SoK: Opportunities for Software-Hardware-Security Codesign for Next Generation Secure Computing

Deeksha Dangwal
Facebook Reality Labs Research
Redmond, WA, USA
ddangwal@fb.com

Meghan Cowan
Facebook Reality Labs Research
Redmond, WA, USA
meghancowan@fb.com

Armin Alaghi
Facebook Reality Labs Research
Redmond, WA, USA
alaghi@fb.com

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA
vtlee@fb.com

Brandon Reagen
New York University
New York, NY, USA
bjr5@nyu.edu

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

ABSTRACT

Users are demanding increased data security. As a result, security is rapidly becoming a first-order design constraint in next generation computing systems. Researchers and practitioners are exploring various security technologies to meet user demand such as trusted execution environments (e.g., Intel SGX, ARM TrustZone), homomorphic encryption, and differential privacy. Each technique provides some degree of security, but differs with respect to threat coverage, performance overheads, as well as implementation and deployment challenges. In this paper, we present a systemization of knowledge (SoK) on these design considerations and trade-offs using several prominent security technologies. Our study exposes the need for *software-hardware-security* codesign to realize efficient and effective solutions of securing user data. In particular, we explore how design considerations across applications, hardware, and security mechanisms must be combined to overcome fundamental limitations in current technologies so that we can minimize performance overhead while achieving sufficient threat model coverage. Finally, we propose a set of guidelines to facilitate putting these secure computing technologies into practice.

ACM Reference Format:
Deeksha Dangwal, Meghan Cowan, Armin Alaghi, Vincent T. Lee, Brandon Reagen, and Caroline Trippel. 2020. SoK: Opportunities for Software-Hardware-Security Codesign for Next Generation Secure Computing. In *Hardware and Architectural Support for Security and Privacy (HASP '20), October 17, 2020, Virtual, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458903.3458911

1 INTRODUCTION

Over the past decade, data security has emerged as a first-order design constraint. Users have begun to demand increased accountability from data aggregators; they want to know how their data is being managed and protected against misuse or abuse. As a result, the development of and interest in secure computing technologies has progressed rapidly over the last few years. For example, technologies such as trusted execution environments (TEEs), homomorphic encryption (HE), and differential privacy (DP) are all rapidly maturing areas of research.

Despite recent advances, each technology presents a range of trade-offs and challenges; these challenges include: (1) implementing the secure computing technology, (2) verifying that the security guards are correctly implemented and provide intended threat model support, (3) porting the technology to support a variety of applications, and (4) maintaining practical performance targets.

Secure computing solutions present trade-offs that impact security and performance which must be carefully balanced to satisfy performance and security specification targets. We argue that, if carefully engineered, secure computing technologies can enable the best of both worlds (good performance and security) instead of mandating one or the other (performance or security). For example, TEEs offer confidentiality and integrity for computations executing within an isolated memory location, i.e., an enclave. TEE support for an application like machine learning (ML) inference on sensitive data would be critical. To deploy the application, the designer needs to port the ML inference to work with the chosen TEE's API and consider the cost of each call into the TEE. For example, SGX enclaves have limited memory capacity. Therefore large ML models need to be encrypted and stored on a larger, enclave-external memory and the resulting computation partitioned properly. Model partitions must be moved into the SGX enclave, which causes page swapping overheads. Combined, these design considerations require expertise across the software/application, hardware, and security in tandem to achieve an efficient and secure implementation. Overcoming these challenges mandates a careful re-examination of complex cross-stack design choices to find effective and efficient solutions.

More broadly, software-hardware-security codesign has emerged as a design philosophy for realizing performance- and power-efficient secure computing technologies. This notion has historical precedence; software-hardware codesign has a proven track record in improving the overall efficiency of computing solutions that we build to support modern applications. Examples include fixed-point approximation for neural networks, domain-specific

HASP '20, October 17, 2020, Virtual, Greece

ACM ISBN 978-1-4503-8898-6/20/10.
https://doi.org/10.1145/3458903.3458911

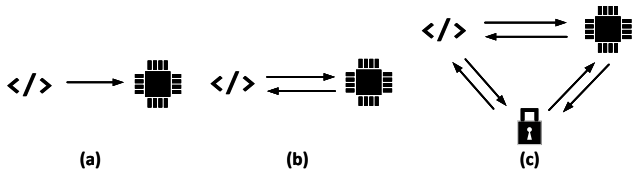

Figure 1: Hardware acceleration versus software-hardware codesign and software-hardware-security codesign. (a) Hardware acceleration does not integrate hardware design feedback. (b) Software/hardware codesign integrates feedback. (c) Software-hardware-security codesign additionally integrates security considerations.

accelerators like digital signal processor (DSP) units, and hardware-friendly mathematical approximations. These examples of software-hardware codesign have had tremendous practical impact; aggressive fixed-point quantization has enabled power-efficient implementations of machine learning and DSPs have enabled high performance telecommunication applications that form the backbone of mobile networks. However, these innovations required feedback between software and hardware design considerations to arrive at the highly optimized solutions and architectures we have today.

A valuable codesign process by adding security to the mix can analogously enhance the efficacy of secure computing technologies. To address the added complexities introduced by security considerations, experts from across hardware, software, and security need to be able to communicate through systematic and well-defined abstraction boundaries. Unfortunately, the process of identifying these opportunities is not well systemized; many seemingly disparate computing techniques fall into the category of codesign. It is, therefore, valuable to concretely systemize when there are codesign opportunities to allow this design iteration to occur for emerging secure computing technologies such as TEEs, HE, and DP.

We propose a systemization of knowledge to analyze the hardware, software, and security design considerations required to realize practical and efficient solutions for emerging secure computing technologies. We start by defining the notion of software-hardware-security codesign (Section 2). We then show there is historical precedence in the selection process of the advanced encryption standard and apply our systemization to emerging secure computing technologies (Section 3): differential privacy, trusted execution environments, and homomorphic encryption. We find that a common theme across these secure computing technologies is the prevalence of *feedback loops* to inform and drive the implementations towards better designs. Finally, we synthesize these insights (Section 4) to motivate the need to put software-hardware-security codesign into practice and enable security as a first-order design constraint.

2 CODESIGN METHODOLOGIES

This section defines the notion of software-hardware-security codesign and shows it is a natural extension to the existing software-hardware codesign process.

2.1 Software-Hardware Codesign

Software-hardware codesign is a popular concept that originated in the 1990s as a way to co-optimize embedded systems software with the underlying hardware implementation [36]. Software-hardware codesign *simultaneously considers both hardware and software constraints to make better design decisions and improve the overall quality of results (i.e., performance, power, accuracy, etc.) of the system* (Figure 1b). The key distinction between standard *hardware acceleration* (Figure 1a) and *software-hardware codesign* is an iterative feedback loop between software and hardware considerations. Hardware acceleration is a one-way process where the application is not affected by hardware constraints; software-hardware codesign allows design consideration feedback in either direction between software and hardware. This distinction is important because this feedback loop is what enables co-optimization of the software and hardware to ultimately yield a more efficient overall solution.

A canonical example of software-hardware codesign when building a hardware accelerator is fixed-point approximation (i.e., fixed-point optimization). In fixed-point approximation, a designer may convert over-provisioned floating-point units to fixed-point arithmetic since hardware implementations of floating points require more power, area, and latency. However, replacing floating-point with fixed-point units introduces some error to the original functionality. For this substitution to be acceptable, the designer must revisit the application to ensure that the approximation does not adversely impact accuracy metrics. In this case, the application drives the initial hardware accelerator design but power-efficiency considerations feeds information back to the application design. The impact on accuracy and performance further informs whether the quantization approximation is acceptable; if not, the hardware is adjusted to provide better fidelity and the process iterates.

2.2 Software-Hardware-Security Codesign

It is natural to extend this idea of having a tightly coupled, synergistic codesign feedback loop to include security constraints. We define the notion of software-hardware-security codesign as *a design process which simultaneously considers software, hardware, and security design parameters when optimizing for power- and performance-efficient, high-fidelity, and threat-model-optimal execution solutions.* (Figure 1c). More precisely, we define *software* to encompass both the software implementation (i.e., source code) as well as application design choices. *Hardware* includes architectural, microarchitectural, and silicon implementation decisions such as compute units, memory sizes, and communication overheads. *Security* refers to parameters associated with, and constraints imposed by the secure computing technology; for instance, these can manifest as encryption scheme parameters and instruction set restrictions. We also include threat model considerations as part of the security specifications that each technology tries to mitigate against.

The concept of software-hardware-security codesign has been proposed in the past under different variations such as software-hardware codesign for instances of secure computing technologies. For example, software-hardware codesign to enhance security has been proposed for resource restricted embedded systems such as [55] for remote attestation and for implementing minimal roots of trust [21]. Power, energy, performance, and area constrained embedded systems must now also consider and satisfy security targets. Our work seeks to highlight this repeating theme and shed light

on the fact that these codesign opportunities should be seriously considered for maturing secure computing technologies.

We organize design consideration insights into a dependency graph to understand the codesign interactions between hardware, software, and security (Figure 1c). This serves as our basic blueprint to systemize the codesign considerations between hardware, software, and security. Each node in the dependency graph represents a collection of design parameters associated with hardware, software, and security. An edge from *A* to *B*, indicates that changing a design parameter in *A* impacts a design consideration in *B*.

Our goal is to establish a set of design consideration dependencies for each secure computing technology, and expose the feedback loops between the nodes and the opportunities for enabling iterative codesign. A complete systemization for a secure computing technology will require many edges between nodes to capture every possible interaction. Similar to how software-hardware codesign delicately balances power, performance, accuracy, area, and energy costs to meet respective specification targets, security considerations must also be carefully balanced with other implementation costs. We are *not* proposing sacrificing security for the sake of prioritizing other quality metrics. Rather we are noting that like power, performance, energy, and area that there is a similar balancing act that can trade-off the strength of the secure computing technology *while still achieving a sufficient security guarantee* that defends against the target threat model. For instance, protecting against *all* possible threats is typically not feasible; security targets needs to be adjusted to a sufficient level of security to address practical attack scenarios.

Next, we look at instances of software-hardware-security codesign and explore historical precedence before systemizing modern technologies. By exposing the role of software-hardware-codesign, we expect that the insight will provide guidance into the research directions required to drive future codesign opportunities.

3 SYSTEMIZATION OF SECURE COMPUTING TECHNOLOGIES

This section applies our systemization to three secure computing technologies and exposes the software-hardware-security codesign opportunities. But first, we start with historical precedence to understand how the codesign process has manifested in the past and then look at emerging secure computing technologies.

3.1 Historical Precedence: Design of the Advanced Encryption Standard

The concept of software-hardware-security codesign is not without precedence and has been used in the past to build some of the hardware systems we use today. A quintessential historical case study is the development and selection process of the Advanced Encryption Standard (AES) which is widely used and available in modern computing stacks. The development of the AES dates back to 1997 and was motivated by the need for stronger end-to-end encryption protocols [1]. The algorithm for backing what we know as AES today originally consisted of several finalist proposals: MARS [9], RC6 [61], Rijndael [14], Serpent [5], and Twofish [66]. Ultimately, the NIST selected the Rijndael algorithm but the process through which they arrived at this conclusion provides valuable insights into the importance of the codesign process.

The algorithm that backs the AES was selected after three rounds of public comment based on software-hardware-security codesign considerations. The selection process report [54] outlines the security, hardware, and software design considerations that were used to ultimately select the final standard. The selection criteria was split into "1) Security, 2) Cost, and 3) Algorithm and Implementation Characteristics"; the latter two considerations manifested in the public comment periods where studies of the software and hardware implementation characteristics for the proposed algorithms were explicitly solicited.

The AES selection process as a software-hardware-security codesign exercise has many aspects summarized in Table 1. The algorithm selection process was driven by both security considerations as well as software and hardware implementation costs. Cryptanalysis of the security strength of the proposed algorithms contributed to but did not solely drive the selection of the final algorithm. Over three years, hardware and software performance analyses were conducted to complement the security analyses; multiple case studies compared the resource efficiency as well as performance of these applications on CPU [39], FPGA [15], and ASIC platforms [32]. The selection process also conducted a thorough analysis of aspects such as power and timing side-channel attacks when considering the security strength and viability of the proposed algorithms. Finally, the process included architectural considerations such as impact of machine word size on security parameters like key size, as well as the impact of restricted computing environments (e.g., systems with limited memory) on the candidate encryption algorithms.

The selection process also contemplated a number of other considerations that go beyond software, hardware, and security. These included simplicity of the application solution, flexibility, and intellectual property restrictions. The selection process also examined a range of future-proofing measures such as larger machine word sizes (64-bit machines were not around then), whether to standardize two algorithms in case one was compromised, and ancestry of the algorithms to minimize risk of hidden backdoors.

The impact of the AES standardization and codesign efforts is clearly visible today where it is ubiquitous for securing communication lines and data assets. Modern implementations of AES now have dedicated hardware acceleration units and instruction set customizations to make them highly optimized. As a result, AES is a classical case study of how software-hardware-security codesign ultimately led to an efficient and secure computing solution.

3.2 Differential Privacy

Differential privacy (DP) is useful when one wants to answer questions about or analyze data in aggregate while protecting individual pieces of data. DP [19] is originally a data information retrieval concept that was proposed to protect the privacy of individual entries in an aggregate database. In a differentially private system, information about individuals is aggregated in a way that does not reveal information any individuals who contributed data to the database while still allowing some public information about the group as a whole. More generally, we say DP is upheld when the

Table 1: Systemization of software-hardware-security codesign considerations for advanced encryption standard (AES), differential privacy (DP), trusted execution environments (TEEs), and homomorphic encryption (HE). Feedback between software, hardware, and security design considerations highlight the need for codesign for each secure computing technology.

	Software ⇆ Hardware	Software ⇆ Security	Hardware ⇆ Security
AES	← Resource and performance results drives application selection → Application selection determines performance and resource needs	← Security needs and specification drive application design ← Cryptanalysis determines application security strength	← Encryption key size impacts hardware efficiency → Hardware implementation determines timing and power side-channels
DP	← Hardware support can facilitate or enable DP in software → Software drives correlated behavior in hardware and lower levels of the stack	← DP requires integration of noise and random decisions into application → Application selection and behavior defines DP privacy budget	→ Correlated hardware behavior may not be DP and expose side channels
TEE	← Exposes hardware security primitives through ISA extensions → Program optimized to leverage hardware ISA extensions	← Requires application modifications to leverage security primitives → Expose application-level threat model requirements	← Offload root of trust to hardware support → Hardware-based remote attestation to enable root of trust → Influences security interface to leverage hardware
HE	← Hardware architecture and speedups depend on application parallelism → High performance overheads mandates hardware support and acceleration	← HE constrains set of efficient instructions and levels of logic ← Requires application quantization to encodable HE values → Application behavior determines noise budget and encryption parameter settings	← Encryption parameter settings determine datapath width and storage size ← HE scheme determines useful computational primitives for acceleration

observer of the computation output on the aggregated database does not reveal any individual entry.

DP is typically implemented by adding noise to data assets. For example, canonical implementations use Laplace mechanisms [20, 65], median mechanisms [62], exponential mechanisms [50], or randomized response mechanisms [22]. The addition of noise allows individual data entries to be protected from an adversary. DP has two variants: local and global. In global DP, a trusted central database aggregates individual data assets; in the local variant, noise is first added to individual data assets before communicating them to a potentially untrusted central database aggregator. Local differential privacy (LDP) is more desirable in modern systems because it protects client data before it enters any potentially untrusted system.

For example, LDP is useful when collecting private sensor information from embedded Internet of Things (IoT) systems. These embedded devices often run on limited compute and power resources. The guarantees of differentially private algorithms change under such limited resources. Prior work [12] for LDP has shown that low-resolution and fixed-point implementations that are prevalent in IoT devices are counterproductive to and hinder private computation. The authors present new resampling and thresholding techniques that they implement in hardware to continue to provide LDP privacy budget guarantees. More importantly, this case study illustrates the need to consider hardware restrictions on the effective implementation of an LDP application.

DP also often requires modifying the application to integrate noise mechanisms. However, DP considerations are not restricted to just the application behavior and applies more broadly to the remainder of the computing stack. For instance, recent work has proposed differentially private type systems [24, 53, 59] and programming language support [7, 51, 63] which integrates differential privacy mechanisms into the compilation process. There is also emerging work that exposes the need for DP for memory access patterns [10, 40] when outsourcing private applications to an untrusted cloud platform to protect from side-channel attacks, even when running the application within a TEE.

DP is another opportunity for software-hardware-security codesign summarized in Table 1. DP requires adjustment of the application to integrate the noise and random decisions to make the application private. The application behavior in turn determines the privacy budget and configuration settings for DP to guarantee application privacy. While an application implementation may be differentially private, the observable behavior across the remainder of the computing stack down to hardware may not be differentially private. Thus, it is important to also consider hardware systems (ex., memory access) when determining whether the DP security guarantees are enforced as the presence of side channel or other vulnerabilities can defeat the efficacy of software-only DP. Finally, recent work such as [12, 48] also shows that there is room for integrating or enhancing existing hardware support to better facilitate DP software such with thresholding and resampling techniques.

3.3 Trusted Execution Environments

Trusted execution environments (TEEs) serve as a proxy for encrypted computation. Intuitively, TEEs provide an execution environment on a host processor to execute private computation[1] that is protected from observation by others parties on the host processor. For instance, a client system may request a TEE to set up and offload a private computation to the host processor via remote attestation. TEEs operate by allocating an isolated hardware-protected memory region called an enclave where a protected application's code and data can reside. Data within enclave memory can only be accessed by code that also resides within enclave memory and special instructions are provided for invoking enclave code.

Computation within an enclave is protected from inspection by other users and system software (with the exception of side-channel attacks). Users can leverage the TEE's secure remote attestation protocol to ensure that their data is indeed being executed within the intended TEE and that the encrypted data that is sent to the server. Memory management features also ensure that the host system cannot tamper with or observe the computation on the supplied data. With TEEs, rather than placing trust solely in a cryptographic algorithm (as in multi-party computation or homomorphic encryption), clients place trust in the manufacturer's (Intel [49], ARM [4], AMD [37], etc.) hardware to ensure that their private computations are only ever observed by the TEE compute units (i.e., not the unprotected host processors).

In TEEs, hardware considerations consist of instruction set extensions - a standard architecture software-hardware codesign technique - for facilitating secure remote attestation and secure entry to and exit from enclave code. This ISA interface (e.g., via SGX ISA extensions) enables software to articulate security requirements at the function granularity. These ISA extensions enable users to run applications on remote machines with confidentiality and in some cases integrity guarantees with respect to a third party adversary on the host machine. In particular, these extensions (1) enable a user to send code to execute on a remote machine as an enclave application, (2) verify that their desired code is what is running on the remote TEE, and (3) establish a secure communication channel with the remote TEE to send secret data. The application in exchange for programming against the ISA interface receives confidentiality and possibly integrity guarantees (for data) with the exception of side channels and integrity guarantees (for code and data) via hardware support for properly partitioned programs. [2] In other words, software is responsible for providing a partitioning that properly leverages the TEE interface.

TEEs can impose restrictions and modifications to the software or application to fully leverage the advantages while minimizing performance overheads. The application developer (or in compiler infrastructure) translates an application and its associated security requirements to leverage the software-exposed ISA security interface. This requires refactoring sensitive application function calls which must be protected to interface with the enclave by annotating valid enclave entry and exit points. This partitioning requires the application designer to be aware of and minimize the overheads associated with moving data into and out of the enclave. Correctly partitioned software allows for preserving integrity (e.g., in the case of SGX) for code and data; it also enables data confidentiality guarantees (sometimes with the exception of side channels). Unfortunately, such a partitioning of the code is currently left up to the user and is not guaranteed to produce a secure result [6, 42, 67].

Given an application, fully offloading the entire computation to a TEE in Intel's SGX implementation can be expensive in terms of performance. As a result, there have been some proposals to partially secure an application computing pipeline to only incur the overheads for a smaller fraction of the computation and execute the remaining computation in plaintext [52, 64]. The high level intuition is that partially encrypted computation will yield intermediary results (unencrypted) which are less sensitive or meaningful. However, there are no systematic methodologies for determining how to partially partition an application while still satisfying cryptographically strong guarantees which makes this an iterative process to balance performance and security levels. Researchers are also exploring ways to produce such a portion automatically [42].

Combined, these design considerations illustrate how TEEs are another example of how software, hardware, and security must be co-optimized together as summarized in Table 1. Hardware exposes a security interface in the form of ISA extensions for supporting secure remote enclave execution. Software is refactored to express its security requirements and map this requirements onto the software-exposed ISA extensions. In the case of SGX, as a specific example, code is partitioned into enclave code and native code using SGX ISA extensions. The result is confidentiality (with the exception of side-channels) for enclave data and integrity for enclave code and data. In other words, hardware and software are working together to provide a particular user-specified level of application security.

3.4 Homomorphic Encryption

Homomorphic encryption (HE) is an emerging secure computing technology that enables computation *directly over ciphertexts without decrypting the contents* and was proposed by Gentry [25, 26]. In non-HE encryption, a client encrypts the plaintext data and sends it to a cloud service provider; the cloud provider then decrypts the data back into plaintext where functions can readily be applied. Once computation completes, the service provider encrypts the result and sends it back to the client. In this setting an honest-but-curious adversary (cloud service provider), such as those assumed by [28, 35], can see the plaintext copy of sensitive user data. HE is different because it allows for a cloud service provider to directly perform computation on ciphertexts; as a result, in HE a cloud provider never sees the plaintext version of the data it is performing computation on. This provides cryptographically strong guarantees against adversaries for the data transmitted to the datacenter for computation as only the client is able to decrypt the data. HE provides security guarantees while still allowing an untrusted third party to perform useful computation on the encrypted data.

HE comes with a number of severe computational restrictions and overheads which mandate the need for software-hardware-security codesign considerations to be practical. First, the performance overheads for executing homomorphically encrypted kernels are generally four to five orders of magnitude [27]. In other

[1] Some TEEs also guarantee integrity of the computation

[2] Some TEEs (e.g., MIT Sanctum [13]) also consider side channels threats while other TEEs (e.g., Intel SGX) do not.

words, executing a single multiplication or addition in HE over two encoded ciphertexts is 10000× to 100000× slower than a single plaintext addition using a CPU ALU. As a result, to bring HE computation back down to practical speeds, HE will require significant codesign, hardware acceleration, and optimizations to reduce this overhead as much as possible. Recent work has shown that certain applications containing abundant application parallelism which can be aggressively exploited by the underlying specialized hardware accelerator are preferable [58, 60].

Second, encrypted ciphertexts in HE are each associated with a noise budget that degrades monotonically with successive computations. If the noise budget is exceeded for a given ciphertext the decryption procedure will fail which yields a random result and effectively losing the encoded value. The noise budget is governed by parameters associated with the HE encryption scheme which can be increased to increase the noise budget. These encryption parameters also determine the size of the ciphertext representation. Increasing the encryption parameter sizes to increase noise budget directly increases the compute datapath width and storage requirements for ciphertexts. As a result, these encryption parameters must be carefully balanced to avoid wasting noise margins but still allow correct decryption. Methods like bootstrapping can reduce the noise for an intermediary ciphertext but the process is impractically slow (an additional 4-6 orders of magnitude overhead) [68]. Therefore, practical implementations of HE computations effectively have a limited logical depth for computation (i.e., *leveled* HE) that they can support bound by the noise budget.

Third, HE is severely limited in the types of computations that it can support efficiently. Most modern constructions of HE such as BFV [23], BGV [8], and CKKS [11] pack and encrypt multiple plaintext values into a single ciphertext representation. Each value is packed into a slot in the ciphertext the same way values are packed into vector registers. Furthermore, values are constrained to fixed point or integer which require quantizing the target application. Modern HE schemes only support single instruction multiple data (SIMD) add, SIMD multiply, and slot rotation, which allows swapping the values between slots in the vectors of the ciphertext representations, over the ciphertext vectors, Each of these three operations as well as the order in which they are executed increase the noise budget differently. In theory, arbitrary computation can be constructed out of addition and multiplication operations but they are not efficient in HE as they require many logical layers of computation to implement. This means that application selection is a key design consideration; vectorizable computation with fewer logical layers with minimal control logic is more efficient in HE. For instance, statically schedulable computation such as matrix multiplication is more amenable to HE than computation that requires data dependent control flow like sparse matrix algebra.

Putting it all together, HE presents a number of software-hardware-security codesign opportunities summarized in Table 1. The high overhead of HE mandates the need for aggressive hardware specialization but the maximum speed ups a hardware architecture can achieve is determined by available application parallelism. Similarly, application selection determines the underlying computational behavior such as instruction mix, control flow, and amenability to vectorization which impacts the noise budget requirements to prevent decryption from failing. This ultimately dictates the noise budget requirements and encryption parameter settings which affect the compute datapath width and storage requirements on the target hardware platform. All together, we find that reaching optimized design solutions with HE is another example of an opportunity for software-hardware-security codesign.

3.5 Other Secure Computing Technologies

There are other instances (in varying stages of maturity) of secure computing technologies that benefit from hardware-software-security codesign insights; for instance, one group of secure computing technologies specifically addresses information leakage through side channels. Consider systems where multiple tenants share some key computing resources such as caches. This is the case in cloud computing applications, which store and compute over private data. In cloud service infrastructures, virtual machines share the underlying hardware resources and isolation is usually not provided by default. Stealthy cache attacks such as FLUSH+RELOAD [29, 31, 70], PRIME+PROBE [33, 38, 47, 56, 57], EVICT+TIME [56] and others [18] are considered practical; this allows adversaries to steal secret AES [38], RSA [70], and ElGamal [47] cryptographic keys, spy over encrypted channels [34], and log keys [30]. A similar scenario also occurs in smart phones, where a malicious application can learn side-channel information about the system through shared resources such as caches [43].

There are ways to mitigate against these side-channel attacks but identifying the root causes and mitigating them requires insights into both hardware and software behaviors to codesign them to meet security requirements. For example, software running on vulnerable systems should be written in such a way that it avoids behavior that enables information leakage. This can be done by writing code such that control flow or memory access patterns eliminate observable side effects that leak information through hardware-induced side channels. Software countermeasures such as page coloring can also be used. Other hardware solutions such as Intel's cache allocation technology, and attack resistant caches [45, 46, 69] can mitigate these vulnerabilities but still require codesigning the underlying hardware systems to be aware of and properly mitigate such vulnerabilities.

The post-quantum cryptography competition [2, 3] illustrates another relevant opportunity for software-hardware-security codesign similar to the standardization of AES. The primary motivation for the post-quantum cryptography effort stems from application concerns that quantum computing will defeat the hardness guarantees of existing cryptographic techniques, i.e., there is a need to re-codesign the software or encryption algorithm to strengthen security properties. This illustrates the ongoing design feedback between security needs and the algorithms that realize them. The latest round of the standardization effort focus on the security and cryptanalysis but, as with prior efforts such as AES, there is an opportunity for hardware and software codesign considerations to ensure that theoretically strong cryptography can be put into practice.

4 PUTTING SYSTEMIZATION INTO PRACTICE

Our systemization shows that to support the next generation of secure computing we must integrate knowledge across software,

hardware, and security. To do this, we need systematic rules of thumb to guide effective integration into the codesign process. This section highlights several opportunities and focus areas to facilitate putting software-hardware-security codesign into practice.

4.1 Security as a First-Order Design Principle

Security is often treated as an afterthought in the software and hardware design process. The Spectre [41] and Meltdown [44] vulnerabilities exemplified the consequences of optimizing for performance first and fixing security issues in hardware afterwards. The interconnected design considerations again make it more clear that security needs to become a first class constraint in design methodologies. Understanding how software, hardware, and security interact is the first step towards unifying security to design methodologies. This systemization provides the insights that can be translated into systematic rules of thumb to effectively guide the codesign process. In other words, rules of thumb for security design considerations and their implications must be communicated to software and hardware experts that provide reasonable actionable steps when building a system.

There is already precedence in the hardware-software codesign space towards establishing systematic rules to guide codesign opportunities. For instance, hardware designers often propagate design guidance to software designers that integer computation is preferred over floating point because it is more power and area efficient. Thus, a good rule of thumb for software designers who are not aware of the underlying hardware implementation is to use integer math whenever possible. Similar rules of thumb to guide hardware and software designers towards security-friendly designs are necessary to both establish this abstraction boundary and integrate security as a first class design consideration.

Our systemization study reveals that each technology needs to communicate unique design constraints and implementation challenges to the hardware and software. For example, for TEEs and enclaves, a designer should minimize the amount of data transferred into and out of the enclave boundary in the application design to reduce the performance overhead. For homomorphic encryption, an application designer should choose applications which are more amenable to vectorization and have limited control flow to minimize noise budget requirements. For DP, selecting the right privacy budget to ensure that both application behaviors and any correlated information leakages at lower levels of the stack are sufficiently hidden. Providing these succinct systematic guidelines between software, hardware, and security allows designers to translate the insights in one domain and effectively apply them to optimize design constraints in another. More importantly, it provides understandable design rules of thumb to guide cross-stack optimization.

4.2 Expose Usable Abstractions for Effective Codesign Interactions

The second key theme that our study exposes is the trend towards interfacing primitives between software, hardware, and security. In order to either automate or make the knowledge transfer between domains efficient, we need to provide a well-established vocabulary at the interface to collaborate effectively. This is similar to how the instruction set architecture establishes a contract between what the software sees and what the hardware implements in modern architectures and accelerators (a hardware accelerator is essentially one large instruction). The abstraction also serves as an interface between disciplines that allow experts in disparate fields to better communicate the challenges and systematic guidance towards co-optimizing solutions.

In the context of software-hardware-security codesign, a similar contract between security and software, and security and hardware remains an open question. For some technologies, these abstraction boundaries and the primitives to leverage them are quickly maturing. For instance, in DP, work in programming languages exposes a type system interface that guarantees DP of an application if properly leveraged and implemented against the API. Work in verification and static analysis [53] augments this interface by providing a way for the application to verify that the DP security guarantees are properly implemented for the application.

Similarly, for secure enclave implementations like SGX, the security mechanism exposes an instruction set extension to provide the application developer an interface to leverage the security mechanisms provided by the enclave. Homomorphic encryption also has a similar abstraction interface where a limited SIMD instruction set provides an interface for the application to reconcile constraints against the technology; there are several early proposals to provide compiler support to make the technology more accessible to programmers [16, 17]. These abstractions are all important because they help manage the complex design dependencies where designers can reason about the security guarantees.

4.3 Uphold Abstraction Boundary Contracts

Establishing clean abstraction boundaries and providing primitives to communicate across software, hardware, and security domains is useful to translate design-time requirements, but does not provide any guarantees on whether these requirements are fulfilled. In other words, we need to verify that the contract between software and security or hardware and security is upheld. For instance, a TEE is not useful if it provides an ISA but does not implement the hardware or software in a way that upholds the security guarantees that it claims. As a result, it is also important to verify that the software, hardware, and security properties that are exchanged for complying with these abstraction interfaces are in fact implemented correctly.

For instance, an application can receive confidentiality guarantees (for data) and integrity guarantees (for code and data) via hardware support for properly partitioned programs in TEEs. Here, software is responsible for providing a partitioning that properly leverages the TEE interface. In the case of enclaves like SGX, contract between software, hardware, and security is verified via remote attestation in which the software verifies that the software verifies that the hardware system that is offering the secure execution guarantees is genuine. The contract that the hardware implementation of the enclave is correct and implemented correctly is rooted in the manufacturer (i.e., security to hardware contract).

A similar set of contracts appears in the design considerations for homomorphic encryption. In HE, the software interfaces with security through noise budget constraints and encryption parameters. If the software implementation satisfies the noise budget

constraints using the provided encryption parameters, HE guarantees computing a correct result; a violation of this contract (i.e., exceeding noise budget) results in the decryption failure scenario which is can be verified with testing. In HE, an application also must conform to the restricted HE instruction set which is a canonical software-hardware contract but in this case the contract also includes restrictions from security.

Moving forward, it will be important to establish both the abstraction boundaries and verifiable guarantees. Threat models and root of trust definitions provide these interfaces to some degree but still are missing the verification aspects. Threat models provide the desired security requirements via adversary capability definitions but do not specifically address mechanisms for how threats are verified to be mitigated by software or hardware. Many of these abstractions boundaries and contracts remain an open question as to what the correct verifiable manifestations is. For instance, it is not clear what a contract may look like between DP and the hardware implementation of a DP software. Establishing ways to verify these contracts thus remains an area of future work that can bolster the design of private and secure systems.

5 LOOKING FORWARD

Our systemization illustrates that the codesign process for secure computing technologies is an interdisciplinary process which currently requires deep specialized knowledge across applications, hardware, and security to conduct. In other words, only an expert or group of experts who understand all three disciplines can successfully perform the codesign exercise. Going forward, to facilitate wider adoption and deployment, experts who understand the software-hardware-security codesign considerations will need to build automation support and infrastructure to abstract away the expertise requirements so that non-experts can focus on building on top of these technologies. To do this, we need to build accessible and robust tooling infrastructure similar to how LLVM and GCC compiler stacks abstracted away the complexities of compilation passes and optimization to allow programmers to focus on building applications.

Ongoing efforts towards these types of tools include compilation support for homomorphic encryption [16, 17], automatic partitioning for SGX [42], and programming abstractions for differential privacy [7, 51, 63]. In each case, the tools abstract away the details of how the underlying secure computing technology is implemented and automates the specialized expertise so that it can be reused by non-experts. This allows a designer who may have no knowledge of these secure computing technologies to still leverage the technology. While many efforts in this space are still nascent, these are the sorts of tools that will enable wider adoption by designers and put secure computing technologies into practice.

6 CONCLUSION

We present a systemization of knowledge for secure computing technologies and highlight opportunities for software-hardware-security codesign. For each technology, we identify the codesign considerations across software, hardware, and security, and expose the design feedback considerations that mandate co-optimization across each to guide efficient system integration. Insights into both emerging technologies as well as historical precedence can guide similar codesign efforts for future technologies. By recognizing the need for co-optimizing secure computing solutions in both historical and modern secure computing techniques, we hope it will inform more systematic and efficient deployments in the future.

REFERENCES

[1] 1997. Announcing Development of a Federal Information Processing Standard for Advanced Encryption Standard.

[2] 2019. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process.

[3] 2020. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process.

[4] Tiago Alves. 2004. Trustzone: Integrated hardware and software security. *White paper* (2004).

[5] Ross J. Anderson, Eli Biham, and Lars R. Knudsen. 1998. Serpent: A Proposal for the Advanced Encryption Standard.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 689–703.

[7] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin Pierce. 2016. Programming language techniques for differential privacy. *ACM SIGLOG News* 3, 1 (2016), 34–53.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 13.

[9] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. 1998. MARS - a candidate cipher for AES.

[10] TH Hubert Chan, Kai-Min Chung, Bruce M Maggs, and Elaine Shi. 2019. Foundations of differentially oblivious algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2448–2467.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[12] Woo-Seok Choi, Matthew Tomei, Jose Rodrigo Sanchez Vicarte, Pavan Kumar Hanumolu, and Rakesh Kumar. 2018. Guaranteeing local differential privacy on ultra-low-power systems. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 561–574.

[13] 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.

[14] Joan Daemen and Vincent Rijmen. 1999. AES proposal: Rijndael. (1999).

[15] Andreas Dandalis, Viktor K Prasanna, and Jose DP Rolim. 2000. A comparative study of performance of AES final candidates using FPGAs. In *International workshop on cryptographic hardware and embedded systems*. Springer, 125–140.

[16] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 546–561.

[17] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 142–156.

[18] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+ abort: A timer-free high-precision l3 cache attack using intel {TSX}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 51–67.

[19] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*. Springer, 1–19.

[20] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[21] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust.. In *Ndss*, Vol. 12. 1–15.

[22] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1054–1067.

[23] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[24] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

[25] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA) *(STOC '09)*. ACM, New York, NY, USA, 169–178.

[26] Craig Gentry. 2010. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* 53, 3 (March 2010), 97–105.

[27] Craig Gentry and Shai Halevi. 2011. Implementing gentry's fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 129–148.

[28] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. 201–210.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[30] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 897–912.

[31] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 490–505.

[32] Tetsuya Ichikawa, Tomomi Kasuya, and Mitsuru Matsui. 2000. Hardware Evaluation of the AES Finalists.. In *AES Candidate Conference*, Vol. 2000. 279–285.

[33] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S $ A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 591–604.

[34] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 85–96.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. Gazelle: A low latency framework for secure neural network inference. *arXiv preprint arXiv:1801.05507* (2018).

[36] Asawaree Kalavade and Edward A Lee. 1993. A hardware-software codesign methodology for DSP applications. *IEEE Design & Test of Computers* 10, 3 (1993), 16–28.

[37] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[38] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[39] Geoffrey Keating. 1999. Performance analysis of AES candidates on the 6805 CPU core. In *Proceedings of the second AES Candidate Conference*. 109–114.

[40] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2017. Accessing data while preserving privacy. *arXiv preprint arXiv:1706.01552* (2017).

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

[42] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 285–298.

[43] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. Armageddon: Cache attacks on mobile devices. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 549–564.

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.

[45] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 203–215.

[46] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. 2016. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro* 36, 5 (2016), 8–16.

[47] 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.

[48] Matthew Maycock and Simha Sethumadhavan. 2015. Hardware enforced statistical privacy. *IEEE Computer Architecture Letters* 15, 1 (2015), 21–24.

[49] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[50] Frank McSherry and Kunal Talwar. 2007. Mechanism design via differential privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. IEEE, 94–103.

[51] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 19–30.

[52] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramaniam, and Murali Annavaram. 2019. Privacy-Preserving Inference in Machine Learning Services Using Trusted Execution Environments. *arXiv preprint arXiv:1912.03485* (2019).

[53] Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[54] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. 2001. Report on the development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology* 106, 3 (2001), 511.

[55] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1429–1446.

[56] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

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

[58] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S Lee, and David Brooks. 2020. Cheetah: Optimizations and Methods for PrivacyPreserving Inference via Homomorphic Encryption. *arXiv preprint arXiv:2006.00505* (2020).

[59] Jason Reed and Benjamin C Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 157–168.

[60] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. 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*. 1295–1309.

[61] Ronald L Rivest, Matthew JB Robshaw, Ray Sidney, and Yiqun L Yin. [n.d.]. The RC6TM block cipher.

[62] Aaron Roth and Tim Roughgarden. 2009. *The median mechanism: Interactive and efficient privacy with multiple queries*. Technical Report.

[63] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for MapReduce.. In *NSDI*, Vol. 10. 297–312.

[64] Théo Ryffel, Edouard Dufour-Sans, Romain Gay, Francis Bach, and David Pointcheval. 2019. Partially encrypted machine learning using functional encryption. *arXiv preprint arXiv:1905.10214* (2019).

[65] Rathindra Sarathy and Krishnamurty Muralidhar. 2011. Evaluating Laplace noise addition to satisfy differential privacy for numeric data. *Trans. Data Priv.* 4, 1 (2011), 1–17.

[66] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. 1999. *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley & Sons, Inc.

[67] 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.

[68] Wenhao Wang, Yichen Jiang, Qintao Shen, Weihao Huang, Hao Chen, Shuang Wang, XiaoFeng Wang, Haixu Tang, Kai Chen, Kristin Lauter, et al. 2019. Toward Scalable Fully Homomorphic Encryption Through Light Trusted Computing Assistance. *arXiv preprint arXiv:1905.07766* (2019).

[69] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 83–93.

[70] 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.