2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

SPEED: Accelerating Enclave Applications via Secure Deduplication

Helei Cui*†, Huayi Duan†‡, Zhan Qin§¶, Cong Wang†‡, and Yajin Zhou§
*School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China
†Department of Computer Science, City University of Hong Kong, Hong Kong, China
‡City University of Hong Kong Shenzhen Research Institute, Shenzhen 518057, China
§College of Computer Science and Technology, Zhejiang University, Hangzhou, China
¶Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, Hangzhou, China
cuihelei@outlook.com, hduan2-c@my.cityu.edu.hk, {qinzhan, yajin_zhou}@zju.edu.cn, congwang@cityu.edu.hk

***Abstract*—The emerging hardware-assisted security technologies facilitate the deployment of secure and trustworthy applications in today's cloud computing infrastructure. Despite promising, the advantages appear to diminish due to limited resources of trusted execution environments and ever-increasing workload to be processed inside. Different from existing task-specific and system-level optimizations, our key observation is that those redundant computations occur commonly among several applications when handling the same input data.**

In light of this, we propose SPEED, a secure and generic computation deduplication system in the context of Intel SGX. It allows SGX-enabled applications to identify redundant computations and reuse computation results, while protecting the confidentiality and integrity of code, inputs, and results. To maximize the benefit of computation deduplication, we design a cross-application deduplication scheme, empowering multiple applications to securely utilize the shared results as long as they perform identical computations. To ease the use of SPEED, we implement a fully functional prototype and provide a concise and expressive API for developers to deduplicate rich computations with minimal effort, as few as 2 lines of code per function call. Extensive evaluations of four popular applications demonstrate that SPEED improves performance by up to 400 times. The source code is available on GitHub for public use.

I. Introduction

Under rapid commoditization of hardware-assisted security technologies, especially Intel SGX [1], today's cloud vendors are reacting swiftly by bringing Trusted Execution Environments (TEEs) to their next-generation security infrastructures, like IBM Cloud Data Guard [2] and Azure Confidential Computing [3]. Along with this trend are the growing interests in migrating various applications or computation tasks to TEEs (namely *enclave*) featured by SGX, such as anonymity network [4], network middlebox [5], and big data analytics [6]–[8]. Despite promising, the performance could become a severe issue due to relatively limited secure resources and ever-increasing workload to be fed into these enclave applications [9]–[11], especially when running on shared physical machines in this cloud computing paradigm.

To accelerate enclave applications inside TEEs, a lot of efforts has been made recently, e.g., using asynchronous system calls to reduce the performance impact of thread synchronization [9], and involving exit-less remote procedure calls to mitigate the cost of enclave exit [10]. While the system-level optimizations may work effectively, there remains a need to further accelerate these applications from a new angle, by eliminating redundant computations appeared in some enclave applications. This often happens when a cloud-based application encounters repeated input data (even from different requesters), such as VirusTotal Scanner [12], Turnitin Plagiarism Checker [13], and Google Safe Browsing [14]. Besides that, incrementally updated datasets are constantly being processed by the same or similar computing tasks, such as feature extraction for machine learning, index building for fast queries, and data aggregation for truth discovery. It is obviously crucial to *cache and reuse* the results of previous computations whenever possible.

In the literature, such technique, namely *memoization* [15] or *computation deduplication* [16], has been intensively studied. Though the detailed implementations and the targeted application scenarios might be different, the general idea is to check if a function call with the same (or similar) input data has been done before and the result can be directly obtained without re-execution. For example, the design proposed by Tang and Yang [16] targets the general computation deduplication across multiple applications or tasks, which binds a specified function's name (or code), input data, and output result together via a collision-resistance hash function. Besides that, many prior systems focus on a specific application for optimizing incremental bulk data processing, e.g., MapReduce [17], [18], C/C++ compiler [19], [20], and DryadLINQ [21]. In addition, there is also a line of work (e.g., [22]–[24]) extend this idea to approximate computation deduplication, i.e., some error-resilient applications (e.g., the emerging recognition, mining, and synthesis applications) can share the common processing results when facing highly-correlated (or similar) input data.

In light of these, we believe bringing the benefit of computation deduplication to the emerging cloud-based security infrastructures is a promising way to accelerate enclave applications running inside TEEs. However, enabling secure and generic computation deduplication in the context of hardware enclaves is still a non-trivial task for the following reasons.

*Most of the work was done when Helei Cui was a postdoc at CityU.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00110

First of all, we need to consider how to bind a particular computation to its result, so as to enable an enclave application to identify and reuse previously computed results directly rather than re-compute them. Different from data deduplication where the duplicates are identified via the data hash only, determining whether two deterministic computations (i.e., given the same input, it always produces the same result) are identical requires considering both function's code and input data [16]. During runtime, a tag can be derived from the combination of code and input data for each function call (computation). Two computations are considered duplicated if their tags are identical. The results of all fresh computation are stored once, and can be reused for any subsequently detected duplicates without re-executing the computations.

Second, we need to consider how to manage reusable results efficiently and securely. Ideally, the results and associated metadata containing sensitive information could naturally reside in an enclave and hence be protected there. But apparently, it is not scalable to maintain such data, which grows as more computations are executed, within the highly constrained enclave space, where the protected memory space (i.e., Enclave Page Cache (EPC)) is very limited for maintaining a small trusted computing base (TCB) [9]. To this end, we need to encrypt these reusable results as well as metadata, and store them outside the enclave. Relevant entries are loaded to the enclave in an on-demand manner.

And finally, we need to consider how to share these encrypted results between different applications for maximizing the result utilization, while limiting the availability to the ones who indeed perform the same computation, i.e., owning the same function's code and input data. We note that a straightforward approach by sharing a system-wide secret key among all applications would be vulnerable to the potential single point of compromise. Thus, we resort to the most prominent cryptographic primitive in secure data deduplication, namely message-locked encryption (MLE) [25]. Different from data deduplication, both the tag for duplicate checking and the key for result encryption are derived from the combination of a function's code and input data. Thus, this can avoid the procedure of key agreement in computation deduplication.

Taking all these into account, this paper presents SPEED, a secure and generic computation deduplication system in the context of Intel SGX. Particularly, SPEED consists of two major components: a secure deduplication runtime as a *trusted library* linked against application enclaves, and a generic encrypted result store. SPEED also provides a concise and expressive API for developers to securely deduplicate rich computations in their SGX-enabled applications with minimal effort. Developers can enjoy truly transparent development experience when importing such an attractive feature. The major contributions are listed as follows:

- We propose the first secure and generic computation deduplication system SPEED for accelerating the SGX-enabled applications. SPEED can protect and reuse computation results across multiple applications.
- We implement a fully functional prototype with Intel SGX SDK. It minimizes the developer effort in deduplicating rich computations among these SGX-enabled applications. As few as 2 lines of code modifications are needed for each function call of target computation.
- We provide concrete case studies on four popular applications, i.e., image feature extraction, data compression, pattern matching, and bag-of-words (BoW) computation. Extensive evaluations on real SGX-enabled machines show that SPEED is more suitable for those time-consuming computations, e.g., up to 90× speedups for SIFT feature extraction [26] and 400× speedups for pattern matching [27].

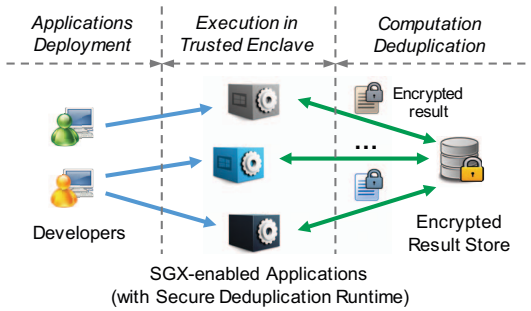


Fig. 1. An overview of SPEED workflow.

II. System Overview

A. System Model

Fig. 1 overviews the workflow of SPEED, which involves four major entities: SGX-enabled application, application developer, secure deduplication runtime (abbr. DedupRuntime), and encrypted result store (abbr. ResultStore). In order to improve the security and privacy of cloud-based applications, the developers are willing to harden their applications with the recent advancement in hardware-assisted security technologies (especially Intel SGX [1]). Next, these applications will be deployed on physical or even virtual machines (e.g., today's cloud computing infrastructure enables flexible deployment of various applications as virtual appliances running on shared hardware [28]) with the support of hardware enclaves. And then they will be run in the trusted execution environments. This is a popular trend with the rapid commoditization of SGX-enabled CPUs [4]–[8].

Meanwhile, we observe that it is very likely to find repeated or overlapped computations among these cloud-based applications, e.g., pattern matching may occur repeatedly over redundant files in an online virus scanner. Based on this, we consider if a particular computation is deterministic yet time-consuming (e.g., either the underlying algorithm is computational complex or the input size is relatively large), it would be more efficient to cache and reuse the same (and preferably small-sized) result rather than re-computing it [16].

Therefore, when implementing such an SGX-enabled application, the developer needs to mark those computations by modifying the function calls with our concise and expressive API. When reaching the marked function during the execution of the application, the secure DedupRuntime, which is associated with the application and responsible for duplicate

checking, will first generate a tag from the combination of the function's code and input data to represent the computation. Then the tag will be forwarded to an encrypted ResultStore, which manages previous computation results, to check whether the computation has been done before. If the answer is "no", it means that the result of the target computation has not been stored yet; therefore, the application will compute the result, then encrypt it properly with DedupRuntime, and eventually store it at the encrypted ResultStore for later use. Otherwise, the corresponding result will be decrypted and reused directly.

For ease of presentation, we use $result \leftarrow \texttt{func}(input)$ to represent a generic computation, where "`func`" implies the actual code of the function call, and the parameter is also viewed as a part of input data.

Discussion on usage scenario. SPEED aims to help the developers (or users) to deduplicate these potentially repeated or overlapped computations of SGX-enabled applications during the runtime in the public cloud. A developer can flexibly decide which computation (function call) needs to be marked with SPEED API. During the runtime, SPEED will only cache and reuse the results of those marked computations. Remarkably, the cached results will be encrypted and only be available to applications that also perform the same computations.

B. Threat Model and Assumptions

Like previous work [6], [7], [29], we assume a powerful adversary who has the ability to control the software stack of physical machines, including hypervisor and OS, but is unable to compromise the trusted hardware enclaves and relevant enclave keys. Particularly, it can exploit a vulnerability in the kernel or gain root access to the OS to observe and modify the encrypted results outside the trusted enclaves. It also attempts to obtain the result via a piece of short information about a computation, explicitly the tag derived from its underlying function's code and input.

Besides, we assume that the SGX-enabled application developers use our SPEED API correctly and focus on the common computations with deterministic results. This is consistent with all prior computation deduplication work (e.g., [15], [16]). We also assume that the integrity of an application is correctly verified before actually running with hardware enclaves, so the result will be correctly computed inside enclaves. And this can be achieved by the attestation mechanism of Intel SGX. In particular, the current SGX architecture supports two forms of attestation: one is a basic (local or intra-platform) assertion between enclaves running on the same platform; and the other one allows an enclave of a particular remote device to present reliable evidence about the running code, where only unmodified code can be run on a genuine processor.

Lastly, like many prior SGX systems [4], [9], [30], we do not consider the recently disclosed side-channel attacks (e.g., [31]) in this work. They can be orthogonally addressed by corresponding countermeasures (e.g., [32]).

C. Design Goals

• **Confidentiality and integrity.** SPEED has to protect the function's code, input data, and computation results, even after they leave the protected memory boundary of the originally trusted enclave. Note that by using the deduplication functionality of SPEED, an application will inevitably know whether an intended computation has been done before. Yet, SPEED has to ensure that this is the only information known to it beyond the computation result. ResultStore is also aware of such deduplication result, which is consistent with other secure deduplication schemes (e.g., [25], [33]–[35]). Also, note that the stored computation result should only be available to the application that can indeed perform the computation.

• **Generality and extensibility.** SPEED has to be designed and implemented in a function-agnostic way with a uniform serialization interface, so as to be compatible with different functions intended for deduplication. Meanwhile, to support a new function, the developer effort should be minimized, e.g., creating a "deduplicable" version of the function via a concise and expressive wrapper API. These are important because significant re-implementing of the entire application would definitely reduce the usability of SPEED.

D. Preliminaries

• **Hardware enclaves.** The recent advance in hardware enclaves of computer processor makes it possible to execute arbitrary application code over sensitive data at native speed without requiring trust in anything but the processor and the application. Despite the underlying implementations varying among different platforms (e.g., Intel SGX [1] and AMD TrustZone [36]), they all provide isolated execution environments. That is, a running enclave is protected by the processor, where its memory cannot be read or wrote by other processes on the same processor outside the enclave, not even the OS and hypervisor. Without losing generality, we focus on Intel SGX in this paper because of its popularity in academia and industry. We believe our high-level idea is also compatible with other platforms.

• **Message-locked encryption (MLE).** MLE [25] is originally formalized for ensuring data confidentiality in secure data deduplication, where the ciphertexts of unpredictable messages cannot be distinguished by an efficient attacker except with negligible probability. In brief, MLE ensures that the same data always result in identical tags for the use of duplicate checking, where the ciphertext could be randomized in some constructions, e.g., randomized convergent encryption (RCE) [25].

• **Authenticated encryption.** Authenticated encryption or its variant *authenticated encryption with associated data* (AEAD) is a kind of encryption scheme that simultaneously provides confidentiality, integrity, and authenticity assurances on the data. In brief, AEAD encrypts and authenticates plaintext data together with authenticated-only data, and produces ciphertext with an authentication code. Later, when performing the decryption and verification routine, any modifications would be detected. Here, we use AES in GCM mode [37], which is a high-performance AEAD scheme and is also provided by the crypto library shipped with Intel SGX SDK.

III. Our Proposed Design

A. Design Intuition

Our design intuition mainly introduces how to accelerate generic computations of an outsourced application inside the trusted execution environments (i.e., hardware enclaves) via securely reusing previous computation results, where these results are well protected and can be accessed by eligible applications that indeed perform the same computations, i.e., owning the same function's code and input data. As mentioned before, we focus on the common and repeated computations either in a single application or among different ones, where each computation can be viewed as a combination of code (of a particular function) and input data [16], i.e., "$\texttt{func}(input)$". Hence, different from conventional data deduplication, where the redundant copies can be identified via the hash of the data only, determining whether two computations are identical requires considering both the function's code and its input data.

In this direction, we first consider ***how to make an application aware of these redundant computations during its runtime (i.e., deduplication occurs before actually executing the underlying functions)***. As is often the case, calling a popular function from a third-party library within different applications or even a self-defined but reusable function within a single application will cause redundant computations when facing the same input data. Therefore, when implementing an SGX-enabled application, the developer should be able to mark those potentially common and repeated computations with a generic software framework, which contains a secure DedupRuntime for transparently handling the underlying deduplication operations, and an encrypted ResultStore for result management. Later, when the application starts running, DedupRuntime can intercept the marked computation, and query ResultStore with a tag derived from the combination of the function's code and input for duplicate checking, as shown in Fig. 2. Note that two computations are considered duplicated if their tags are identical.

Regarding those reusable results managed by ResultStore, we prefer keeping only small-sized metadata inside the enclaves and storing their actual content outside due to the limited protected memory resource. So we need to consider ***how to protect these valuable results and make them available to applications that perform the same computation***. Remarkably, with the strong protection of hardware enclaves, the confidentiality and integrity of computation results can be guaranteed, where they are encrypted and authenticated before leaving the trusted environment. Thus, a basic idea is to share a system-wide secret key among all trusted applications as adopted in [16]. However, this approach would be vulnerable to the potential single point of compromise.

To resolve this tension, we need to leverage an encryption scheme, of which the key is derived from computation itself; hence the application that performs the same computation can recover the key without agreeing on a single key in advance. MLE [25] is such a scheme with the keyless property, which

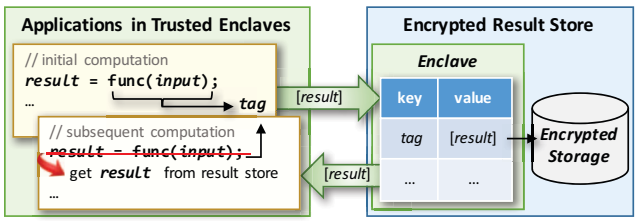


Fig. 2. A high-level idea of SPEED: The result of a fresh computation is encrypted and stored once, and can be reused for the subsequently detected duplicates without re-executing the computation.

enables subsequent uploaders to derive the same encryption key from the same data. However, as it was originally designed for secure data deduplication, we still need to make it suitable for our target computation deduplication scenario and compatible with the standard encryption scheme in SGX (more details in Section III-C). In what follows, we start from a basic design that a single secret key is used for the result encryption, regardless of the number of applications that can access the encrypted ResultStore.

B. Our Basic Design

For ease of exposition, we begin with a basic design as our starting point to introduce the general procedure of our proposed computation deduplication system SPEED. That is, in order to store these reusable results outside enclaves, they will be encrypted with a secret key k before leaving the originally trusted enclaves, and later they can be utilized by the same application or others as long as it owns the same key. Regarding the encryption scheme, we select a standard authenticated encryption scheme, e.g., AES in GCM mode, which provides data confidentiality as well as integrity. Here, we use $[res]$ to represent the result ciphertext, which covers its authentication code and initialization vector.

We now present the major routine of our computation deduplication design during the runtime of a deployed application, and leave the details on how to mark a target computation by developers to Section IV-C.

Specifically, before executing a computation $\texttt{func}(m)$ marked by the developer, the secure DedupRuntime of the application first generates a hash tag t from the combination of the function's code and its input data m, i.e., $t \leftarrow \texttt{Hash}(\texttt{func}, m)$, where $\texttt{Hash}(\cdot)$ is a collision-resistant hash function, e.g., SHA-256. Then the tag t is sent to the encrypted ResultStore via a secure channel for duplicate checking.

To record those reusable results, the encrypted ResultStore maintains a metadata dictionary D in the enclave, where each entry is indexed by the tag t and its value is the corresponding result ciphertext $[res]$. Particularly, the actual content of $[res]$ is stored outside enclave for space efficiency, just keeping a pointer in the metadata dictionary. So, if the target computation has not been done before, i.e., $null \leftarrow D.\texttt{get}(t)$, then the application will execute the function with its input, and obtain the result, i.e., $res \leftarrow \texttt{func}(m)$. Later, the result will be encrypted with the key k, i.e., $[res] \leftarrow \texttt{AES.Enc}(k, res)$, and forwarded to ResultStore for updating the metadata

Algorithm 1 Initial Computation with SPEED

Input: A computation $\texttt{func}(m)$ in an application, where m is the input data; and a metadata dictionary D managed by the encrypted ResultStore of SPEED.

Output: Computation result res, and updated D.

Inside the enclave of an application:

$t \leftarrow \texttt{Hash}(\texttt{func}, m)$;
Send t to the encrypted ResultStore via a secure channel;
Obtain $false$ as the response, which indicates the result of the target computation has not been stored yet, i.e., $null \leftarrow D.\texttt{get}(t)$;
$res \leftarrow \texttt{func}(m)$; *// Compute the result*
Pick a randomness $r \xleftarrow{R} \{0,1\}^*$;
$h \leftarrow \texttt{Hash}(\texttt{func}, m, r)$;
$k \leftarrow \texttt{AES.KeyGen}(1^\lambda)$;
$[res] \leftarrow \texttt{AES.Enc}(k, res)$; *// Encrypt the result*
$[k] \leftarrow k \oplus h$; *// Protect the key*
Send $(r, [k], [res])$ to the encrypted ResultStore;
return res;

Inside the enclave of ResultStore***:***

Update the dictionary via $D.\texttt{put}(t, (r, [k], [res]))$, where the actual content of $[res]$ is stored outside the trusted enclave for space efficiency;

dictionary, i.e., $D.\texttt{put}(t, [res])$. Otherwise, if ResultStore can locate the corresponding result, i.e., $[res] \leftarrow D.\texttt{get}(t)$, then the application owning the key k can directly obtain the result via $res \leftarrow \texttt{AES.Dec}(k, [res])$.

Discussion. We note that this single key design is useful when targeting the repeated computations within a single application. However, it is not robust enough when deduplicating the redundant computations among multiple applications, because agreeing on a single secret key will make the involved applications extremely brittle in the case of a single point of compromise [33]. To address this issue, we will present our main design of SPEED with an efficient encryption scheme that does not need to share a key for deduplication purpose.

C. Support Cross-Application Computation Deduplication

In order to maximize the utilization of those reusable results, we desire to make them available to all eligible applications without sharing a system-wide secret key in advance. To achieve this, we resort to the most efficient construction of MLE, i.e., RCE[1], and make it compatible with the standard encryption scheme in SGX. The detailed procedures for handling the initial computation and the subsequent computation are shown in Algorithm 1 and Algorithm 2, respectively.

Different from the above basic design, when performing the result encryption, the application does not require to use a system-wide secret key. Instead, it uses a randomly generated key via a standard key generation method, i.e.,

[1]Roughly, the encryption procedure in RCE is accomplished by first picking a random symmetric encryption key and then encrypting the message with that key. At last, this message encryption key is encrypted with another key, which is deterministically derived from the message itself, as a one-time pad [25].

Algorithm 2 Subsequent Computation with SPEED

Input: A computation $\texttt{func}(m)$ in an application, where m is the input data; and a metadata dictionary D managed by the encrypted ResultStore of SPEED.

Output: Computation result res.

Inside the enclave of an application:

$t \leftarrow \texttt{Hash}(\texttt{func}, m)$;
Send t to the encrypted ResultStore via a secure channel;
Obtain $true$ together with $(r, [res], [k])$ as the response, which indicates such computation result has been stored, i.e., $(r, [res], [k]) \leftarrow D.\texttt{get}(t)$;
$h \leftarrow \texttt{Hash}(\texttt{func}, m, r)$;
$k \leftarrow [k] \oplus h$; *// Recover the key*
$res \leftarrow \texttt{AES.Dec}(k, [res])$; *// Decrypt the result*
return res;

$k \leftarrow \texttt{AES.KeyGen}(1^\lambda)$. Then this random encryption key k is encrypted via XORing a secondary key h, i.e., $[k] \leftarrow k \oplus h$. Specifically, to obtain h, the DedupRuntime of the application picks a randomness $r \xleftarrow{R} \{0,1\}^*$ as a challenge message, then attaches it with the combination of function's code and input data m, and computes it via $\texttt{Hash}(\texttt{func}, m, r)$. Finally, DedupRuntime needs to send $(r, [k], [res])$ for updating the metadata of reusable results at the encrypted ResultStore.

In the above design, the results can still be encrypted with a standard authenticated encryption scheme as our basic design (e.g., AES in GCM mode), which protects the confidentiality and integrity simultaneously. And this also ensures that the applications, performing the same computation, can always recover the corresponding random encryption key k, so as to decrypt the result ciphertext. But different from the original RCE scheme, the involved challenge message r further ensures that the encrypted result can be decrypted correctly if and only if the application indeed performs the identical computation, i.e., owning the same function's code and input data. And this is verified by DedupRuntime executed in the enclave of the application, as shown in Fig. 3. In brief, if the application is not capable of computing the result by itself, then it cannot succeed in result decryption.

D. Security Analysis

In the following part, we analyze the security strength of SPEED. First of all, the security of the applications relies mainly on hardware enclaves, which guarantees the confidentiality and integrity of involved code, inputs, and results inside the trusted execution environments. Besides, the data flow outside enclaves, which are involved in our deduplication procedure, such as the tag t, the challenge message r, and the result ciphertext $[res]$, are all encrypted and authenticated with standard cryptographic tools (e.g., AES in GCM mode). And this also prevents the cache poisoning attack [16], where an adversary attempts to poison ResultStore with bad results. Therefore, we turn our focus on the security of the result encryption scheme for the support of cross-application computation deduplication in Section III-C.

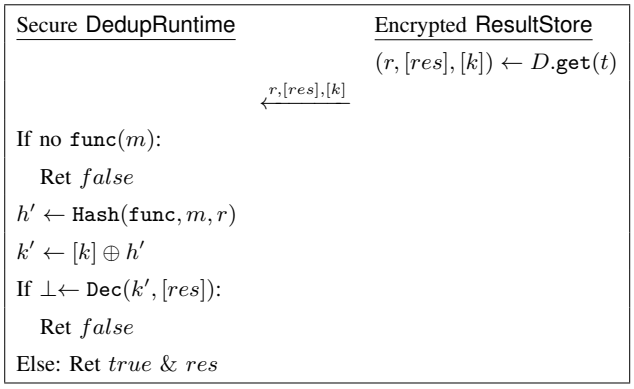


Fig. 3. The verification protocol running inside the trusted enclaves of SPEED. If the query application does not have the code "`func`" and its input "m", then it will not be able to recover the correct key k and decrypt the result ciphertext "$[res]$" (the symbol "$\perp$" indicates the attempted decryption does not pass the authenticity check).

Our result encryption scheme is built on top of RCE scheme, which is a randomized MLE scheme that achieves the best possible privacy for deduplication, i.e., the encryption of an unpredictable message must be indistinguishable from a random string of the same length [25]. The difference lies in two aspects: First, the tag t for computation deduplication is derived from the combination of the function's code and input data, instead of a message as in data deduplication. Second, the secondary key h for protecting the random symmetric encryption key k is computed with an additional challenge message r, which is randomly chosen by the initial computation and securely kept inside enclaves.

Therefore, the result ciphertext stored at ResultStore (even outside enclaves) is encrypted and authenticated. Meanwhile, even if a malicious application can obtain the result ciphertext $[res]$ together with $[k]$ and r by using some short information about the computation (i.e., the tag t), it still cannot correctly decrypt them unless it indeed performs the same computation (i.e., owning the same function's code and input data). We note that this allows SPEED to defend against the query forging attack [16] even in a leakage setting where the tag of a computation could be leaked.

In summary, our result encryption scheme does not degrade the security strength of the original RCE scheme, but further provides a verification mechanism, similar to [34]. So the equality information about the deduplicated computations can be limited to the applications with the same computations.

Additionally, we note that the offline brute-force dictionary attack [33] over predictable computation (i.e., both the underlying function's code and its input data are predictable) cannot be launched by an attacker who compromises the machine of ResultStore, because both the tag and the challenge message are protected with hardware enclaves in our target scenario.

Mitigating denial-of-service attacks. To deal with a reasonably high request volume, the design of our encrypted ResultStore is light-weight. However, a malicious application may issue a large number of "update" requests for polluting the ResultStore with useless results. To defend against it, we can adopt the rate-limiting strategy into SPEED, which involves a quota mechanism to limit the cache space for each application [16].

Discussion on controlled deduplication. In the above design, an application that performs the same computation can always derive the encryption key, so as to decrypt the corresponding result. However, such a "keyless" encryption scheme does not naturally provide flexible access control mechanism. To ensure that only authorized applications can access ResultStore, it requires an additional authorization mechanism [33].

Discussion on memory access pattern. Even though the reusable results are always encrypted outside enclaves, it may still raise the concern of leaking memory access pattern [7]. That is, the clear data inside enclaves (e.g., the result res and the corresponding random key k generated at the application, and the challenge message r stored at ResultStore) may be extracted via software side-channels [29]. We are aware that this issue can be addressed by integrating existing oblivious memory access solutions [29], [38]. However, this inevitably incurs extra overhead, and we will explore a good balance between security and performance in our future work.

IV. Implementation Details

A. SGX Programming Model

To facilitate the understanding of our implementation details, we first give a brief introduction to the SGX programming model. In general, hardening an application with hardware enclaves requires the developer to reconstruct and recompile the code with Intel SGX SDK. Specifically, these SGX-enabled applications must be partitioned into two counterparts: a *trusted enclave* for running critical code on sensitive data, and an *untrusted host* for running noncritical code and enclave management, such as creation, deconstruction, and communication. The enclave and the host interact with each other via two types of well-defined secure API, i.e., `ECALL` and `OCALL`. The former is called by the host to enter the enclave, and the latter is called by the enclave to access system utilities that are prohibited inside the enclave.

B. SPEED Core

At the heart of our SGX-enabled SPEED system is a secure DedupRuntime for transparently handling the underlying deduplication operations, and an encrypted ResultStore for managing previously computed results. Remarkably, they are both implemented in a function-agnostic way, so as to minimize the developer effort (see Section IV-C). We now present the implementation details of the two major components in our SPEED prototype.

• **Secure deduplication runtime.** It implements the main deduplication functionalities, e.g., intercepting marked function calls, querying ResultStore, and retrieving the possible computation results. As a trusted library linked against application enclaves, most code of DedupRuntime is executed in the enclave. It covers a set of function parsers for serialization, and customized `OCALL`s wrapping request and networking logic.

The main routine of DedupRuntime is to generate a hash tag from the combination of the target function's code and input data for duplicate checking. To this end, a direct approach is to connect the code and data together, and then compute the tag via a hash function. But in practice, this might become less effective when considering the difference caused by developer or compiler, e.g., the same code may be compiled into different executable files in different compilation environment. Therefore, to enhance the adaptability, our DedupRuntime takes the following two inputs. The first one is the *description* of a marked function, which includes library family, version number, function signature, and other relevant information, e.g., (`"zlib", "1.2.11", int deflate(···)`). With these, DedupRuntime can verify that the application indeed owns the actual code of the function by scanning the underlying trust library, and derive a universally unique value for function identification. The second input of this routine is the input data of the marked function, e.g., a file for compression. Then a hash tag t is computed via $\texttt{Hash}(\cdot)$ with the two inputs. After that, the control is passed to an `OCALL`, which prepares and sends a `GET_REQUEST` with the tag t to ResultStore for duplicate checking.

In our SPEED prototype, we implement synchronous communication. So the same `OCALL` needs to wait until receiving corresponding `GET_RESPONSE` replied by ResultStore. Once the `OCALL` returns and the control switches back to the enclave, if the response is positive, the associated data will be parsed, verified and decrypted. Otherwise, the input function is parsed and executed[2]; the results are authenticated, encrypted, and sent to the ResultStore as an asynchronous `PUT_REQUEST` via another `OCALL` in a similar way.

We note that the parsers, `OCALL`s and related data structures (e.g., `XXX_REQUEST` and `XXX_RESPONSE`) are implemented in a function-agnostic way with uniform serialization interface, so they are capable of handling different functions intended for deduplication. To support new function, for example from another trusted library, the only step is to associate it with a proper parser from existing ones or create a new one with customized serialization for the function's input and output.

• **Encrypted result store.** The implementation of ResultStore is relatively straightforward. The main data structure used here is an enclave-protected dictionary storing previous computation results keyed by the tag t. To maximize the utility of limited enclave memory, the dictionary entry is designed to be small: it maintains some metadata (e.g., challenge message r and authentication MAC), and a pointer to the real result ciphertexts that are kept outside the enclave.

Unlike DedupRuntime, the main body of encrypted ResultStore runs outside the enclave. Upon receiving a request, ResultStore first applies preliminary parsing, and then delegates the request to one of two customized `ECALL`s dependent on whether it is a `GET_REQUEST` or `PUT_REQUEST`. The duty of the `ECALL` is to marshal data at enclave boundary and access the dictionary inside the trusted enclave. After it returns, ResultStore prepares a corresponding `GET_RESPONSE` or `PUT_RESPONSE`, which is sent back to the requesting DedupRuntime.

Remark. Similar to the prior work [16], we consider deploying ResultStore at the same machine of the outsourced applications. We can also deploy a master ResultStore on a dedicated server, which periodically synchronizes the popular (i.e., frequently appeared) results from different machines. The application may not be able to access the latest computed results from others on different machines, and needs to compute and encrypt the result with a self-selected random key. Nevertheless, we emphasize that this will not cause redundancy at the master ResultStore. Because the tags of underlying computations are deterministic and only one version of result ciphertext (associated with corresponding $[k]$ and r) needs to be stored. And later, such result ciphertext can still be decrypted correctly by eligible applications that indeed perform the same computations.

C. API and Use Cases

To ease the use of our proposed deduplication system SPEED, we expose a concise and expressive API to the developer of SGX-enabled applications. The API is centered on a `Deduplicable` object, which wraps the interaction with underlying trusted DedupRuntime, conversion between data formats, and all other intermediate operations. Our current prototype uses extensive C++ template features in the design and implementation of `Deduplicable`, allowing it to accept, in principle, any functions[3]. To make a function deduplicable, the developer only needs to create a `Deduplicable` version by providing the aforementioned simple description, and then uses the new version as normal. This usually requires a change of only 2 lines of code per function call. Four concrete examples are shown in Fig. 4.

V. Experimental Evaluation

A. Experiment Setup

All experiments are run on two SGX-enabled machines with Intel Xeon E3-1505 v5 (4 cores @2.80GHz, 8MB cache) with 16GB of RAM, where the OS is Ubuntu 16.04 LTS and the SGX SDK[4] is v1.8. Particularly, we use `gcc-5.4.0` to compile the SGX applications, and the enclave memory is set to the maximum 128MB (90MB usable). Regarding the required collision-resistant hash function and authenticated encryption scheme, we use `SHA-256` and `AES-GCM-128`, respectively, both are provided by the crypto library shipped with SGX SDK.

For demonstration purpose, we select four popular basic applications existed in many other applications, and port them to SGX enclaves. The first one is the SIFT [26] feature extraction, which is a famous algorithm in computer vision applications, such as object recognition, image stitching, and

[2]Note that the required library itself (e.g., `zlib`) should be available as a trusted library, i.e., properly ported, at the applications.

[3]While the current API is in C++, SPEED can support C language as well via function pointers. We leave this feature to future work.

[4]Intel SGX SDK: https://software.intel.com/en-us/sgx-sdk/download

Case 1

```
// wrapper function
string sift_wrapper(const string& pic, int width,
  int height);

// original function call
string sift_feature = sift_wrapper(pic, width, height);

// modified function call with SPEED API
Deduplicable<string, const string&, int, int>
  dedup_func("libsiftpp", "0.8.1", "sift_wrapper",
  sift_wrapper);

string sift_feature = dedup_func(pic, width, height);
```

Case 2

```
// wrapper function
string zlib_compress_wrapper(const string& fileContext);

// original function call
string compressed = zlib_compress_wrapper(fileContext);

// modified function call with SPEED API
Deduplicable<string, const string&> dedup_func("zlib",
  "1.2.11","zlib_compress_wrapper",zlib_compress_wrapper);

string compressed = dedup_func(fileContext);
```

Case 3

```
// wrapper function
int pcre_pattern_matching_wrapper(const string& str);

// original function call
int matching_times = pcre_pattern_matching_wrapper(str);

// modified function call with SPEED API
Deduplicable<int, const string&> dedup_func(
  "libpcre", "10.23", "pcre_pattern_matching_wrapper",
  pcre_pattern_matching_wrapper);

int matching_times = dedup_func(str);
```

Case 4

```
// original function call
string str_bow = bow_mapper(str, delim);

// modified function call with SPEED API
Deduplicable<string, const string&, const string&>
  dedup_func("sgx_mapreduce", "0.1.0", "bow_mapper",
  bow_mapper);

string str_bow = dedup_func(str, delim);
```

Fig. 4. Code snippets of our four use cases: 1) image feature extraction via `libsiftpp`; 2) data compression via `zlib`; 3) text pattern matching via `libpcre`; 4) BoW computation via `mapreduce`. The function signature is passed as template parameters. Note: the wrapper functions in Case 1, 2, and 3 are used for normalizing/simplifying the underlying function calls, which can be customized by developers and are not restricted by SPEED.

3D scene modeling. The second one is the data compression, which is often used for bandwidth optimization [39]. The third one is the pattern matching, which appears in virus scanners (e.g., `ClamAV` [40]) and many other searching scenarios. The last one is the BoW computation on top of MapReduce framework, which is widely used in natural language processing and information retrieval [41].

Particularly, the exact functions we are going to deduplicate are `sift`$(\cdot)$ from the library `libsiftpp`[5], `deflate`$(\cdot)$ from the library `zlib`[6], `pcre_exec`$(\cdot)$ from the library `libpcre`[7], and `bow_mapper`$(\cdot)$ customized from the `Mapper`$(\cdot)$ function of the library `mapreduce`[8], respectively. Regarding the test data of Case 1 and 2, we randomly select different sized images from the Internet and text files from the Boost Library[9]. For Case 3, we select over 4 million valid network packets from the m57-Patents Scenario dataset[10] and 4SICS-2015 dataset[11] as input, and use over 3,700 patterns from Snort rules[12] in the matching task. For Case 4, we randomly select 300,000 web pages from the CommonCrawl dataset[13]. All experimental results represent the mean of 10 trials.

B. Evaluation

• **Developer effort.** To make a function deduplicable with SPEED, a developer needs to convert the original function call into the `Deduplicable` version. Fig. 4 has shown the actual modifications of four deduplicable functions in our exemplary applications. Recall that we mainly target the functions that will be executed in trusted environments, i.e., the original function should come from an SGX-enabled trusted library, provided by a third party or the developer herself. Nevertheless, we emphasize that this kind of development effort is inevitable if the function will be run inside the enclaves. Therefore, to deduplicate a wide range of computations in trusted enclaves, it only requires very little modifications, as few as 2 lines of code per function call.

• **Application performance.** To demonstrate the effectiveness of SPEED, we compare the running time of our ported four exemplary applications in three cases: without using SPEED (the baseline); initial computation with using SPEED (i.e., deduplication is not executed); and subsequent computation with using SPEED (i.e., deduplication is executed).

Fig. 5 shows the relative running time comparison under different input size/volume, e.g., about $76-94\times$ speedups for the SIFT feature extraction and $316-412\times$ speedups for the pattern matching (with over 3,700 rules), but only $3.8-4\times$ speedups for the data compression and $3.7-4\times$ speedups for the BoW computation. It can be shown that the feature extraction and pattern matching benefit more from the deduplication procedure, and the involved overhead for the initial computation varies for different applications, e.g., up to 34% delay for the BoW computation but less than 2% delay for the feature extraction. This is because the two tasks in Case 1 and 3 are relatively slow compared with the extra cost for results encryption, but the speed of the remaining two tasks are relatively fast that is on the same level as the introduced cryptographic operations (as shown in

[5]A lightweight C++ implementation of SIFT: http://vision.ucla.edu/~vedaldi/code/siftpp.html

[6]Data compression library: https://zlib.net

[7]PCRE (Perl compatible regular expressions) library: http://www.pcre.org

[8]A C++ MapReduce library: https://github.com/nmandal/MapReduce

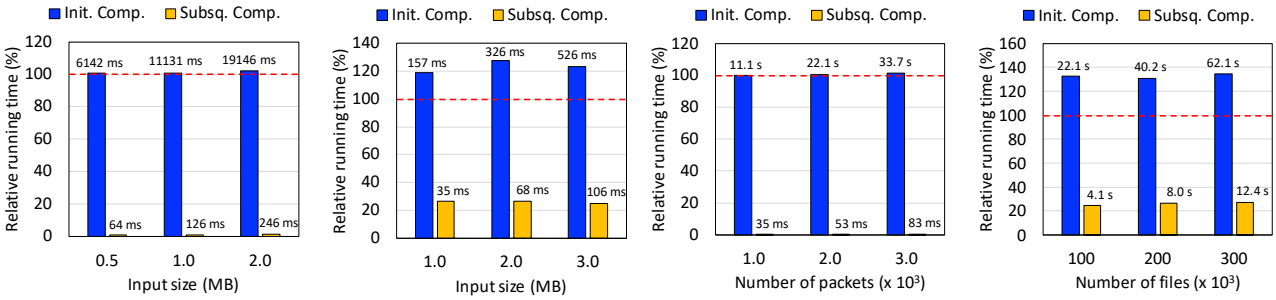
[9]Boost performance comparison: http://www.boost.org/doc/libs/1_41_0/libs/regex/doc/gcc-performance.html

[10]m57-Patents Scenario: a public trace file released by an IDS system Bro, http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario

[11]4SICS 2015 dataset: https://www.netresec.com/?page=PCAP4SICS

[12]Snort rules: https://www.snort.org/downloads\#rules

[13]WET files of October 2018 crawl archive: http://commoncrawl.org/



(a) Feature extraction via `libsiftpp` (b) Data compression via `zlib` (c) Pattern matching via `libpcre` (d) BoW computation via `mapreduce`

Fig. 5. The relative running time of four applications under different input size. The y-axis is the running time relative to the original application, where the red dashed line at 100% indicates its running time without applying SPEED. Note: "Init. Comp." (initial computation) represents the computation time cost (including the time for secure storing result) without deduplication, but "Subsq. Comp." (subsequent computation) is the time that deduplication is executed.

TABLE I
EXEMPLARY EVALUATION OF CRYPTOGRAPHIC OPERATIONS IN DEDUPRUNTIME UNDER FOUR DIFFERENT SIZED INPUT.

Input (KB)	Tag Gen. (ms)	Key Gen. (ms)	Key Rec. (ms)	Result Enc. (ms)	Result Dec. (ms)
1	0.028	0.062	0.048	0.015	0.021
10	0.186	0.159	0.145	0.031	0.022
100	1.198	1.182	1.169	0.188	0.049
1024	6.008	2.779	2.775	1.731	0.257

Table I). Thus, we conclude that SPEED is more suitable for deduplicating a time-consuming function (preferably with a small-sized result) in practice, which could be a single but complex computational algorithm or a sequence of processing tasks. And we emphasize that SPEED is with maximum generality and extensibility, which allows the developers to deduplicate their rich computations in SGX context.

• **Cryptographic operations.** To better understand the latency introduced to the application when applying our SGX-enabled computation deduplication system SPEED, we provide an exemplary evaluation of the performance of the major cryptographic operations in the secure DedupRuntime. As shown in Table I, the processing times of these operations are in linear to the size of input data, and the potential overhead is relatively small because the selected schemes are efficient. For example, it takes about 1.198 ms to generate a tag from a function with 100KB input data, and another 1.169 ms to recover the encryption key (shortened as "Key Rec.") if the result can be found. Meanwhile, the result encryption and decryption (the last two columns) are even faster with the same sized input, literally an order of magnitude. In addition, we note that once the result is computed by the application in the case of initial computation, the remaining "PUT" operations (including key generation and protection (shortened as "Key Gen."), result encryption, and update at ResultStore) can be processed in a separated thread for better efficiency.

• **Throughput evaluation.** To evaluate the throughput of our encrypted ResultStore, we use four different sized data (from

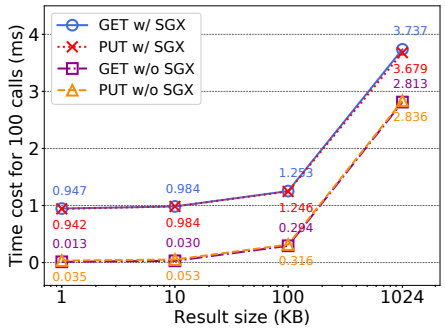


Fig. 6. Throughput evaluation and comparison of two major operations of ResultStore. Note that the both operations in SPEED are with SGX.

1KB to 1MB) to measure the time cost of processing the two types of request, i.e., `GET_REQUEST` and `PUT_REQUEST`. Fig. 6 shows the time cost of processing 100 times of each operation at ResultStore, where the incoming data are all different. It can be shown that both operations with SGX (i.e., the upper two lines) are very fast, and quite close. Meanwhile, Fig. 6 also shows the performance of the same operations without using SGX technique, i.e., running outside enclaves. From the results, we observe that the speed of each operation with SGX is much slower when facing a small sized result, e.g., 1KB. Note that the additional cost in SGX comes from the underlying control switches in `OCALL` and `ECALL` as mentioned in Section IV-B, and the gap is getting smaller with the growth of result size. To mitigate such overhead, we can further adopt off-the-shelf system-level optimization techniques, e.g., asynchronous system call mechanism [9] or exit-less remote procedure call mechanism [10].

VI. RELATED WORK

• **Redundancy elimination in network traffic and computation.** In the literature, the concept of redundancy elimination has been widely adopted in the areas of network traffic alleviation and computation speedup. For example, Anand *et al.* [42] designed a coordinated packet-level redundancy elimination architecture for the network traffic. Hua *et al.* [43] proposed to eliminate duplicate data within the network by

checking the data fingerprints in the SDN controller. Also, a similar idea has appeared in near-duplicate detection systems for high-quality content-centric applications [44], [45].

Apart from these, the techniques for computation deduplication have been explored for decades, where the core idea is to cache and reuse computation results. The early concept, which uses a global cache separated from a subject application to record results of potentially repeated computations, was proposed by Michie [15]. Later, several studies (e.g., [17]–[21], [46], [47]) extended this idea into different application scenarios with various caching strategies. Recently, Tang and Yang brought this idea to a more generic scenario and proposed UNIC [16], which enables applications to deduplicate their rich computations. However, UNIC mainly operates in plaintext domain except using a system-wide key to ensure the correctness of result, and does not consider the confidentiality of the cached results, which are stored unencrypted.

Different from prior designs, our work targets the SGX-enabled applications, and provides a secure and generic computation deduplication system, where the confidentiality and integrity of involved code, inputs, and stored results are well protected.

• **Deduplication over encrypted data.** Our work is also closely related to the studies on cross-user data deduplication over encrypted data (just to list a few) [25], [33]–[35], [48]. In [25], Bellare *et al.* introduced a cryptographic primitive message-locked encryption (MLE), and provided a formal definition to capture the best possible security for the purpose of secure deduplication. Despite very promising, MLE and its variants are inherently vulnerable to offline brute-force attacks over predictable data [25]. To resist this, the follow-up studies (e.g., [33], [48]) resort to an additional independent key server to obliviously provide message-derived encryption keys. Besides these, Liu *et al.* [49] proposed the first single-server scheme to defend such attacks, where a number of online users with the matched short hash would play the role of the additional server via the password authenticated key exchange (PAKE) protocol.

Different from these designs for data storage saving, our work aims to provide a secure computation deduplication system for accelerating the computations in the context of trusted enclaves.

• **Secure systems based on Intel SGX.** With the widespread deployment of Intel SGX-enabled CPU (since Skylake microarchitecture), there is an emerging trend towards securing different types of applications via hardware enclaves on untrusted platforms. Just to name a few, Schuster *et al.* implemented VC3 [6] that secures MapReduce applications in Hadoop framework. Kim *et al.* showed possible security enhancements in network applications [50], such as SGX-enabled software-defined inter-domain routing, P2P anonymity networks (Tor), and middleboxes. Hunt *et al.* introduced Ryoan [30] that allows users to run untrusted applications in a distributed sandbox while protecting their secret data. Recently, Zheng *et al.* designed Opaque [7] that is a secure outsourced data analytics platform on Spark SQL without suffering from access pattern leakage.

Given such popularity, how to reduce the performance overhead of SGX has become a demanding topic. One line of studies manages to eliminate the expensive context switches in enclave applications [51]. The idea is typically to perform `OCALL`s and `ECALL`s with multiple threads via some shared buffer, in an asynchronous way [52]. Another attempt is to design more efficient memory management scheme, for example by maintaining user-level page table in enclave for exit-less paging [10], or by customizing compact and cache-friendly data structures for fine-grained state management [53]. Since the current EPC has a limited size, yet another approach tries to just enlarge the usable EPC (without paging) in an efficient way. For instance, to facilitate growing EPC to the size of physical memory, a recent work improves the data structure for integrity verification [54].

Different from all of them, we look at the performance issue of SGX application from a new angle. That is, the potential redundancy in the computation itself. By deduplicating the repeated and often time-consuming computation, our approach has demonstrated notable performance gain in a wide range of applications protected by SGX. We believe it to be a valuable complement to the literature.

VII. Conclusion

In this paper, we propose SPEED, a generic system that enables secure computation deduplication over SGX-enabled applications. It enables these applications to identify redundant computations and reuse computation results, while protecting the confidentiality and integrity of the involved code, inputs, and results. To maximize the result utilization, our extended cross-application deduplication scheme empowers other applications to securely utilize the shared results, without sharing a system-wide key. To ease the use of SPEED, our fully functional prototype provides a concise and expressive API for developers to deduplicate rich computations with minimal effort, as few as 2 lines of code per function call. Extensive evaluations of four popular applications show that SPEED is more suitable for those time-consuming computations. The source code[14] is available on GitHub for public use.

As a future direction, we will explore an automatic extension to enable the application to adjust its deduplication strategy via dynamic analyzing the underlying computations during its runtime.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong under Grant CityU 11276816, Grant CityU 11212717, and Grant CityU C1008-16G, in part by the Innovation and Technology Commission of Hong Kong under ITF Project ITS/168/17, in part by the National Natural Science Foundation of China under Grant 61572412 and 61872438, and in part by the Fundamental Research Funds for the Central Universities.

[14]Source code: https://github.com/CongGroup/SPEED

REFERENCES

[1] Intel, "Software Guard Extensions (SGX)," https://software.intel.com/en-us/sgx.

[2] "A More Protected Cloud Environment: IBM Announces Cloud Data Guard Featuring Intel SGX," https://itpeernetwork.intel.com/ibm-cloud-data-guard-intel-sgx/, 2017.

[3] "Azure Confidential Computing," https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 2018.

[4] S. M. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments," in *Proc. of USENIX NSDI*, 2017.

[5] J. Han, S. Kim, J. Ha, and D. Han, "SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module," in *Proc. of ACM Asia-Pacific Workshop on Networking (APNet)*, 2017.

[6] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *Proc. of IEEE S&P*, 2015.

[7] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *Proc. of USENIX NSDI*, 2017.

[8] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "A Practical Encrypted Data Analytic Framework with Trusted Processors," in *Proc. of ACM CCS*, 2017.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. of USENIX OSDI*, 2016.

[10] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *Proc. of ACM EuroSys*, 2017.

[11] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database Using SGX," in *Proc. of IEEE S&P*, 2018.

[12] "VirusTotal," https://www.virustotal.com, 2018.

[13] "Turnitin," http://turnitin.com, 2018.

[14] "Google Safe Browsing," https://safebrowsing.google.com, 2018.

[15] D. Michie, "Memo Functions and Machine Learning," *Nature*, vol. 218, no. 5136, pp. 19–22, 1968.

[16] Y. Tang and J. Yang, "Secure Deduplication of General Computations," in *Proc. of USENIX ATC*, 2015.

[17] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: MapReduce for Incremental Computations," in *Proc. of ACM SoCC*, 2011.

[18] Y. Zhang, S. Chen, Q. Wang, and G. Yu, "i^2 MapReduce: Incremental MapReduce for Mining Evolving Big Data," *IEEE Trans. on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1906–1919, 2015.

[19] "ccache: a fast C/C++ compiler cache," https://ccache.samba.org.

[20] C. Dietrich, V. Rothberg, L. Füracker, A. Ziegler, and D. Lohmann, "cHash: Detection of Redundant Compilations via AST Hashing," in *Proc. of USENIX ATC*, 2017.

[21] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic Management of Data and Computation in Datacenters," in *Proc. of USENIX OSDI*, 2010.

[22] P. Guo and W. Hu, "Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications," in *Proc. of ACM ASPLOS*, 2018, pp. 271–284.

[23] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: A Cache for Approximate Computing," in *Proc. of ACM International Symposium on Microarchitecture*, 2015, pp. 50–61.

[24] Y. Tian, Q. Zhang, T. Wang, and Q. Xu, "Lookup Table Allocation for Approximate Computing with Memory Under Quality Constraints," in *Proc. of IEEE DATE*, 2018, pp. 153–158.

[25] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-Locked Encryption and Secure Deduplication," in *Proc. of EUROCRYPT*, 2013.

[26] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *IJCV*, vol. 60, no. 2, pp. 91–110, 2004.

[27] X. Yuan, H. Duan, and C. Wang, "Bringing Execution Assurances of Pattern Matching in Outsourced Middleboxes," in *Proc. of IEEE ICNP*, 2016.

[28] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," *ACM Trans. on Computer Systems*, vol. 33, no. 3, p. 8, 2015.

[29] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *Proc. of USENIX Security*, 2016.

[30] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *Proc. of USENIX OSDI*, 2016.

[31] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *Proc. of ACM CCS*, 2017.

[32] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory Safety for Shielded Execution," in *Proc. of ACM EuroSys*, 2017.

[33] M. Bellare, S. Keelveedhi, and T. Ristenpart, "DupLESS: Server-Aided Encryption for Deduplicated Storage," in *Proc. of USENIX Security*, 2013.

[34] J. Xu, E.-C. Chang, and J. Zhou, "Weak Leakage-Resilient Client-Side Deduplication of Encrypted Data in Cloud Storage," in *Proc. of ACM ASIACCS*, 2013.

[35] H. Cui, C. Wang, Y. Hua, Y. Du, and X. Yuan, "A Bandwidth-Efficient Middleware for Encrypted Deduplication," in *Proc. of IEEE DSC*, 2018.

[36] AMD, "Secure Technology," https://www.amd.com/en/technologies/security, 2019.

[37] D. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," *Submission to NIST Modes of Operation Process*, vol. 20, 2004.

[38] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proc. of NDSS*, 2017.

[39] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proc. of ACM SIGCOMM*, 2016.

[40] "ClamAV," https://www.clamav.net, 2018.

[41] X. Yuan, H. Cui, X. Wang, and C. Wang, "Enabling Privacy-Assured Similarity Retrieval over Millions of Encrypted Records," in *Proc. of ESORICS*, 2015.

[42] A. Anand, V. Sekar, and A. Akella, "SmartRE: An Architecture for Coordinated Network-wide Redundancy Elimination," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 87–98, 2009.

[43] Y. Hua, X. Liu, and D. Feng, "Smart In-Network Deduplication for Storage-Aware SDN," in *Proc. of ACM SIGCOMM*, 2013.

[44] Y. Hua, W. He, X. Liu, and D. Feng, "SmartEye: Real-Time and Efficient Cloud Image Sharing for Disaster Environments," in *Proc. of IEEE INFOCOM*, 2015.

[45] H. Cui, X. Yuan, Y. Zheng, and C. Wang, "Enabling Secure and Effective Near-Duplicate Detection Over Encrypted In-Network Storage," in *Proc. of IEEE INFOCOM*, 2016.

[46] A. Heydon, R. Levin, and Y. Yu, "Caching Function Calls Using Precise Dependencies," in *Proc. of ACM PLDI*, 2000.

[47] C. Tan, L. Yu, J. B. Leners, and M. Walfish, "The Efficient Server Audit Problem, Deduplicated Re-execution, and the Web," in *Proc. of ACM SOSP*, 2017.

[48] Y. Zheng, X. Yuan, X. Wang, J. Jiang, C. Wang, and X. Gui, "Toward Encrypted Cloud Media Center With Secure Deduplication," *IEEE Trans. on Multimedia*, vol. 19, no. 2, pp. 251–265, 2017.

[49] J. Liu, N. Asokan, and B. Pinkas, "Secure Deduplication of Encrypted Data without Additional Independent Servers," in *Proc. of ACM CCS*, 2015.

[50] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications," in *Proc. of ACM HotNets*, 2015.

[51] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *Proc. of ACM ISCA*, 2017.

[52] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless Calls Made Practical in Intel SGX," in *Proc. of ACM SysTEX*, 2018.

[53] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren, "LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed," *arXiv preprint arXiv:1706.06261*, 2017.

[54] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proc. of ACM ASPLOS*, 2018.