-Degree*) because I/O logic overshadows the private computation contained in the benchmark.

Overheads of SE *Microarchitecture*. Performance slowdowns incurred by our stateless microarchitecture design (SE) are significant at 36.6×, 19×, and 12× for AES-128, Simon, and QARMA, respectively. These slowdowns have a near-linear relationship to the latencies of the chosen encryption scheme. Thus, lightweight ciphers have the potential to greatly reduce the overheads of SE. For example, a hypothetical cipher requiring 3 cycles for an encryption operation would have a 4.2× performance slowdown, on average, across our analyzed workloads. Such low-latency ciphers are in development and could benefit SE in the near future [36].

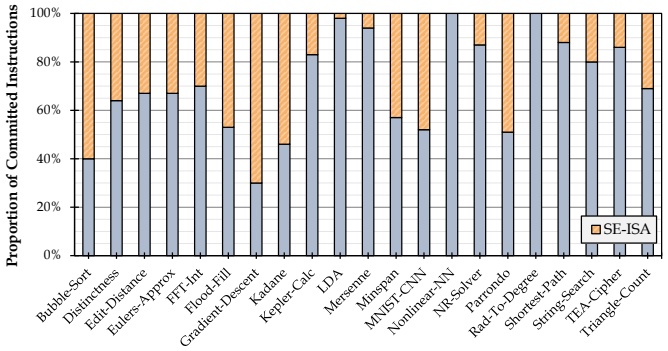


Fig. 7: **Proportion of Dynamic Instruction Count.** This figure shows the proportion of dynamic instructions dispatched to the SE unit (*i.e.*, belonging to the SE ISA extension). On average, 30% of committed instructions were from the SE ISA extension.

Overheads of SE-OPT *Microarchitecture*. Our proposed optimizations significantly reduce the performance overheads caused by exposed SE encryption latencies. Average performance slowdowns incurred by SE-OPT are 2.5×, 2.1×, and 2× for AES-128, Simon, and QARMA, respectively. SE-OPT is still latency-sensitive to some degree, especially for applications that frequently stall for stores. But, in general, the cryptographic latencies are largely hidden, making the use of longer-latency ciphers more tolerable. Our optimizations are particularly effective for benchmarks with extensive register-register arithmetic, like *Eulers-Approx*. In these instances, cipher latencies are completely masked, resulting in little differences in performance across encryption schemes.

Our optimizations are so performant that the slowdowns incurred by SE-OPT begin to approach those caused by software modifications alone (*i.e.*, the modified executable running on a x86 core where SE encryption/decryption is disabled). The dashed lines in Figure 6 represents software overheads. We see that SE-OPT overheads approach that of software for most benchmarks. More specifically, SE-OPT with QARMA only incurs a 17.5% performance degradation, on

average, atop software slowdowns. This demonstrates that the ***cipher latencies in the SE unit are almost completely masked by our optimizations***.

This result informs future directions for reducing the overheads of SE. We hypothesize that the use of 128-bit data types significantly contributes to software overheads (*e.g.*, Booleans are expanded by 16×). Variable data type sizes, reduced salt length, or vectorized types (*i.e.*, SIMD) could reduce these overheads and are promising directions for future work.

Comparison to Homomorphic Encryption. To measure performance benefits compared to HE, we evaluate both frameworks using the workloads presented in Porcupine [25]. These workloads consist of short computation kernels (*e.g.*, dot product) that have limited multiplicative depth, computing only a few operations per data element. For this reason, we do not include these workloads in our main evaluation. We chose Porcupine because these kernels have been optimized specifically for SEAL BFV [57], enabling us to directly run state-of-the-art HE executables with no modifications. We ran the SEAL workloads in a native gem5 instance and compared these results to the SE-OPT microarchitecture parameterized with AES encryption running the same Porcupine computation sequences. Our evaluation, shown in Table II, finds that ***SE provides a speedup over SEAL HE by three orders of magnitude***. Furthermore, we note that the HE workloads evaluated here do not perform bootstrapping. Bootstrapping eliminates noise buildup in computations with significant multiplicative depth and is regarded as very expensive—the most expensive operation in HE. For this reason, bootstrapping is currently not implemented by many HE libraries, including SEAL. If these kernels were to have greater multiplicative depth, bootstrapping requirements would greatly increase HE's overheads, amplifying SE's advantage.

	HE		SE		SE
Kernel	Cycle Count	Slowdown over Native	Cycle Count	Slowdown over Native	Speedup over HE
Gx	537,902,255	24,385×	35,722	1.6×	15,058×
Hamming Distance	588,518,480	3,621×	983,636	6.1×	598×
Polynomial Regression	750,253,318	2,095×	1,019,353	2.8×	736×
Roberts Cross	1,006,490,559	34,954×	34,563	1.2×	29,120×
Dot Product	335,496,620	5,364×	763,955	12.2×	439×
Linear Regression	380,780,634	1,354×	803,238	2.9×	474×
AVERAGE		**11,962**×		**4.5**×	**7,738**×
GEOMEAN		**6,006**×		**3.2**×	**1,851**×

TABLE II: **Comparison to Homomorphic Encryption.** This table presents the processor cycle count and relative performance overheads for the Porcupine kernels [25] when running on SEAL BFV and on SE-OPT parameterized with AES encryption.

B. Security Analysis

In this section, we discuss how the ciphertexts produced by the SE unit are resistant to cryptanalysis and information leakage. The ciphertexts of secret values are exposed outside of the SE unit in software-accessible or leaky hardware structures (*e.g.*, the register file). We assume that the chosen data encryption scheme ensures confidentiality of these values outside of the SE unit. This assumption can readily be met via the use of strong ciphers like AES-128, Simon-128/128, and QARMA$_{11}$-128. Of our assessed encryption schemes, AES-128 is the most tested and secure, as it has been subject to more than twenty years of unsuccessful cryptanalysis.[6] Both Simon-128/128 and QARMA$_{11}$-128 have demonstrated significant resistance to cryptanalysis and provide 128-bits and 256-bits of security, respectively [5], [7].

[6]While implementation attacks have been demonstrated [8], [45], [47], we avoid these pitfalls with a side-channel resistant hardware design.

Even with the use of strong ciphers, our data encryption scheme must use cryptographic salts to prevent attackers from making correlations between ciphertexts throughout computation, as correlating the input and resultant ciphertexts for a conditional move instruction trivially reveals the secret condition. In this section, we evaluate the sufficiency of our salting scheme and discuss how our microarchitecture design gains resilience to timing analysis.

Resilience to Distinguishing Attacks. We demonstrate that our data encryption scheme is resilient to *distinguishing attacks*—a primary form of cryptanalysis where an attacker can distinguish ciphertexts from purely random data due to the presence of patterns, allowing them to recover sensitive values from relations between inputs and outputs. We use the Dieharder [15] random number test suite to analyze the similarity of ciphertexts and detect correlations between sensitive plaintexts and main-core-observed data traces. Dieharder includes statistical tests from the NIST standard for random and pseudorandom number generators [53]. We ran Dieharder on encrypted inputs, outputs, and intermediate values produced by our genomic *Edit-Distance* benchmark, which is of particular interest because it contains only 13 unique data values (*i.e.*, four nucleotides and nine integer/Boolean values). To demonstrate that the dataset contains significant correlations, we ran Dieharder when using only weak XOR encryption. All tests failed, indicating the significant patterns in the benchmark's data traces. When securing *Edit-Distance* using SE's encryption scheme under AES-128, Simon, and QARMA, ***all 114 Dieharder randomness tests passed***, demonstrating that our design presents a significant challenge to cryptanalysis attacks.

Resilience to Timing Analysis. In addition to accessing ciphertexts, attackers can also observe timing properties of the SE unit in an attempt to learn secret data. SE is resilient to timing attacks by design, as all instruction semantics exhibit deterministic timing for both microarchitectural designs. For example, consider the following equation that specifies the timing properties of SE-OPT with the register cache (RC) and decryption cache (DC):

$$\mathrm{L}(\texttt{op}) = \begin{cases} 1 & \text{if } \texttt{reg id} \in RC, \\ 1 & \text{if } \texttt{ctx} \in DC, \\ \mathrm{L}(\texttt{DEC}) & \text{otherwise.} \end{cases} + \begin{cases} 1 & \text{if op is ALU,} \\ 3 & \text{if op is FPU} \end{cases} + \mathrm{L}(\texttt{ENC})$$

where $\mathrm{L}(\cdot)$ denotes the latency of an operation. By construction, these latencies are not plaintext- or key-dependent. While it is trivial to prove that the SE ISA has deterministic timing, a vulnerable or buggy microarchitectural implementation could violate this proof. Such assurances can be provided through verification of a digital circuit design (*e.g.*, an RTL design). Unlike other hardware security mechanisms, the SE TCB is small and independent of surrounding microarchitectural details, lending itself to formal verification. We are pursuing the formal verification of SE in future work.

VII. RELATED WORK

SE leverages trusted hardware to provide data confidentiality in the presence of compromised or untrusted software. SE's security goals are closest to that of homomorphic encryption (HE) [29]. However, SE's design methodology and implementation are similar to trusted execution environments (TEEs) [2], [22], [56]. Thus, we discuss both of these related works in this section. For more direct comparisons between SE and related works, see Sections II-D and VI-A.

A. Homomorphic Encryption (HE)

Homomorphic encryption (HE) [29] is a class of encryption schemes that preserve specific operations performed on ciphertext values after decryption. HE provides strong data confidentiality

while eliminating all trust in software and hardware—a data owner only needs to trust the mathematical foundations of the encryption scheme. However, currently available HE libraries suffer from high performance overheads and programming limitations. As mentioned in Section VI-A, bootstrapping is a costly HE operation needed to eliminate noise that builds over subsequent multiply operations. Bootstrapping is regarded as prohibitively expensive and, as a result, many libraries and applications simply limit multiplicative depth in order to avoid bootstrapping. In addition to this restriction, many libraries cannot readily express conditional move operators or non-linear functions (*e.g.*, SEAL [57]), making them far less expressive than SE. Emerging Boolean HE schemes like TFHE [20] theoretically can support conditional move operations. But, these schemes suffer additional slowdowns and have not seen significant adoption.

Current advancements in HE are focused on overcoming the prohibitive performance overheads of software libraries through the use of optimized algorithms [18], [25] and custom hardware [12], [50], [52], [54]. HEAX [52] is a high-performance architecture for CKKS homomorphic encryption that is able to achieve a 200× performance improvement for ciphertext multiplications. More recent work leverages SIMD processing and massively parallel architectures to gain significant speedups over HE software implementations [12], [54]. Unfortunately, emerging HE research rarely reports absolute performance numbers, motivating our direct performance comparison in Section VI-A. Commensurate comparisons are also challenging because the community does not use standard benchmarks and often reports performance overheads over a non-native baseline (*e.g.*, [52]).

B. Privacy-Preserving ISA Extensions

Data Oblivious ISA Extension (OISA) [65] presents a RISC-V ISA extension to thwart side channels by preserving data-oblivious execution in the processor architecture. Specifically, OISA uses dynamic information flow tracking to detect when sensitive values are used for an unsafe (non-oblivious) instruction. OISA does not encrypt sensitive values, leaving them vulnerable to other disclosure mechanisms. Encrypt-Everything-Everywhere (E^3) [19] presents a software framework and (optional) ISA extension that keeps sensitive data encrypted throughout computation by leveraging either homomorphic operations via its HE interfaces or non-homomorphic operations via an isolated hardware co-processor. While E^3 is conceptually similar to SE, its design is significantly different, as it uses larger ciphertext values (up to 4069 bits), has a much larger hardware footprint, and leverages HE schemes in its implementation.

C. Trusted Execution Environments (TEEs)

Trusted execution environments (TEEs) have significantly different security goals as they are designed to protect the integrity and confidentiality of program execution [17], [27], [39], [51], [60], [61]. While these designs provide some data confidentiality guarantees, they trust that the application software is honorable and often exclude timing side channels from their threat model.[7] For these reasons, we regard SE and TEEs as significantly different security mechanisms that employ similar hardware techniques. ARM TrustZone [2], [41] and Intel SGX [3], [22], [43], [56] isolate applications to enforce the confidentiality and integrity of programs with moderate overheads (<1.2× and 1.7-4.5×, respectively) [56], [64]. However, these secure processor extensions leave the L1-cache and register file unencrypted [22], [56], or fail to provide protection in main memory [2], leaving sensitive data vulnerable to exfiltration [11], [14], [37], [60], [61], [63].

Recent advances in TEEs have sought to eliminate microarchitectural side channels by further isolating sensitive processes. Sanctum [23] and MI6 [13] protect in-order and out-of-order processors, respectively, by flushing microarchitectural state and partitioning the LLC. Both achieve low performance overheads (1.1-1.2× for most workloads) with no programming limitations and trust enclave software within their TCB. The Ascend processor [27], [51] implements fixed-time execution and ORAM to prevent off-chip adversaries from learning the execution traces of untrusted software. This design provides strong security with moderate performance overheads (4-13×) but requires critical computation to be outsourced to the Ascend chip, which is separate from the main processor. These designs have much larger trusted computing bases than SE, as comprehensive isolation and permissions checking requires extensive processor modifications.

[7] Ascend [27], [51] is an exception to this statement as it regularizes function timings.

VIII. Conclusions

Software systems are often trusted to store and process sensitive third-party data despite the pervasiveness of software vulnerabilities. In this work, we explore how a small hardware root of trust can provide confidentiality of third-party data in an untrusted or vulnerable environment. We introduce Sequestered Encryption (SE)—a hardware capability that provides comprehensive data privacy by removing all sensitive plaintext values from software-accessible architectural state. SE encrypts secret data in all external processor architecture and microarchitecture, ensuring that software does not have direct access to sensitive plaintext values. With optimizations, SE achieves <2.5× performance slowdowns over native execution and exhibits a three orders-of-magnitude advantage over software HE libraries, demonstrating that architectural approaches can emerge as dynamic, expressive, and performant data privacy solutions that possess zero trust in software.

Acknowledgments

This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA. The authors would particularly like to thank the ADA Center sponsors and task liaisons for their ongoing input on this project.

Appendix

Table III gives a comprehensive listing of our performance results for the SE and SE-Opt microarchitectures when parameterized by three different data encryption schemes: AES-128 (40 cycles) [5], Simon-128/128 (20 cycles) [6], and QARMA_{11}-128-σ_1 (12 cycles) [5]. In the second column, we list the performance overheads resulting from software modifications alone (*i.e.*, data-oblivious program modifications and usage of the SE C++ data type library).

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):1–35, 2018.

[2] Tiago Alves. Trustzone: Integrated hardware and software security. *White paper*, 2004.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[4] Apple Inc. Storing keys in the secure enclave. https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave.

Benchmark	Software	SE			SE-OPT		
		AES-128	Simon	QARMA	AES-128	Simon	QARMA
Bubble-Sort	188.9%	10,586.4%	5,382.7%	3,303.1%	390.7%	268.9%	231.4%
Distinctness	142.2%	6,657.0%	3,398.4%	2,094.4%	143.6%	143.4%	143.4%
Edit-Distance	46.5%	2,735.2%	1,377.1%	839.2%	74.3%	59.9%	53.7%
Eulers-Approx	0.0%	3,929.0%	1,939.6%	1,143.9%	0.0%	0.0%	0.0%
FFT-Int	28.7%	2,044.6%	1,024.7%	622.1%	48.8%	34.2%	32.2%
Flood-Fill	324.5%	10,347.0%	5,302.4%	3,291.1%	583.5%	423.3%	360.7%
Gradient-Descent	24.8%	2,609.5%	1,271.8%	736.7%	32.9%	32.9%	32.9%
Kadane	155.5%	9,230.6%	4,685.4%	2,868.2%	155.5%	155.5%	155.5%
Kepler-Calc	73.8%	794.1%	422.8%	275.2%	82.4%	75.8%	76.7%
LDA	0.2%	36.9%	17.0%	9.5%	0.7%	0.4%	0.0%
Mersenne	23.8%	300.2%	152.0%	94.9%	31.2%	28.7%	25.1%
Minspan	152.6%	7,084.7%	3,611.5%	2,224.4%	572.5%	352.3%	253.4%
MNIST-CNN	126.4%	1,623.1%	781.5%	448.2%	159.2%	132.3%	129.2%
Nonlinear-NN	0.8%	8.8%	3.3%	1.5%	0.8%	0.8%	0.8%
NR-Solver	11.0%	428.9%	216.9%	132.4%	29.9%	19.4%	13.7%
Parrondo	204.4%	11,923.1%	6,033.7%	3,677.9%	208.7%	208.7%	208.7%
Rad-To-Degree	0.3%	1.9%	0.7%	0.3%	0.3%	0.3%	0.3%
Shortest-Path	27.6%	967.0%	493.0%	302.0%	30.9%	30.8%	30.0%
String-Search	121.1%	1,668.9%	889.5%	577.5%	128.0%	124.3%	123.9%
TEA-Cipher	2.1%	756.2%	376.5%	225.0%	3.7%	2.3%	2.5%
Triangle-Count	46.6%	1,037.7%	485.8%	265.5%	540.6%	293.8%	195.4%
AVERAGE	81.0%	3,560.5%	1,803.2%	1,101.6%	153.2%	113.7%	98.5%
GEOMEAN	17.5%	1,071.4%	523.6%	307.6%	28.2%	23.3%	19.0%

TABLE III: **Performance Overheads.** In this table, we show the slowdowns for our SE and SE-OPT microarchitecture designs when parameterized with AES-128 (40 cycles) [5], Simon-128/128 (20 cycles) [6], and QARMA_{11}-128-σ_1 (12 cycles) [5] data encryption. The second column reports slowdowns resulting from only the software modifications required to support SE (*i.e.*, SE encryption/decryption is disabled in the SE unit).

[5] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.

[6] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Implementation and performance of the Simon and Speck lightweight block ciphers on ASICs. *Unpublished work*.

[7] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, 2013.

[8] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[9] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Galane Basha Namomsa, Plato Gebremedhin, Fitsum Assamnew Andargie, Brandon Reagen, and Todd Austin. VIP-Bench: A Benchmark Suite for Evaluating Privacy-Enhanced Computation Frameworks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 139–149, 2021.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[11] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1213–1227, 2018.

[12] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

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

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[15] Brown, Robert G. The DieHarder Test Suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 769–784, 2019.

[17] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[18] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[19] Eduardo Chielle, Nektarios Georgios Tsoutsos, Oleg Mazonka, and Michail Maniatakos. Encrypt-Everything-Everywhere: ISA Extensions for Private Computation. *IEEE Transactions on Dependable and Secure Computing*, 2020.

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

[21] Christian Stigen Larsen. A fast Mersenne Twister in C++ (GitHub). https://github.com/m0n0ph1/mersenne-twister-1, 2012.

[22] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

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

[24] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium (USENIX Security 03)*, volume 12, pages 91–104, 2003.

[25] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 375–389, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR, abs/1710.00458*, 6:93–94, 2017.

[27] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM workshop on Scalable Trusted Computing*, pages 3–8, 2012.

[28] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 469–484, 2019.

[29] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.

[30] Kevin C Gotze, Gregory M Iovino, and Jiangtao Li. Secure provisioning of secret keys during integrated circuit manufacturing, August 22 2017. US Patent 9,742,563.

[31] Kevin C Gotze, Jiangtao Li, and Gregory M Iovino. Fuse attestation to secure the provisioning of secret keys during integrated circuit manufacturing, November 11 2014. US Patent 8,885,819.

[32] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *International Conference on Information Security and Cryptology*, pages 176–192, 2009.

[33] IBM Security. Cost of a data breach 2020. https://www.ibm.com/security/data-breach, 2020.

[34] Intel Corporation. TinyCrypt Cryptographic Library (GitHub). https://github.com/intel/tinycrypt, August 2017.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[36] Michael Kounavis, Sergej Deutsch, Santosh Ghosh, and David Durham. K-Cipher: A Low Latency, Bit Length Parameterizable Cipher. In *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020.

[37] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, 2017.

[38] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, et al. Cryptographic Capability Computing. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 253–267, 2021.

[39] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM Sigplan Notices*, 35(11):168–177, 2000.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy (S&P)*, pages 359–376. IEEE, 2015.

[41] Naiwei Liu, Meng Yu, Wanyu Zang, and Ravi Sandhu. On the Cost-Effectiveness of TrustZone Defense on ARM Platform. In *Information Security Applications*, pages 203–214. Springer International Publishing, 2020.

[42] McCoy, Calvin. Simon_Speck_Ciphers (Github). https://github.com/inmcm/Simon_Speck_Ciphers, July 2018.

[43] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[44] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[46] Zhiniang Peng. Danger of using fully homomorphic encryption: A look at Microsoft SEAL. *arXiv preprint arXiv:1906.07127*, 2019.

[47] Colin Percival. Cache missing for fun and profit, 2005.

[48] Phantom1003. QARMA64 (Github). https://github.com/Phantom1003/QARMA64, October 2019.

[49] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., August 2015. USENIX Association.

[50] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.

[51] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. Design and implementation of the Ascend secure processor. *IEEE Transactions on Dependable and Secure Computing*, 16(2):204–216, 2017.

[52] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1295–1309, 2020.

[53] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz Allen & Hamilton, 2001.

[54] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021.

[55] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR Cryptol. ePrint Arch.*, page 549, 2017.

[56] Matthias Schunter. Intel Software Guard Extensions: Introduction and Open Research Challenges. In *Proceedings of the 2016 ACM Workshop on Software Protection (SPRO '16)*. Association for Computing Machinery, 2016.

[57] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, November 2020. Microsoft Research, Redmond, WA.

[58] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 299–310, 2013.

[59] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, 2017.

[60] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.

[61] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *32nd International Symposium on Computer Architecture (ISCA)*, pages 25–36. IEEE, 2005.

[62] Nancy Sumrall and Manny Novoa. Trusted computing group (TCG) and the TPM 1.2 specification. In *Intel Developer Forum*, volume 32, 2003.

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

[64] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2):81–93, 2017.

[65] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data Oblivious ISA extensions for Side Channel-Resistant and High Performance Computing. *Cryptology ePrint Archive*, 2018.

[66] Samee Zahur and David Evans. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.*, 2015:1153, 2015.