VC3: Trustworthy Data Analytics in the Cloud using SGX

Felix Schuster*, Manuel Costa, Cédric Fournet, Christos Gkantsidis
Marcus Peinado, Gloria Mainar-Ruiz, Mark Russinovich
Microsoft Research

Abstract—We present *VC3*, the first system that allows users to run distributed MapReduce computations in the cloud while keeping their code and data secret, and ensuring the correctness and completeness of their results. *VC3* runs on unmodified Hadoop, but crucially keeps Hadoop, the operating system and the hypervisor out of the TCB; thus, confidentiality and integrity are preserved even if these large components are compromised. *VC3* relies on SGX processors to isolate memory regions on individual computers, and to deploy new protocols that secure distributed MapReduce computations. *VC3* optionally enforces *region self-integrity* invariants for all MapReduce code running within isolated regions, to prevent attacks due to unsafe memory reads and writes. Experimental results on common benchmarks show that *VC3* performs well compared with unprotected Hadoop: *VC3*'s average runtime overhead is negligible for its base security guarantees, 4.5% with write integrity and 8% with read/write integrity.

I. INTRODUCTION

Cloud providers provision thousands of computers into data centers and make them available on demand. Users rent this computing capacity to run large-scale distributed computations based on frameworks such as MapReduce [4], [20]. This is a cost-effective and flexible arrangement, but it requires users to trust the cloud provider with their code and data: while data at rest can easily be protected using bulk encryption [35], at some point, cloud computers typically need access to the users' code and data in plaintext in order to process them effectively. Of special concern is the fact that a single malicious insider with administrator privileges in the cloud provider's organization may leak or manipulate sensitive user data. In addition, external attackers may attempt to access this data, e. g., by exploiting vulnerabilities in an operating system or even a hypervisor deployed in the cloud infrastructure. Finally, attackers may also tamper with users' computations to make them produce incorrect results. Typically, cloud users hope for the following security guarantees:

I Confidentiality and integrity for both code and data; i. e., the guarantee that they are not changed by attackers and that they remain secret.

II Verifiability of execution of the code over the data; i. e., the guarantee that their distributed computation globally ran to completion and was not tampered with.

In theory, multiparty computation techniques may address these demands. For instance, data confidentiality can be achieved using *fully homomorphic encryption* (FHE), which enables cloud processing to be carried out on encrypted

data [22]. However, FHE is not efficient for most computations [23], [65]. The computation can also be shared between independent parties while guaranteeing confidentiality for individual inputs (using e. g., garbled circuits [29]) and providing protection against corrupted parties (see e. g., SPDZ [19]). In some cases, one of the parties may have access to the data in the clear, while the others only have to verify the result, using zero-knowledge proofs (see e. g., Pinocchio [48], Pantry [13], and ZQL [21]). Still, our goals cannot currently be achieved for distributed general-purpose computations using these techniques without losing (orders of magnitude of) performance. Other systems use specific types of computation and provide practical guarantees, but do not protect all code and data (see e. g., CryptDB [50] and Cipherbase [6]).

We present *Verifiable Confidential Cloud Computing* (*VC3*), a MapReduce framework that achieves the security guarantees (I and II) formulated above, with good performance. Our threat model accounts for powerful adversaries that may control the whole cloud provider's software and hardware infrastructure, except for the certified physical processors involved in the computation. Denial-of-service, side-channels, and traffic-analysis attacks are outside the scope of this work.

Our main contribution is the design, implementation, and evaluation of a practical system that integrates hardware primitives, cryptographic protocols, and compilation techniques. We use trusted SGX processors [3], [27], [41] as a building block, but we need to solve several challenges not directly addressed by the hardware. The first is to partition the system into trusted and untrusted parts, to minimize its TCB. *VC3* runs on unmodified Hadoop, but our design crucially keeps Hadoop, the operating system and the hypervisor out of the TCB. Thus, our confidentiality and integrity guarantees hold even if these large software components are compromised. To keep the TCB small in our design, users simply write the usual *map* and *reduce* functions in C++, encrypt them, bind them to a small amount of code that implements our cryptographic protocols, and finally upload the code to the cloud. On each worker node, the cloud operating system loads the code into a secure region within the address space of a process and makes use of the security mechanisms of SGX processors to make the region inaccessible to the operating system and the hypervisor. Subsequently, the code inside the region runs our key exchange protocol, decrypts the *map* and *reduce* functions, and runs the distributed computation that processes the data. By comparison, recent work [9] proposes loading a library variant of Windows 8 together with an application into an SGX-isolated region; this allows running unmodified Windows

IEEE computer society

binaries, but results in a TCB that is larger than *VC3*'s by several orders of magnitude.

The second challenge is to guarantee integrity for the whole distributed computation, since the processors guarantee only integrity of memory regions on individual computers. We thus propose an efficient job execution protocol that guarantees the correct and confidential execution of distributed MapReduce jobs: the computing nodes produce secure summaries of the work they perform, and they aggregate the summaries they receive from their peers. By verifying the final summaries included in the results, the user can check that the cloud provider did not interfere with the computation. At the same time, the cloud provider can freely schedule and balance the computation between the nodes, as long as all data is eventually processed correctly.

The final challenge is to protect the code running in the isolated memory regions from attacks due to unsafe memory accesses. SGX processors allow this code to access the entire address space of its host process, thus unsafe memory accesses can easily leak data or enable other attacks. Since implementing full memory safety for C/C++ [43], [44], [60] is expensive, we instead provide a compiler that efficiently enforces two *region self-integrity* invariants for code in an isolated region: *region-write-integrity* which guarantees that writes through pointers write only to address-taken variables or heap allocations in the isolated region, and that indirect call instructions target only address-taken functions in the region; and *region-read-write-integrity*, which further guarantees that reads through pointers read only from addresses inside the region. Users who want these additional security assurances may use our compiler.

We implemented *VC3* for the popular Hadoop distribution *HDInsight* on the Windows operating system. Our implementation is based on the new hardware security mechanisms of Intel SGX, but it could in principle target other secure computing technologies [46]. Experimental results on common benchmarks show that *VC3* performs well compared with unprotected Hadoop; *VC3*'s average runtime overhead is negligible for its base security guarantees, 4.5% with write integrity and 8% with read/write integrity.

In summary we make the following contributions:

• We describe the design and implementation of *VC3*, the first system executing MapReduce jobs with good performance while guaranteeing *confidentiality* and *integrity* of *code* and *data*, as well as the correctness and completeness of the results. We propose a partitioning of the system that achieves a small TCB: we keep Hadoop, the operating system and the hypervisor out of the TCB. Our design runs on unmodified Hadoop and works well with Hadoop's scheduling and fault-tolerance services.

• We design and implement two new security protocols, for each MapReduce job, first for cloud attestation and key exchange, then for running the job and gathering evidence of its correct execution. We establish their security by reduction to standard cryptographic assumptions. The security proofs appear in the extended version of this paper [55].
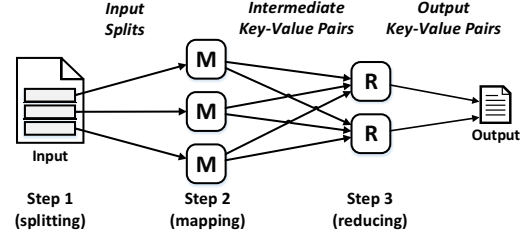

Fig. 1: The steps of a MapReduce job as discussed in this work with mappers (**M**) and reducers (**R**).

• We design and implement efficient, compiler-based, *region-write-integrity* and *region-read-write-integrity* invariants for all user code running within isolated regions.

• We report on the performance of a practical implementation of *VC3* under realistic conditions by running 7 applications on a Hadoop cluster.

We proceed as follows: we provide background (§II), introduce our adversary model (§III), present an overview of our design (§IV), present our cryptographic protocols (§V and §VI), describe our region self-integrity invariants and how to enforce them (§VII), discuss limitations (§VIII), present our implementation (§IX), evaluate our approach (§X), discuss related work (§XI), and conclude (§XII).

II. BACKGROUND

A. MapReduce

MapReduce [20] is a popular programming model for processing large data sets: users write *map* and *reduce* functions, and the execution of both functions is automatically parallelized and distributed.

The abstract data-flow of a parallel MapReduce job is depicted in Figure 1. Each job is a series of three steps: *splitting*, *mapping*, and *reducing*. In the splitting step, the framework breaks raw input data into so called *input splits*. Input splits are then distributed between mappers. Each mapper node parses its splits into *input key-value pairs*, and calls the map function on each of them to produce *intermediate key-value pairs*. The framework groups these pairs by key and distributes them between reducers (*partitioning* and *shuffling*). Each reducer node calls the reduce function on sets of all the values with the same key to produce *output key-value pairs*.

Probably the most popular framework for the execution and deployment of MapReduce jobs is *Hadoop* [4]. Hence, we chose it as our default execution environment.

B. Intel SGX

SGX [3], [32], [41] is a set of x86-64 ISA extensions that makes it possible to set up protected execution environments (called enclaves) without requiring trust in anything but the processor and the code users place inside their enclaves. Enclaves are protected by the processor: the processor controls access to enclave memory. Instructions that attempt to read or write the memory of a running enclave from outside the enclave will fail. Enclave cache lines are encrypted and

integrity protected before being written out to RAM. This removes a broad class of hardware attacks and limits the hardware TCB to only the processor. The software TCB is only the code that users decide to run inside their enclave.

Enclave code can be called from untrusted code by means of a callgate-like mechanism that transfers control to a user-defined entry point inside the enclave. Enclave execution may be interrupted due to interrupts or traps. In such cases, the processor will save the register context to enclave memory and scrub the processor state before resuming execution outside the enclave. Enclaves reside within regular user mode processes. Enclave code can access the entire address space of its host process. This feature allows for efficient interaction between enclave code and the outside world.

SGX supports sealed storage and attestation [3]. While different in many details, these features have the same basic purpose as sealed storage and attestation in other trusted computing hardware. During enclave construction (by untrusted software), the processor computes a digest of the enclave which represents the whole enclave layout and memory contents. This digest is roughly comparable to the PCR values of the TPM [62]. Untrusted software, like the operating system (OS) or the hypervisor, can freely interfere with enclave creation, but such interference will cause the processor to register a different digest for the enclave. The sealing facilities provide each enclave with keys that are unique to the processor and the enclave digest. Local attestation allows an enclave to prove to another enclave that it has a particular digest and is running on the same processor. This privileged mechanism can be used to establish authenticated shared keys between local enclaves. It also enables the deployment of enclaves that support remote attestation. To this end, each SGX processor is provisioned with a unique asymmetric private key that can be accessed only by a special *quoting enclave* (QE) [3]. We refer to this special QE as SGX QE. The SGX QE signs digests of local enclaves together with digests of data produced by them, creating so called *quotes*. A quote proves to a remote verifier that certain information came from a specific enclave running on a genuine SGX processor.

C. Cryptographic Assumptions

We now introduce standard notations and security assumptions for the cryptography we use.

We write $m \mid n$ for the tagged concatenation of two messages m and n. (That is, $m_0 \mid n_0 = m_1 \mid n_1$ implies both $m_0 = m_1$ and $n_0 = n_1$.)

Cryptographic Hash, PRF, and Enclave Digest

We rely on a keyed pseudo-random function, written $\mathsf{PRF}_k(text)$ and a collision-resistant cryptographic hash function, written $\mathsf{H}(text)$. Our implementation uses HMAC and SHA-256.

We write $\mathsf{EDigest}(C)$ for the SGX digest of an enclave's initial content C. We refer to C as the *code identity* of an enclave. Intuitively, $\mathsf{EDigest}$ provides collision resistance; the SGX specification [32] details its construction.

Public-key Cryptography

We use both public-key encryption and remote attestation for key establishment.

A public-key pair pk, sk is generated using an algorithm $\mathsf{PKGen}()$. We write $\mathsf{PKEnc}_{pk}\{text\}$ for the encryption of $text$ under pk. In every session, the user is identified and authenticated by a public-key pk_u. We assume the public-key encryption scheme to be at least *IND-CPA* [10]: without the decryption key, and given the ciphertexts for any chosen plaintexts, it is computationally hard to extract any information from those ciphertexts. Our implementation uses an *IND-CCA2* [10] RSA encryption scheme.

We write $\mathsf{ESig}_P[C]\{text\}$ for a quote from a QE with identity P that jointly signs $\mathsf{H}(text)$ and the $\mathsf{EDigest}(C)$ on behalf of an enclave with code identity C. We assume that this quoting scheme is *unforgeable under chosen message attacks* (UF-CMA). This assumption follows from collision-resistance for H and $\mathsf{EDigest}$ and UF-CMA for the EPID group signature scheme [15]. Furthermore, we assume that Intel's quoting protocol implemented by QEs is secure [3]: only an enclave with code identity C may request a quote of the form $\mathsf{ESig}_P[C]\{text\}$.

Authenticated Encryption

For bulk encryption, we rely on a scheme that provides *authenticated encryption with associated data* (AEAD). We write $\mathsf{Enc}_k(text, ad)$ for the encryption of *text* with associated data *ad*, and $\mathsf{Dec}_k(cipher, ad)$ for the decryption of *cipher* with associated data *ad*. The associated data is authenticated, but not included in the ciphertext. When this data is communicated with the ciphertext, we use an abbreviation, writing $\mathsf{Enc}_k[ad]\{text\}$ for $ad \mid \mathsf{Enc}_k(text, ad)$. (Conversely, any IV or authentication tag used to implement AEAD is implicitly included in the ciphertext.) We assume that our scheme is both *IND-CPA* [11] (explained above) and *INT-CTXT* [11]: without the secret key, and given the ciphertexts for any chosen plaintexts and associated data, it is hard to forge any other pair of ciphertext and associated data accepted by Dec_k. Our implementation uses AES-GCM [40], a high-performance AEAD scheme.

III. ADVERSARY MODEL

We consider a powerful adversary who may control the entire software stack in a cloud provider's infrastructure, including hypervisor and OS. The adversary may also record, replay, and modify network packets. The adversary may also read or modify data after it left the processor using probing, DMA, or similar techniques. Our adversary may in particular access any number of other jobs running on the cloud, thereby accounting for coalitions of users and data center nodes. This captures typical attacks on cloud data centers, e.g., an administrator logging into a machine trying to read user data, or an attacker exploiting a vulnerability in the kernel and trying to access user data in memory, in the network, or on disk.

We assume that the adversary is unable to physically open and manipulate at least those SGX-enabled processor packages that reside in the cloud provider's data centers. Denial-
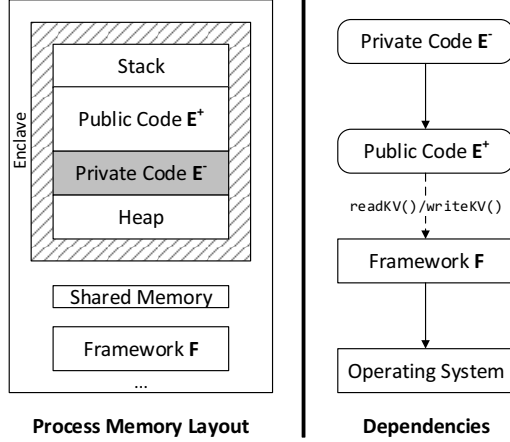
Fig. 2: Left: Memory layout of process containing SGX enclave and framework code. **Right:** Dependencies between the involved components.

Fig. 3: High-level concept of a *VC3* enhanced MapReduce job: code and data are always kept encrypted when outside the processor chip.

of-service, network traffic-analysis, side-channels, and fault injections are also outside the scope of this paper.

We consider the user's implementation of the *map* and *reduce* functions to be benign but not necessarily perfect: the user's code will never intentionally try to leak secrets from enclaves or compromise the security of *VC3* in any other way, but it may contain unintended low-level defects.

IV. DESIGN OVERVIEW

Our goal is to maintain the confidentiality and integrity of distributed computations running on a network of hosts potentially under the control of an adversary. This section outlines our design to achieve this with good performance and keeping large software components out of the TCB.

In *VC3*, users implement MapReduce jobs in the usual way: they write, test, and debug *map* and *reduce* functions using normal C++ development tools. Users may also statically link libraries for particular data analytics domains (e. g., machine learning) with their code; these libraries should contain pure data-processing functions that do not depend on the operating system (we provide mathematical and string libraries in our prototype).

When their *map* and *reduce* functions are ready for production, users compile and encrypt them, obtaining the private enclave code E^-. Then they bind the encrypted code together with a small amount of generic public code E^+ that implements our key exchange and job execution protocols (see §V). Users then upload a binary containing the code to the cloud; they also upload files containing encrypted data. In the cloud, enclaves containing E^- and E^+ are initialized and launched by public and *untrusted framework code F* on worker nodes. Figure 2 depicts the memory layout of a process containing the described components; it also shows their dependencies. In *VC3*, a MapReduce job starts with a key exchange between the user and the public code E^+ running in the secure enclave on each node. After a successful key exchange, E^+ is ready to decrypt the private code E^- and to process encrypted data following the distributed job execution protocol.
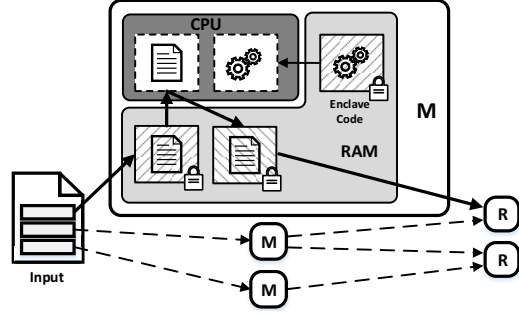
To keep the operating system out of *VC3*'s TCB, we kept the interface between E^+ and the outside world narrow. Conceptually, it has only two functions: `readKV()` and `writeKV()`, for reading and writing key-value pairs from and to Hadoop (akin to receiving and sending messages). Since F and enclave share the virtual address space of a process, data is passed from E^+ inside the enclave to F outside the enclave over a shared memory region outside the enclave. Other than relying on this narrow interface, the code in the enclave is self-sufficient: it has no further dependencies on the operating system. The enclave has its own stack which we reserve on start-up (it includes a guard page to detect stack out-of-memory conditions); it has its own heap, carved out of the enclave memory region; and we guarantee that only one thread at a time executes the user code (the enclave is created with a single thread control structure, to ensure that all execution of enclave code is single threaded); parallelism is achieved in MapReduce jobs by running many instances of the *map* and *reduce* functions in parallel in separate enclaves. With this design, the operating system and the hypervisor can still mount attacks such as not scheduling processes, dropping or duplicating network packets, not performing disk I/O, and corrupting data when it is out of the enclaves. While we cannot guarantee progress if the operating system mounts denial of service attacks, our job execution protocol guarantees that the results are correct and complete if the distributed computation terminates successfully.

Note that while in the cloud, both E^- and the user's data are always kept encrypted, except when physically inside the trusted processor chip on a mapper or reducer node, as shown in Figure 3. Inside the processor chip, the user's *map* and *reduce* functions run on plaintext data at native speed. At the same time, we allow Hadoop to manage the execution of *VC3* jobs. The framework code (F) implements the Hadoop streaming interface [5] to bind to unmodified Hadoop deployments; *VC3*'s map and reduce nodes look like regular worker nodes to Hadoop. Thus, Hadoop can use its normal scheduling and fault-tolerance mechanisms to manage all dataflows, including performance mechanisms for load balancing and straggler mitigation. But the Hadoop framework, the operating system, and the hypervisor are kept out of the TCB.

41

Some of these properties could also be achieved using trusted hypervisors [17], [28], [36], [38], [58], but trusted hypervisors are problematic in a cloud environment. They force a potentially large privileged software component that is under the control of the (possibly adversarial) cloud provider into the TCB of every user. While users can use attestation to authenticate a digest of such software, it is unclear how they can establish trust in it, especially in light of periodic software upgrades. While the software TCB of a *VC3* application *may* have as many or more lines of code as small special-purpose hypervisors [38], [58], the former code is entirely chosen and compiled by the user, whereas the latter are not. It is also unclear how these special-purpose hypervisors could be extended to coexist with the more functional hypervisors used in cloud systems [8]. Finally, note that *VC3*'s hardware TCB is only the SGX processor package; this is smaller than traditional systems based on TXT [31] or a TPM (large parts of the motherboard); this is important in a cloud setting where the hardware is under the control of the cloud provider.

The final aspect of the *VC3* design is that users may enforce *region self-integrity* invariants using our compiler. The region integrity invariants act as an additional layer of protection that allows the trusted code inside the enclave to continuously monitor its internal state to prevent memory corruption and disclosure of information due to low-level defects. Users who want the additional security assurances may use our compiler, but we emphasize that this optional; users may use other mechanisms, including manual inspection, testing, and formal verification, to check that their code does not have defects.

V. JOB DEPLOYMENT

After preparing their code, users deploy it in the cloud. The code is then loaded into enclaves in worker nodes, and it runs our key exchange protocol to get cryptographic keys to decrypt the *map* and *reduce* functions. After this, the worker nodes run our job execution and verification protocol. This section and the next present our cryptographic protocols for the exchange of keys and the actual MapReduce job execution, respectively. Before describing these protocols in detail, we first discuss the concept of *cloud attestation* used in *VC3*.

A. Cloud Attestation

As described above, in SGX, remote attestation for enclaves is achieved via *quotes* issued by QEs. The default SGX QE only certifies that the code is running on *some* genuine SGX processor, but it does not guarantee that the processor is actually located in the cloud provider's data centers. This may be exploited via a type of cuckoo attack [47]: an attacker could, for example, buy *any* SGX processor and conduct a long term physical attack on it to extract the processor's master secret. If no countermeasures were taken, she would then be in a position to impersonate any processor in the provider's data centers. Note that our threat model excludes physical attacks only on the processors inside the data centers.

To defend against such attacks, we use an additional *Cloud*

QE, created by the cloud provider whenever a new SGX-enabled system is provisioned. The purpose of the Cloud QE is to complement quotes by the SGX QE with quotes stating that the enclave runs on hardware owned and certified by the cloud provider, in a certain data center. At the same time, to defend against possibly corrupted cloud providers, we only use the Cloud QE in conjunction with the SGX QE. (Note that the cloud provider cannot fake quotes from the SGX QE, since our threat model excludes physical attacks on the processors inside the data centers.) The procedure to provision Cloud QEs is simple. Before a new machine enters operation in a data center, a Cloud QE is created in it. This Cloud QE then generates a public/private key pair, outputs the public key and seals the private key which never leaves the Cloud QE.

In the following, we assume two fixed signing identities for SGX and for the cloud, we write $\mathsf{ESig}_{SGX}[C]\{text\}$ and $\mathsf{ESig}_{Cloud}[C]\{text\}$ for quotes by the main SGX QE and the Cloud QE, respectively, and write $\mathsf{ESig}_{SGX,Cloud}[C]\{text\}$ for their concatenation $\mathsf{ESig}_{SGX}[C]\{text\} \mid \mathsf{ESig}_{Cloud}[C]\{text\}$.

We foresee that cloud providers will create groups of processors based on geographical, jurisdictional, or other boundaries that are of interest to the user, and will publish the appropriate public keys to access these groups of processors.

B. Key Exchange

To execute the MapReduce job, enclaves first need to get keys to decrypt the code and the data, and to encrypt the results. In this section we describe our protocol for this. Our key exchange protocol is carefully designed such that it can be implemented using a conventional MapReduce job that works well with existing Hadoop installations. We first describe the protocol using generic messages, and then show how to integrate it with Hadoop. We present a multi-user variant in Appendix A and a *lightweight* variant in Appendix B.

Recall that the user is identified and authenticated by her key pk_u for public-key encryption and each SGX processor runs a pair of SGX and Cloud QEs. Before running the protocol itself, the user negotiates with the cloud provider an allocation of worker nodes for running a series of jobs.

Setting up a new job involves three messages between the user and each node:

1) The user chooses a fresh job identifier j and generates a fresh symmetric key k_{code} to encrypt E^-, then sends to any node involved the code for its *job enclave* ($C_{j,u}$):

$$C_{j,u} = E^+ \mid \mathsf{Enc}_{k_{code}}[]\{E^-\} \mid j \mid pk_u.$$

2) Each node w starts an enclave with code identity $C_{j,u}$. Within the enclave E^+ derives a symmetric key k_w [1] and encrypts it under the user's public key:

$$m_w = \mathsf{PKEnc}_{pk_u}\{k_w\}.$$

---

[1]This can be the enclave's *sealing key* or a key generated using the random output of the x86-64 instruction RDRAND.

The enclave then requests quotes from the SGX and Cloud QEs with text m_w, thereby linking m_w to its code identity $C_{j,u}$ (and thus also to the job-specific j and pk_u). The message m_w and the quotes are sent back to the user:

$$p_w = m_w \mid \mathsf{ESig}_{SGX,Cloud}[C_{j,u}]\{m_w\}.$$

3) The user processes a message p_w from each node w, as follows: the user verifies that both quotes sign the message payload m_w with the code identity $C_{j,u}$ sent in the initial message; then, the user decrypts m_w and responds with *job credentials* encrypted under the resulting node key k_w:

$$JC_w = \mathsf{Enc}_{k_w}[]\{k_{code} \mid \mathbf{k}\}$$

where k_{code} is the key that protects the code E^- and

$$\mathbf{k} = k_{job} \mid k_{in} \mid k_{inter} \mid k_{out} \mid k_{prf}$$

is the set of authenticated-encryption keys used in the actual job execution protocol (see §VI). Specifically, k_{job} is used to protect verification messages while k_{in}, k_{inter}, and k_{out} are used to protect input splits, intermediate key-value pairs, and output key-value pairs respectively; k_{prf} is used for keying the pseudo-random function PRF.

4) Each node resumes E^+ within the job enclave, which decrypts the job credentials JC_w using k_w, decrypts its private code segment E^- using k_{code}, and runs E^-.

On completion of the protocol, the user knows that any enclave that contributes to the job runs the correct code, and that she shares the keys for the job with (at most) those enclaves.

Our protocol provides a coarse form of forward secrecy, inasmuch as neither the user nor the nodes need to maintain long-term private keys. (The user may generate a fresh pk_u in every session.) The protocol can also easily be adapted to implement a Diffie-Hellmann key agreement, but this would complicate the integration with Hadoop described in §V-C.

An outline of the security theorem for the key exchange is given below; the formal theorem statement, auxiliary definitions, and proof appear in the extended version of this paper [55].

Theorem 1. *Enclave and Job Attestation (Informally)*

1) *If a node completes the exchange with user public key pk_u and job identifier j, then the user completed the protocol with those parameters; and*
2) *all parties that complete the protocol with (pk_u, j) share the same job code E^+, E^- and job keys in $\mathbf{k}$; and*
3) *the adversary learns only the encrypted size of E^-, and nothing about the job keys in $\mathbf{k}$.*

C. Integrating Key Exchange with Hadoop

Hadoop does not foresee online connections between nodes and the user, hence we need another mechanism to implement the key exchange in practice. We now describe the *in-band* variant of key exchange that is compatible with unmodified Hadoop installations and is implemented in our *VC3* prototype.

The *in-band* variant of key exchange is designed as a lightweight *key exchange job* that is executed before the *actual job*. The online communication channels between nodes and user are replaced by the existing communication channels in a MapReduce job: $user \to mapper \to reducer \to user$. By design, our in-band key exchange also does not require nodes to locally maintain state between invocations. (Per default, Hadoop does not foresee applications to store files permanently on nodes.) This is achieved by diverting the enclaves' unique and secret *sealing keys* from their common use. The exact procedure is described in the following.

The user creates $C_{j,u}$ and the accompanying parameters for the *actual job* as described. The user then deploys this exact $C_{j,u}$ first for the special *key exchange job*. It is important that the same $C_{j,u}$ is executed on the same nodes for both jobs.

When launched on a mapper or reducer node, E^+ obtains the enclave's unique *sealing key* (unique to the processor and digest of $C_{j,u}$, see §II-B) and uses it as its node key k_w. Each node outputs the corresponding p_w in the form of a MapReduce key-value pair. Mapper nodes immediately terminate themselves subsequently, while reducer nodes remain active until having forwarded all intermediate key-value pairs containing the mappers' p_w. E^- is not (and cannot be) decrypted during the *key exchange job*. The user obtains all p_w from the final outputs of the *key exchange job*. The user creates the job credentials JC_w for each node as described. Finally, the user writes JC_w for all nodes to a file and deploys it together with $C_{j,u}$ for the *actual job*.

During the *actual job*, E^+ derives the unique sealing key (equivalent to k_w) on each node again and uses it to decrypt the corresponding entry in D, obtaining k_{code} and $\mathbf{k}$. Afterward, E^- is decrypted and the execution of the job proceeds as normal. Note how it is essential to use the exact same $C_{j,u}$ in both jobs. Otherwise, the sealing keys used in the *key exchange job* could not be re-obtained during the execution of the *actual job*. Thus, E^+ needs to implement the required functionality for both jobs.

VI. JOB EXECUTION AND VERIFICATION

After obtaining keys to decrypt the secret code and data, worker nodes need to run the distributed MapReduce computation. A naïve approach for protecting the computation would be to simply encrypt and authenticate all the key-value pairs exchanged between the nodes. A hostile cloud environment would though still be in the position to arbitrarily drop or duplicate data. This would allow for the manipulation of outputs. A dishonest cloud provider might also simply be tempted to drop data in order to reduce the complexity of jobs and thus to save on resources. Furthermore, care has to be taken when encrypting data in a MapReduce job in order not to negatively impact the load-balancing and scheduling capabilities of Hadoop or the correctness of results. In this section we present our protocol that tackles these problems and guarantees the overall integrity of a job and the confidentiality of data. As before, we first describe the protocol using generic messages, and then show how to integrate it with Hadoop.
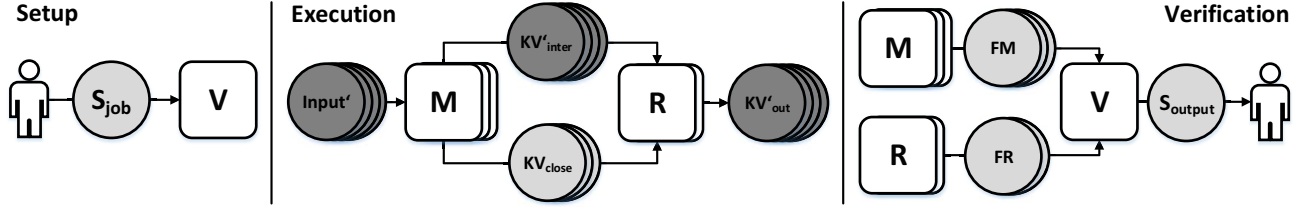
Fig. 4: Schematic overview of our job execution protocol. The verifier (**V**), mappers (**M**), and reducers (**R**) are depicted as squares. A light-gray circle displays a message/key-value pair that is sent *once* by an entity; a dark-gray circle one that is sent multiple times. The user is depicted at both far ends.

For now, we rely on a not further specified *verifier* that can communicate securely with the user and is trusted by the user. In practice, the verifier can run on a user's local machine or in an enclave. We show later how the verifier can also be implemented "in-band" as a distinct MapReduce job. Our implementation uses a distinct tag for each type of message; these tags are omitted below for simplicity. The entire protocol is implemented in E^+. Figure 4 gives a schematic overview of the message flows in the protocol.

Step 1: Setup

As a preliminary step, the user uploads chunks of AEAD-encrypted data as *input splits* to the cloud provider. Each encrypted input split *Input* is cryptographically bound to a fresh, unique identifier (ID) ℓ_{in}:

$$Input' = \mathsf{Enc}_{k_{in}}[\ell_{in}]\{Input\}$$

(In practice, we use the 128-bit MAC of the AES-GCM encryption as ID. Book keeping can though be simpler for incremental IDs.) All encrypted input splits *Input'* are stored by the cloud provider. The user decides on a subset of all available input splits as input for the job: $B_{in} = \{\ell_{in,0}, \ell_{in,1}, \ldots, \ell_{in,n-1}\}$; chooses a number of logical reducers for the job: R; and passes the job specification $S_{job} = j \mid k_{job} \mid R \mid B_{in}$ securely to the verifier. The number of mapper instances is not fixed *a priori* as Hadoop dynamically creates and terminates mappers while executing a job. We write $m \in \mathbf{m}$ for the mapper indexes used for the job. (This set of indexes is *a priori* untrusted; one goal of the protocol is to ensure that all reducers agree on it.)

Step 2: Mapping

Hadoop distributes input splits to running mapper instances. As input splits are encrypted, Hadoop cannot parse them for key-value pairs. Hence, the parsing of input splits is undertaken by E^+. Mappers keep track of the IDs of the input splits they process, and they refuse to process any input split more than once.

Intermediate Key-Value Pairs

Mappers produce *intermediate* key-value pairs $KV_{inter} = \langle K_{inter} : V_{inter}\rangle$ from the input splits they receive. Hadoop assigns these to reducers for final processing according to each pair's key (the *shuffling* step). For the functional correctness of a job, it is essential that key-value pairs with identical keys are processed by the same reducer; otherwise the job's final output could be fragmented. However, the user typically has a strong interest in keeping not only the value V_{inter} but also the key K_{inter} of an intermediate key-value pair

secret. Thus, our mappers wrap plaintext intermediate key-value pairs in encrypted intermediate key-value pairs KV'_{inter} of the following form:

$$K'_{inter} = r \equiv \mathsf{PRF}_{k_{prf}}(K_{inter}) \bmod R$$
$$V'_{inter} = \mathsf{Enc}_{k_{inter}}[j \mid \ell_m \mid r \mid i_{m,r}]\{\langle K_{inter} : V_{inter}\rangle\}$$
$$KV'_{inter} = \langle K'_{inter} : V'_{inter}\rangle$$

By construction, we have $K'_{inter} \in 0..R-1$, and all intermediate key-value pairs KV with the same key are assigned to the same logical reducer. The derivation of K'_{inter} is similar to the standard *partitioning* step performed by Hadoop [4].

In the associated authenticated data above, ℓ_m is a secure unique job-specific ID randomly chosen by the mapper $m \in \mathbf{m}$ for itself (in our implementation we use the x86-64 instruction `RDRAND` inside the enclave); r is the reducer index for the key; and $i_{m,r}$ is the number of key-value pairs sent from this mapper to this reducer so far. Thus, $(\ell_m, r, i_{m,r})$ uniquely identifies each intermediate key-value pair within a job. Note that, in practice, many plaintext KV_{inter} from one mapper to one reducer may be batched into a single KV'_{inter}.

Mapper Verification

For verification purposes, after having processed all their inputs, our mappers also produce a special *closing intermediate key-value pair* for each $r \in R$:

$$KV_{close} = \langle r : \mathsf{Enc}_{k_{inter}}[j \mid \ell_m \mid r \mid i_{m,r}]\{\}\rangle$$

This authenticated message ensures that each reducer knows the total number $i_{m,r}$ of intermediate key-value pairs (zero or more) to expect from each mapper. In case a reducer does not receive exactly this number of key-value pairs, or receives duplicate key-value pairs, it terminates itself without outputting its final verification message (see next step).

Furthermore, each mapper sends the following final verification message to the verifier:

$$FM = \mathsf{Enc}_{k_{job}}[j \mid \ell_m \mid B_{in,m}]\{\}$$

where $B_{in,m}$ is the set of IDs of all input splits the mapper $m \in \mathbf{m}$ processed. This authenticated message lets the verifier aggregate information about the distribution of input splits.

Step 3: Reducing

Assuming for now that Hadoop correctly distributes all intermediate key-value pairs KV'_{inter} and KV_{close}, reducers produce and encrypt the final *output key-value pairs* for the job:

$$KV'_{out} = \langle \ell_{out} : \mathsf{Enc}_{k_{out}}[\ell_{out}]\{KV_{out}\}\rangle$$

where KV_{out} is a split of final output key-value pairs, with secure unique ID ℓ_{out}. (Again, we may use the MAC of the AES-GCM encryption as unique ID.) By design, the format of V'_{out} is compatible with the format of encrypted input splits, allowing the outputs of a job to be immediate inputs to a subsequent one.

Reducer Verification

After having successfully processed and verified all key-value pairs KV'_{inter} and KV_{close} received from mappers, each reducer sends a final verification message to the verifier:

$$FR = j \mid r \mid B_{out,r} \mid \mathsf{Enc}_k(j \mid r \mid B_{out,r} \mid P_r, \{\})$$
$$P_r \subseteq (\ell_m)_{m \in \mathbf{m}}$$

The authenticated message *FR* carries the set $B_{out,r}$ of IDs ℓ_{out} for all outputs produced by the reducer with index $r \in R$. It also authenticates a sorted list P_r of mapper IDs, one for each closing intermediate key-value paper it has received. (To save bandwidth, P_r is authenticated in *FR* but not transmitted.)

Step 4: Verification

The verifier receives a set of *FM* messages from mappers and a set of *FR* messages from reducers. To verify the global integrity of the job, the verifier first checks that it received exactly one *FR* for every $r \in 0..R-1$.

The verifier collects and sorts the mapper IDs $P_{verifier} \subseteq (\ell_m)_{m \in \mathbf{m}}$ from all received *FM* messages, and it checks that $P_{verifier} = P_r$ for all received *FR* messages, thereby ensuring that all reducers agree with $P_{verifier}$.

The verifier checks that the sets $B_{in,m}$ received from the mappers form a partition of the input split IDs of the job specification, thereby guaranteeing that every input split has been processed once.

Finally, the verifier accepts the union of the sets received from the reducers, $B_{out} = \bigcup_{r \in 0..R-1} B_{out,r}$, as the IDs of the encrypted job output. The user may download and decrypt this output, and may also use B_{out} in turn as the input specification for another job (setting the new k_{in} to the previous k_{out}).

A. Security Discussion

We outline below our security theorem for the job execution and subsequently discuss the protocol informally; the formal theorem statement, auxiliary definitions, and proof appear in the extended version of this paper [55].

Theorem 2. *Job Execution (Informally)*

1) *If the verifier completes with a set of output IDs, then the decryptions of key-value pairs with these IDs (if they succeed) yield the correct and complete job output.*

2) *Code and data remains secret up to traffic analysis: The adversary learns at most (i) encrypted sizes for code, input splits, intermediate key-value pairs, and output key-value pairs; and (ii) key-repetition patterns in intermediate key-value pairs.*

We observe that, if the verifier completes with a set of output IDs, then the decryptions of key-value pairs with these IDs (if they succeed) yield the correct and complete job output.

For each cryptographic data key, AEAD encryption guarantees the integrity of all messages exchanged by the job execution protocol; it also guarantees that any tampering or truncation of input splits will be detected.

Each message between mappers, reducers, and verifier (KV'_{inter}, KV_{close}, *FM*, and *FR*) includes the job-specific ID j, so any message replay between different jobs is also excluded. Thus, the adversary may at most attempt to duplicate or drop some messages within the same job. Any such attempt is eventually detected as well: if the verifier does not receive the complete set of messages it expects, verification fails; otherwise, given the *FM* messages from the set $\mathbf{m}'$ of mappers, it can verify that the mappers with distinct IDs $(\ell_m)_{m \in \mathbf{m}'}$ together processed the correct input splits. Otherwise, if any inputs splits are missing, verification fails. Furthermore, given one *FR* message for each $r \in 0..R-1$, the verifier can verify that every reducer communicated with every mapper. Given R, the verifier can also trivially verify that it communicated with all reducers that contributed to the output.

Reducers do not know which mappers are supposed to send them key-pairs. Reducers though know from the KV_{close} messages how many key-value pairs to expect from mappers they know of. Accordingly, every reducer is able to locally verify the integrity of all its communication with every mapper. Although the adversary can remove or replicate entire streams of mapper/reducer communications without being detected by the reducer, this would lead to an incomplete set P_r of mapper IDs at the reducer, eventually detected by the verifier.

B. Analysis of Verification Cost

We now analyze the cost for the *verification* of a job with M mappers and R reducers. *VC3*'s full runtime cost is experimentally assessed in §X.

There are $M + R$ verification messages that mappers and reducers send to the verifier. These messages most significantly contain for each mapper the set $B_{in,m}$ of processed input split IDs and for each reducers the set $B_{out,r}$ of IDs of produced outputs. Each ID has a size of 128 bits. Typically, input splits have a size of 64 MB or larger in practice. Hence, mappers need to *securely* transport only 16 bytes to the verifier per 64+ MB of input. As reducers should batch many output key-value pairs into one KV'_{out}, a similarly small overhead is possible for reducer/verifier communication. There are $M \times R$ verification messages sent from mappers to reducers. These messages are small: they contain only four integers. The computational cost of *verification* amounts to the creation and verification of the MACs for all $M + R + M \times R$ verification messages. Additionally, book keeping has to be done (by all entities). We consider the cost for *verification* to be small.

C. Integrating the Verifier with Hadoop

For the job execution protocol it is again desirable to avoid online connections between the involved entities. We now describe a variant of the protocol that implements an *in-band* verifier as a simple MapReduce job. Our *VC3* prototype implements this variant of the job execution protocol.

Mappers send *FM* messages in the form of key-value pairs to reducers. Reducers output all *FM* key-value pairs received from mappers and also output their own *FR* messages in the form of key-value pairs. The *verification job* is given S_{job} of the *actual job* and is invoked on the entire corresponding outputs. The mappers of the *verification job* parse input splits for *FM* and *FR* messages and forward them to exactly one *verification reducer* by wrapping them into key-value pairs with a predefined key K'_{inter}. On success, the *verification reducer* outputs exactly one key-value pair certifying B_{out} as valid output for S_{job}. This key-value pair can finally easily be verified by the user. In practice, the verification job can be bundled with a regular job that already processes the outputs to be verified while parsing for verification messages. In such a case, one of the regular reducers also acts as *verification reducer* (we use the reducer with $r = 0$). The bundled job in turn creates its own verification messages *FM* and *FR*. This way, it is possible to chain an arbitrary number of secure MapReduce jobs, each verifying the integrity of its immediate successor with low overhead.

VII. REGION SELF-INTEGRITY

The final aspect of our design is the enforcement of region self-integrity invariants for user code loaded into enclaves. By design, code within an enclave can access the entire address space of its host process. This enables the implementation of efficient communication channels with the outside world but also broadens the attack surface of enclaves: if enclave code, due to a programming error, ever dereferences a corrupted pointer to untrusted memory outside the enclave, compromise of different forms becomes possible. For example, the enclave code may write through an uninitialized pointer or a *null* pointer; if the pointer happens to contain an address that is outside of the enclave, data immediately leaks out. Conversely, reads through such pointers may cause the user code to read data from arbitrary addresses outside the enclave; in this case, the untrusted environment is in the position to inject arbitrary data into the enclave. Such a data injection may in the simplest case affect the correctness of computations but may also, depending on the context of the corrupted pointer dereference, pave the way for a control-flow hijacking attack eventually allowing the adversary to capture all the enclave's secrets. We stress that we assume the code inside the enclave is not malicious, but it may have low-level defects; applications written in languages like C and C++ have a long history of problems induced by unsafe memory accesses.

Since memory safety implementations for C/C++ have high overhead [43], [44], [60], we instead address this problem with a compiler that efficiently enforces two security invariants for code running inside the enclave. Before presenting the invariants, we introduce some terminology. An *address-taken variable* is a variable whose address is taken in the code, e. g. &v, or an array (the address of arrays is implicitly taken). By *write through a pointer* we mean writing to the memory targeted by the pointer (this includes array accesses, which use a pointer and an offset). Likewise for *read through a pointer*. We define two invariants:

Region-write-integrity guarantees that writes through pointers write only to address-taken variables in the enclave or to allocations from the enclave heap. Additionally, it guarantees that indirect call instructions can target only the start of address-taken functions in the enclave.

Region-read-write-integrity includes the region-write-integrity guarantee, plus the guarantee that reads through pointers read only from addresses inside the enclave.

Region-write-integrity prevents memory corruption: it prevents corruption of all non-address-taken variables in the program (typically a large fraction of the stack frames contain only non-address-taken variables [34]) and it prevents corruption of all compiler-generated data such as return addresses on the stack. It also prevents information leaks caused by writes to outside of the enclave. Region-read-write-integrity additionally prevents use of un-authenticated data from outside the enclave, which may be injected by an attacker.

The integrity invariants are enforced with dynamic checks on memory reads, writes and control-flow transitions. The compiler inserts dynamic checks when it cannot verify the safety of memory operations or control-flow transitions at compile time. Note that direct writes/reads to local or global variables access memory at fixed offsets in the stack-frame or from the enclave base address are guaranteed to neither corrupt memory, nor access memory outside the enclave (we reserve space for global variables when creating the enclave; and we stop the program if we exhaust stack space). Hence, we only need to check memory accesses through pointers. Checking memory reads adds runtime overhead. We therefore let users choose between no integrity, region-write-integrity, and full region-read-write-integrity, depending on the runtime cost they are willing to pay.

The checks on indirect calls together with the integrity of return addresses enforce a form of control-flow integrity (CFI) [1], but our invariants are stronger than CFI. Attacks on CFI [24] typically require an initial step to corrupt memory (e. g., through a buffer overflow) and/or leak information. CFI solutions do not try to prevent memory corruption; they aim only to mitigate the malicious effects of such corruption by restricting the sets of possible targets for indirect control-flow transitions. On the other hand, our read and write checks are mainly designed to prevent memory corruptions and information leaks; the main purpose of our control-flow checks is to guarantee that the checks on writes and reads cannot be bypassed: control never flows to unaligned code potentially containing unexpected and unchecked memory reads or writes.

Our invariants share some properties with recent proposals for efficient execution integrity, such as CPI [34] and WIT [2], but our invariants and enforcement mechanisms are adapted to the enclave environment. For example, on x64 (*VC3*'s target environment), CPI relies on hiding enforcement information at a random address (with leak-proof information hiding); while that is effective in large address-spaces, it would be less effective inside *VC3*'s small (512MB) memory regions.

46

Our enforcement data structures are also a factor of 8 smaller than WIT's, and unlike CPI and WIT we do not require sophisticated compiler analysis (e. g., points-to analysis).

The instrumentation for memory accesses is applied for all enclave code except for the functions that implement communication with the outside world through the shared memory area, but these functions encrypt/decrypt and authenticate the data being written/read. Next, we describe how we enforce the integrity invariants.

A. Enforcing region-write-integrity

To enforce that writes through pointers go to address-taken variables in the enclave or memory allocated from the enclave heap, we maintain a bitmap to record which memory areas inside the enclave are writable. The bitmap maps every 8-byte slot of enclave memory to one bit. When the bit is set, the memory is writable. The bitmap is updated at runtime, when stack frames with address-taken variables are created and destroyed and when heap memory is allocated and freed. When the compiler determines that a stack frame contains address-taken variables, it generates code to set the corresponding bits on the bitmap on function entry, and to reset them on function exit. The compiler also ensures that address-taken variables have free 8-byte slots around them (and similarly for heap allocations), to detect sequential overflows. The compiler also records the addresses and sizes of address-taken global variables in a data structure that our runtime uses to set the corresponding bits in the bitmap at enclave startup. Our heap implementation sets/resets the bits in the bitmap on heap allocations/deallocations. When the compiler cannot prove statically that a write conforms to region-write-integrity, it inserts a check of the form (*VC3* works on x64 processors):

```
        mov         rax,r8
        and         rax,0xFFFFFFFFE0000000
        xor         rax,0x20000000
        je          $L1
        int         3
$L1:mov         rdx,_writeBitmap
        mov         rcx,r8
        shr         rcx,9
        mov         rax,r8
        shr         rax,3
        mov         rcx,[rdx+rcx*8]
        bt          rcx,rax
        jb          $L2
        int         3
$L2:mov         [r8],4  #unsafe write
```

The first part of the check, up to the L1 label, checks that the address being written to is within the enclave address range. If the check fails, the program stops with an exception; we chose this exception because it uses an efficient encoding: a single byte. If the check succeeds, we then check that the address is marked as writable in the bitmap. The initial range check on the address allows us to allocate the bitmap to cover only a small portion of the address space. If the bitmap check also succeeds, the write is allowed to proceed (label L2).

This design is efficient: the bitmap is a compact representation of which addresses are writable: one bit per 8

bytes of enclave address space and, as shown above, we can access it with fast code sequences. The compiler also includes optimizations to make write checks more efficient (§IX).

To implement the checks on indirect control-flow transitions, we maintain a separate bitmap that records where the entry points of address-taken functions are. This bitmap maps each 16-byte slot of enclave memory to a bit. The bit is set if an address-taken function starts at the beginning of the slot. The compiler aligns address-taken functions on 16-byte boundaries, and records the addresses of these function in a data structure that our runtime uses to set the corresponding bits in the bitmap at enclave startup. Using a 16-byte slot keeps the bitmap small and wastes little space due to alignment; we use smaller slots for the write bitmap to reduce the amount of free space around address-taken variables and heap allocations. When generating code for an indirect control-flow transfer, the compiler emits code to check that the target is 16-byte aligned and that the corresponding bit is set in the bitmap (the code sequence is similar to the write checks).

Note that code outside the enclave cannot corrupt the bitmaps used for the integrity checks, since the bitmaps are allocated inside the enclave. Even writes inside the enclave cannot corrupt the bitmaps, because they are always instrumented and the write bitmap disallows writes to the bitmaps.

B. Enforcing region-read-write-integrity

To enforce region-read-write-integrity, the compiler further emits checks of the form:

```
        mov         rax,r8
        and         rax,0xFFFFFFFFE0000000
        xor         rax,0x20000000
        je          $L1
        int         3
$L1:mov         rdx,[r8]  #unsafe read
```

These checks guarantee that the memory being read is within the enclave region. If it is not, the program stops. An alternative design would be to simply mask the bits in the address to make sure they are within the enclave, without stopping if they are not [66]. While that is more efficient, it is safer to stop the program when the error is detected. Again, when memory accesses are guaranteed to not violate region-read-write-integrity, for example direct accesses to scalar variables on the enclave stack, the compiler elides the read checks at compile time.

VIII. DISCUSSION

We now discuss several attack scenarios on *VC3* which are partly outside the adversary model from §III.

A. Information Leakage

One basic principle of MapReduce is that all key-value pairs with the same key be processed by the same reducer. Thus, inasmuch as a network attacker can count the number of pairs delivered to each reducer, we should not expect semantic security for the intermediate keys (as in "key-value pair") as soon as there is more than one reducer. Next, we discuss this

information leakage in more detail: For the whole job, each key K_{inter} is mapped to a fixed, uniformly-sampled value $K'_{inter} \in 0..R - 1$. where R is the number of reducers for the job chosen by the user (§VI). For each intermediate key-value pair, the adversary may observe the mapper, the reducer, and K'_{inter}. Intuitively, the smaller the overall number of unique intermediate keys K_{inter} in relation to R, the more the adversary may learn on the actual distribution of keys. For example, in the case of a presidential election vote count, there are only two possible intermediate keys (the names of both candidates). If $R > 1$, then the adversary easily learns the distribution of the votes but not necessarily the name of the successful candidate. Conversely, if there are many keys (each with a small number of key-value pairs) relative to R, then leaking the total number of pairs dispatched to each reducer leaks relatively little information. In particular, when all intermediate keys are unique, no information is leaked. Attackers may also use more advanced traffic analyses against *VC3* [16], [56], [68]. For example, by observing traffic, an attacker may correlate intermediate key-value pairs and output key-value pairs to input splits; over many runs of different jobs this may reveal substantial information about the input splits. We plan to address these attacks with padding, clustering, and distributed shuffle techniques [45].

B. Replay Attacks

The adversary could try to profit in various ways from fully or partially replaying a past MapReduce job. Such replay attacks are generally prevented in case the *online* key exchange (§V-B) is employed, as the user can simply refuse to give JC_w a second time to any enclave. This is different for the *in-band* version of our approach (§V-C): an enclave is not able to tell if it ran on a set of input data before as it cannot securely keep state between two invocations. (The adversary can always revert a sealed file and reset the system clock.) Given $C_{j,u}$ and JC_w corresponding to a certain processor under their control, the adversary is in the position to arbitrarily replay parts of a job that the processor participated in before or even invoke a new job on any input splits encrypted under k_{in} contained in JC_w. This allows the adversary to repeatedly examine the runtime behavior of E^- from outside the enclave and thus to amplify other side-channel attacks against *confidentiality*. The resilience of *VC3* against such attacks can be enhanced by hardcoding a job's specification into mappers to restrict the input splits they should accept to process. Finally, Strackx et al. recently proposed an extension to SGX that provides *state continuity* for enclaves [57] and, if adopted, could be used in *VC3* to largely prevent replay attacks.

IX. IMPLEMENTATION

We implemented *VC3* in C++ for Windows 64-bit and the HDInsight distribution of Hadoop. Jobs are deployed as 64-bit native code in the form of an executable (*fw.exe*) which contains the framework code F, and a dynamic link library (*mapred.dll*) that contains the enclave code E^+ and E^-.

A. SGX Emulation

We successfully tested our implementation in an SGX emulator provided by Intel. However since that emulator is not performance accurate, we have implemented our own software emulator for SGX. Our goal was to use SGX as specified in [32] as a concrete basis for our *VC3* implementation and to obtain realistic estimates for how SGX would impact the performance of *VC3*. Our software emulator does not attempt to provide security guarantees.

The emulator is implemented as a Windows driver. It hooks the `KiDebugRoutine` function pointer in the Windows kernel that is invoked on every exception received by the kernel. Execution of an SGX opcode from [32] will generate an illegal instruction exception on existing processors, upon which the kernel will invoke our emulator via a call to `KiDebugRoutine`. The emulator contains handler functions for all SGX instructions used by *VC3*, including EENTER, EEXIT, EGETKEY, EREPORT, ECREATE, EADD, EEX-TEND, and EINIT. We use the same mechanism to handle accesses to model specific registers (MSR) and control registers as specified in [32]. We also modified the `SwapContext` function in the Windows kernel to ensure that the full register context is loaded correctly during enclave execution.

The code in each handler function is modeled after the corresponding pseudo code in [32]. We emulate the *enclave page cache (EPC)* by allocating a contiguous range of physical memory (using the memory manager function `MmAllocateContiguousMemorySpecifyCache`) and using a data structure along the lines of the *Enclave Page Cache Map* of [32] to keep track of it.

B. Performance Model

We are interested in estimating the performance of *VC3* on a hypothetical SGX-enabled processor. We assume that the performance of the existing processor instructions and mechanisms would be unaffected by the extensions of [32]. Furthermore, the execution of most SGX instructions does not appear to be relevant to *VC3* performance. As the enclave setup instructions ECREATE, EADD, EEXTEND and EINIT constitute only a one-time cost at initialization of *VC3*, we exclude them from the performance model. Other instructions (EGETKEY, EREPORT) are called only once during a *VC3* run and seem unlikely to have a noticeable impact on performance. In all cases, we believe that the cost on our emulator overestimates the cost on a hypothetical hardware implementation.

These simplifications allow us to focus our performance model on the cost of entering and exiting enclaves, which we conservatively model as roughly the cost of an address space switch. In particular, upon each transition, we perform a kernel transition, do a TLB flush, and execute a number of delay cycles. We perform these actions in the handlers for EENTER (enter enclave), ERESUME (enter enclave) and EEXIT (exit enclave). As interrupts during enclave execution also cause transitions, we also add the performance penalty

at the beginning and end of each interrupt that occurs during enclave execution. In particular, we patch the low-level interrupt handling code in the Windows kernel to add the TLB flush and the delay cycles. We performed a sensitivity analysis by running sample applications repeatedly while varying the number of delay cycles, but we found that our optimizations to control enclave transitions (batching of key-value pairs) allow us to reduce the performance impact of transitions to negligible levels even for large numbers of delay cycles. Therefore, for the experiments described in the evaluation, we used 1,000 delay cycles, which is similar to the cost of crossing other security boundaries such as performing system calls.

The SGX facility for encrypting and integrity protecting data before they are written from CPU caches to platform memory [41] could also affect performance. It is impossible for us to model this effect accurately since the additional cost of cache misses depends strongly on how the crypto protections for memory are implemented in hardware. However, we can estimate the cache miss rate of typical *VC3* applications. We have used the processor's performance counter for Last Level Cache Misses (LLC Misses) to measure the memory bandwidth required by the enclave code of each of the applications described in §X. In particular, we bound the execution to one core and started that core's counter upon enclave entry and stopped it upon enclave exit. We ran one application at a time. Several of the applications, in particular the reducers, used hundreds of MB of memory, which is significantly larger than the processor's L3 cache size (6 MB). The measured memory bandwidths were well below the bandwidths of modern memory encryption engines, which indicates that SGX memory encryption should not have a noticeable performance impact on *VC3*.

C. Enclave Creation

We implemented a driver (*fw.sys*) to provide functionality for enclave creation. The driver obtains the physical addresses of EPC memory from the emulator, maps pages into user mode and calls SGX instructions involved in enclave creation. Fw.sys is expected to be installed on all nodes; it would typically be distributed with the operating system.

D. Enclave and Protocols

Fw.exe acts as the host process of the enclave. It performs untrusted I/O interaction with Hadoop via the streaming protocol [5] over stdin/stdout. E^+ implements the in-band variants of both the key exchange and the job execution protocols, which work on top of the Hadoop protocol. Our implementation uses two optimizations: (i) We batch read/writes of key-value pairs from within the enclave. This is important because transitions in and out of the enclave come at a cost; we want to avoid them when possible. Our implementation processes key-value pairs in batches of 1000. (ii) We use the AES-NI instructions [30] to accelerate our implementation of AES-GCM, including the PCLMULQDQ instruction [25]. Our implementation of E^+ consists of roughly 5500 *logical lines of code* (LLOC)

of C, C++ and Assembly. About 2500 LLOC of these implement standard cryptographic algorithms. The user can inspect, change and recompile the code of E^+, or even use our protocol specification to completely re-implement it.

E. In-enclave Library

As a convenience for application development, we have created an enclave-compatible C++ runtime library. Existing C/C++ libraries which have operating system dependencies cannot be used in an enclave environment because system calls are conceptually not available [32]. Accordingly, we could neither use common implementations of the *Standard C Library* nor of the *C++ Standard Template Library*. Our library contains functions which we found useful when writing our sample applications: a set of mathematical functions, string classes, containers, and a heap allocator which manages an in-enclave heap and is the default backend for `new`. This library is relatively small (3702 LLOC) and we stress that users may choose to change it, use other libraries instead, or write their own libraries.

F. Compiler

We implemented the compiler that supports our enclave self-integrity invariants as a modification to the Microsoft C++ compiler version 18.00.30501. The implementation consists of two main parts: changes to code generation to emit our runtime checks when needed, and changes to generate data that our runtime library needs to initialize our enforcement bitmaps in the enclave. We now describe each of the parts.

We inserted our new code generation module immediately before the compiler phase that generates machine dependent code. Although our implementation of *VC3* is only for Intel x64 processors at the moment, this will allow us to target other architectures in the future. Our code generation module is intra-function only, i.e., it does not perform global static analysis. We do a pass over the instructions in a function to find any address-taken local variables; if we find any such variables, we emit code in the function's prolog and epilog to update the corresponding bits in our write bitmap. In the prolog we set the bits in the bitmap, in the epilog we clear them. Our implementation does this efficiently by generating the appropriate bit masks and setting/resetting up to 64 bits at a time. We also change the locations of these variables in the function's stack frame to make sure they do not share 8-byte memory slots with other variables (recall that we keep our bitmap information as 1 bit per every 8-bytes of enclave memory). When iterating over the instructions in a function, we insert a write check if we find a store instruction that is not a direct write to a local or a global variable. Direct writes to local or globals are stores to fixed offsets in the stack-frame or the enclave base address and are guaranteed to be inside the enclave. We also insert indirect call checks for every indirect call instructions that we find in the function. Note that we do not insert checks on function returns, because the integrity of the return addresses in the enclave stack is guaranteed by our

49

write checks. We also perform a simple intra-function analysis to simplify write checks when possible: when the write's target is a local or global variable, but the write is to a computed offset in the variable, for example an array access, we replace the full write check with a check of the form `offset < size`. Finally, when generating code to enforce region-read-write-integrity we also generate our read checks when we find load instructions whose target is not a local or global variable.

Our compiler also needs to generate data that our runtime library uses to initialize our enforcement bitmaps when starting the enclave. We generate two kinds of data: a list of addresses of address-taken functions, and a list of the addresses and sizes of address-taken global variables. These lists are simply generated by emitting the addresses in special sections of the object files whenever the compiler finds an instruction that takes the address of a function or a global variable. We perform this operation while iterating over all the code to generate the runtime checks, i.e., we do not require an extra pass over the code. We also iterate over all the initializers in global data, to find address-taken functions or address-taken global variables there. The linker merges and removes duplicates from this information when generating the binary to load into the enclave. When we create the enclave, our runtime library iterates over the addresses in these lists and sets the appropriate bits in our enforcement bitmaps.

G. Other tools

We also created several other tools to support *VC3*, including tools to generate symmetric and asymmetric keys, and tools to encrypt and decrypt data. We created a tool called *packer.exe* that encrypts E^- and merges it with E^+ to create the self-contained and signed *mapred.dll*. E^- and E^+ are first compiled into distinct DLLs. The packer statically resolves dependencies between the two DLLs and relocates both to a fixed virtual base address. It also makes sure that the DLLs' sections (e.g., `.text` and `.data`) are page-aligned, as they would be when loaded into a user mode process by the standard Windows *image loader* [53]. This is necessary to make sure that the enclave code can be loaded into memory and run unaltered without the help of the standard image loader. Users need to be able to reliably compute the enclave digest in advance. Otherwise, they could not verify statements by QEs. Our tools are incorporated into the Microsoft Visual Studio environment. They automatically create mapred.dll from a user's C++ MapReduce code.

X. Evaluation

We used the applications listed in Table I to evaluate *VC3*. We chose a mix of real-world applications and well-known benchmarks, including IO-intensive and CPU-intensive applications. We measured the performance of the applications on Hadoop, and also in isolation to remove the overhead-masking effects of disk I/O, network transfers, and spawning of Hadoop tasks. Before discussing our results, we briefly describe each application.

Application	LLOC	Size input	Size E^- (vc3)	map tasks
UserUsage	224	41 GB	18 KB	665
IoVolumes	241	94 GB	16 KB	1530
Options	6098	1.4 MB	42 KB	96
WordCount	103	10 GB	18 KB	162
Pi	88	8.8 MB	15 KB	16
Revenue	96	70 GB	16 KB	256
KeySearch	125	1.4 MB	12 KB	96

TABLE I: Applications used to evaluate *VC3*.

UserUsage and IoVolumes: Real applications that process resource usage information from a large compute/storage platform consisting of tens of thousands of servers. *UserUsage* counts the total process execution time per user. *IoVolumes* is a join. It filters out failed tasks and computes storage I/O statistics for the successful tasks.

Options: Simulates the price of European call options using Monte Carlo methods [42]. The large size of the application in terms of LLOC (see Table I) stems from the inclusion of a set of optimized mathematical functions.

WordCount: Counts the occurrences of words in the input.[2]

Pi: Benchmark that statistically estimates the value of Pi.[3]

Revenue: Reads a synthetic log file of users visiting websites and accumulates the total ad revenue per IP (from [49]).

KeySearch: Conducts a known plaintext attack on a 16-byte message encrypted with RC4 [63].

All experiments ran under Microsoft Windows Server 2012 R2 64-Bit on workstations with a 2.9 GHz Intel Core i5-4570 (Haswell) processor, 8 GB of RAM, and a 250 GB Samsung 840 Evo SSD. We used a cluster of 8 workstations connected with a Netgear GS108 1Gbps switch. All code was compiled with the Microsoft C++ compiler version 18.00.30501 for x64, optimizing for speed. We compiled our 7 applications in four configurations:

`baseline` runs the applications on plaintext data and without following the job execution protocol. Also, no performance penalty for enclave transitions (TLB flush, delay cycles, and swapping of the stack) is applied and unnecessary copying of data across (non-existent) enclave boundaries is avoided.

`vc3` runs the same application on *VC3* with encrypted mapper and reducer inputs and outputs. Sizes of the E^- DLL range from 12 KB for KeySearch to 42 KB for Options (see Table I); the generic E^+ DLL has a size of 210 KB. The enclave memory size was set to be 512 MB and the cost of an enclave transition (including interrupts) to one TLB flush and 1,000 delay cycles. This version provides the base security guarantees of *VC3*.

`vc3-w` uses the same configuration as `vc3`, but applications were compiled to further guarantee *region-write-integrity*.

---

[2]http://wiki.apache.org/hadoop/WordCount
[3]http://hadoop.sourcearchive.com/documentation/0.20.2plus-pdfsg1-1/ PiEstimator_8java-source.html
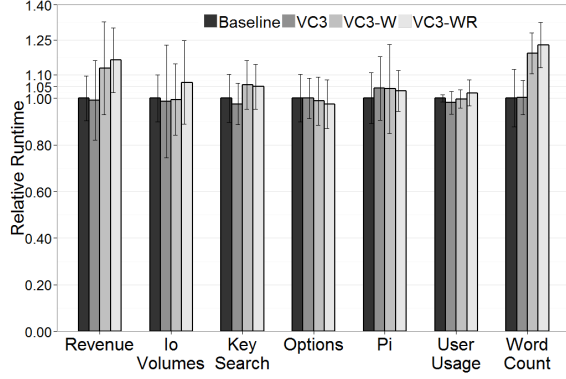
50

Fig. 5: Execution time of running MapReduce jobs in a Hadoop cluster over typical input data-sets. Running times are normalized to the performance of running the same job in normal mode and with unencrypted data (`baseline`). Note: `vc3-w` and `vc3-wr` correspond to `vc3` with extra region-write-integrity checks and region-read-write-integrity checks respectively.
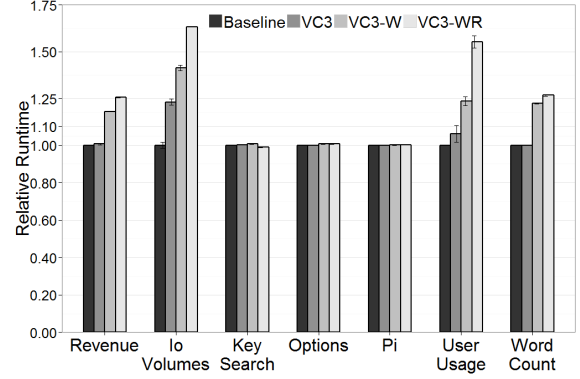

Fig. 6: Execution time of running the map phase of MapReduce jobs in isolation over typical input data-sets. Running times are normalized to the performance of running the same computation in the `baseline` configuration.

`vc3-wr` uses the same configuration as `vc3`, but applications were compiled to further guarantee *region-read-write-integrity*.

A. Performance on Hadoop

We measured the execution times of `baseline` and `vc3` in an unmodified Hadoop environment. We used the Hortonworks distribution of Hadoop 2 (HDP 2.1) for Windows with 8 worker nodes (one per workstation). We used the default configuration options for resource management, and configured our jobs to use 8 reduce tasks; except for Pi, Options, and KeySearch that conceptually use 1. We ran each job and each configuration at least 10 times and measured the execution time. To facilitate comparisons, we normalized the running times with the average running time for each job using the `baseline` configuration. Figure 5 plots the average ratios for each job and configuration, and the values of two standard deviations below and above each average.

Figure 5 shows that `vc3`'s performance is similar to `baseline`; the differences in performance are well below the experimental variability for all jobs. `vc3`'s overhead is negligible with its base security guarantees. When introducing the write and read-write integrity checks, the performance overhead increases on average by 4.5% and 8% respectively. The increased overhead is a small price for the extra security guarantees. We believe these results show that *VC3* can be used in practice to provide general-purpose secure cloud computation with good performance.

B. Performance in Isolation

When running applications, Hadoop performs many activities, such as spawning mappers and reducers, waiting for disk I/O, network transfers, and others, that may mask the overheads of *VC3*. To better understand the performance impact of *VC3* on the execution times of individual map and reduce tasks, we ran the mappers and reducers in isolation, i.e., on a single machine without Hadoop. We repeated each experiment 10 times, and, as in Section X-A, we normalize using the average of the `baseline` run. Figure 6 plots the average ratios for the map tasks, as well as the values of two standard deviations below and above the average. (The results for reduce tasks are similar — we omit them for brevity.)

On average, `vc3`'s overhead was 4.3% compared to `baseline`, `vc3-w`'s was 15.3%, and `vc3-wr`'s was 24.5%. The overheads were negligible for the three compute intensive jobs (Key Search, Options, and Pi); these jobs spend little time in copying and encryption/decryption operations, and most of the time they compute using plain-text data off of the processor's caches; in addition, for these jobs the compiler was effective at eliding checks on safe memory accesses, and hence the overheads of `vc3-w` and `vc3-wr` are also negligible.

The IoVolumes and UserUsage jobs were slower than `baseline` in all configurations. The IoVolumes(UserUsage) job was 23.1%(6.1%), 41.4%(23.6%) and 63.4%(55.3%) slower in the `vc3`, `vc3-w`, and `vc3-wr` configurations respectively. The overheads are higher in these cases, because these applications are IO-intensive. Since they perform little computation, the relative cost of encryption is higher. Revenue and WordCount are also IO-intensive, but these applications implement a combine operation which increases the computation performed at the mapper, hence reducing the relative cost of encryption. The combine operation performs a group-by of the key-value pairs generated by the map function, and calls a combine function that performs a partial reduction at the mapper. This is a common optimization to reduce network traffic (IoVolumes does not implement combining, which contributes to its larger overhead). We thus observe little performance difference between `baseline` and `vc3` for Revenue and WordCount. The write(read-write) integrity checks increased their running times by 18%(26%) and 22%(27%) respectively. The performance differences between `vc3` and `vc3-w`/`vc3-wr` are due to the region self-integrity checks

and they vary according to the ability of the compiler to check if memory accesses are safe at compile-time.

We now turn our attention to the performance difference between `baseline` and `vc3`. This difference is due to: (i) copying data to and from the enclave, (ii) encryption and decryption operations, and (iii) enclave transitions (either due to normal operations or due to system interrupts). We measure the copy operations and the total encryption and decryption times and use that to explain the difference in the execution times of `vc3` versus `baseline`. We find that the crypto(copying) operations contributed 13.3(4.8) percentage points for IoVolumes; the crypto operations dominated the overhead for UserUsage. Despite the use of hardware acceleration to speed up encryption and decryption (with the AES-NI instructions, see section IX), there is a performance penalty for using encrypted data; this is unavoidable in our setting.

We believe that in all cases the overheads are reasonable for the provided security guarantees. Moreover, when run as part of Hadoop jobs, these overheads have small (if any) impact on the total run time (Figure 5).

C. Effectiveness of region self-integrity

We also conducted fault-injection experiments to verify the effectiveness of the region self-integrity invariants. We wrote a tool that injects three types of faults in the source code of applications: writes to a random address outside of the enclave, reads from a random address outside the enclave, and pointer dereferences that corrupt a return address inside the enclave. For each type of fault, we conducted 10 experiments per application. In all cases the region self-integrity checks caught the invalid access and stopped the application.

XI. RELATED WORK

Applications of SGX were first discussed in [27]. Haven [9] is a recently proposed SGX-based system for executing Windows applications in the cloud. Haven loads a given application together with a library OS variant of Windows 8 into an enclave. Haven makes a different trade-off between security and compatibility: it can run unmodified Windows binaries, but its TCB is larger than *VC3*'s by several orders of magnitude. Unlike *VC3*, Haven neither guarantees integrity for distributed computations, nor does it provide our region self-integrity properties. Brenner et al. presented an approach to run Apache ZooKeeper in enclaves [14].

Several systems protect confidentiality of data in the cloud. Fully homomorphic encryption and multiparty computation [21], [22] can achieve data confidentiality, but they are not efficient enough for general-purpose computation. CryptDB [50] and MrCrypt [61] use partial homomorphic encryption to run some computations on encrypted data; they neither protect confidentiality of code, nor guarantee data integrity or completeness of results. On the other hand, they do not require trusted hardware. TrustedDB [7], Cipherbase [6], and Monomi [64] use different forms of trusted hardware to process database queries over encrypted data, but they do not protect the confidentiality and integrity of all code and data.

Monomi splits the computation between a trusted client and an untrusted server, and it uses partial homomorphic encryption at the server. Mylar [51] is a platform for building Web applications that supports searches over encrypted data.

Several systems combine hardware-based isolation [37], [46], [59] with trusted system software [17], [28], [36], [38], [54], [58], [69], which is typically a trusted hypervisor. The Flicker [39] approach uses TXT [31] and avoids using a trusted hypervisor by time-partitioning the host machine between trusted and untrusted operation. Virtual Ghost [18] avoids using a trusted hypervisor and specialized hardware-based isolation mechanisms by instrumenting the kernel.

Some systems allow the user to verify the result of a computation without protecting the confidentiality of the data or the code [48]. Pantry [13] can be used to verify the integrity of MapReduce jobs which are implemented in a subset of C. Pantry incurs a high overhead. Hawblitzel et al. presented the concept of formally verified Ironclad Apps [26] running on partially trusted hardware. They report runtime overheads of up to two orders of magnitude.

Several security-enhanced MapReduce systems have been proposed. Airavat [52] defends against possibly malicious map function implementations using differential privacy. SecureMR [67] is an integrity enhancement for MapReduce that relies on redundant computations. Ko et al. published a hybrid security model for MapReduce where sensitive data is handled in a private cloud while non-sensitive processing is outsourced to a public cloud provider [33]. PRISM [12] is a privacy-preserving word search scheme for MapReduce that utilizes private information retrieval methods.

XII. CONCLUSIONS

We presented *VC3*, a novel approach for the verifiable and confidential execution of MapReduce jobs in untrusted cloud environments. Our approach provides strong security guarantees, while relying on a small TCB rooted in hardware. We show that our approach is practical with an implementation that works transparently with Hadoop on Windows, and achieves good performance. We believe that *VC3* shows that we can achieve practical general-purpose secure cloud computation.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[4] Apache Software Foundation. Hadoop. http://wiki.apache.org/hadoop/, Accessed: 11/05/2014.

[5] Apache Software Foundation. HadoopStreaming. http://hadoop.apache.org/docs/r1.2.1/streaming.html, Accessed: 11/05/2014.

[6] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[7] S. Bajaj and R. Sion. TrustedDB: A trusted hardware-based database

52

with privacy and data confidentiality. In *IEEE Transactions on Knowledge and Data Engineering*, volume 26, 2014.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[9] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology—CRYPTO*, 1998.

[11] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT*, 2000.

[12] E.-O. Blass, R. Di Pietro, R. Molva, and M. Önen. Prism—privacy-preserving search in MapReduce. In S. Fischer-Hübner and M. Wright, editors, *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[13] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[14] S. Brenner, C. Wulf, and R. Kapitza. Running ZooKeeper coordination services in untrusted clouds. In *USENIX Workshop on Hot Topics in Systems Dependability (HotDep)*, 2014.

[15] E. Brickell and J. Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. In *IEEE International Conference on Social Computing (SocialCom)*, 2010.

[16] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[17] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[18] J. Criswell, N. Dautenhahn, and V. Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[19] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology—CRYPTO*, 2012.

[20] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[21] C. Fournet, M. Kohlweiss, G. Danezis, and Z. Luo. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security Symposium*, 2013.

[22] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[23] C. Gentry and S. Halevi. Implementing Gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology—EUROCRYPT*, 2011.

[24] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, 2014.

[25] S. Gueron and M. E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode, 2010. No. 323640-001.

[26] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[27] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[28] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[29] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

[30] Intel Corp. Intel 64 and IA-32 architectures software developer's manual—combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c, 2013. No. 325462-048.

[31] Intel Corp. Intel trusted execution technology. software development guide, 2013. No. 315168-009.

[32] Intel Corp. Software guard extensions programming reference, 2013. No. 329298-001.

[33] S. Y. Ko, K. Jeon, and R. Morales. The Hybrex model for confidentiality and privacy in cloud computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[34] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[35] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[36] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. MiniBox: A two-way sandbox for x86 native code. In *Usenix ATC*, 2014.

[37] D. Lie, M. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[38] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig. Trustvisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2010.

[39] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *European Conference on Computer Systems (EuroSys)*, 2008.

[40] D. McGrew and J. Viega. The Galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process, 2004.

[41] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[42] G. Morris and M. Aubury. Design space exploration of the European option benchmark using hyperstreams. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2007.

[43] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[44] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, 2010.

[45] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2014.

[46] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. Oasis: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[47] B. Parno. Bootsrapping trust in a "trusted" platform. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.

[48] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[49] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD International Conference on Management of Data*, 2009.

[50] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[51] R. A. Popa, E. Stark, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using Mylar. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[52] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[53] M. Russinovich and D. Solomon. *Windows Internals, Part 1*. Microsoft Press Corp., 6th edition, 2012.

[54] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security Symposium*, 2012.

[55] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, G. Mainar-Ruiz, and M. Russinovich. $VC3$: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, Microsoft Research, 2014.

[56] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and SSH timing attacks. In *USENIX Security Symposium*, 2001.

[57] R. Strackx, B. Jacobs, and F. Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Anual Computer Security Applications Conference (ACSAC)*, 2014.

[58] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[59] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing (ICS)*, 2003.

[60] L. Szekeres, M. Payer, T. Weiz, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, 2013.

[61] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[62] Trusted Computing Group. Trusted platform module main specification. version 1.2, revision 103, 2007.

[63] K. H. Tsoi, K.-H. Lee, and P. H. W. Leong. A massively parallel RC4 key search engine. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2002.

[64] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *VLDB*, 2013.

[65] M. Van Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2010.

[66] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[67] W. Wei, J. Du, T. Yu, and X. Gu. Securemr: A service integrity assurance framework for mapreduce. In *Anual Computer Security Applications Conference (ACSAC)*, 2009.

[68] C. Wright, L. Ballard, S. Coull, F. Monrose, and G. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *IEEE Symposium on Security and Privacy*, 2008.

[69] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

APPENDIX

A. Multi-User Key Exchange

We describe a variant of our basic key exchange protocol in case there are several users $u \in U$, each contributing their own set of input splits (using separate keys), and all getting access to the output. For simplicity, it is assumed that online communication channels between the involved parties exist. It is though possible to implement the multi-user protocol as regular MapReduce job without such channels the same way it is described in §V-C for the single-user protocol.

1) Each user is still identified by its public encryption key pk_u.

2) The users agree on the code to run; they exchange fresh random shares j_u, $k_{code,u}$, $k_{job,u}$, $k_{out,u}$ and compute $j = \oplus_{u \in U} j_u$, $k_{code} = \oplus_{u \in U} k_{code,u}$, $k_{job} = \oplus_{u \in U} k_{job,u}$, $k_{out} = \oplus_{u \in U} k_{out,u}$. They then prepare the enclave code $C_{j,u}$ as above (using the same randomness for the encryption), except that all their public keys $(pk_u)_{u \in U}$ are included in the $C_{j,u}$ package.

3) Each enclave prepares and sends the message of the base protocol p_w to every user, each encapsulating a distinct, fresh, symmetric key $k_{w,u}$. (The collection of messages may be jointly quoted once by each effective QE.)

4) Each user independently receives, verifies, and decrypts its message, then sends the encrypted job credentials $\mathsf{Enc}_{k_{i,u}}[]\{k_{code} \mid k_{job} \mid k_{in,u} \mid k_{inter,u} \mid k_{out} \mid k_{prf,u}\}$, where $k_{in,u}$, k_{out} are the authenticated-encryption keys for their input and the output file, and where $k_{inter,u}$ and $k_{prf,u}$ are their fresh shares of the keys for the intermediate key-value pairs and the pseudo-random function PRF respectively.

5) Each enclave decrypts all job credentials, checks that they all provide the same keys k_{code}, k_{job}, k_{out}, and computes $k_{inter} = \oplus_{u \in U} k_{inter,u}$ and $k_{prf} = \oplus_{u \in U} k_{prf,u}$.

At this stage, k_{code}, k_{job}, and k_{out} are known to every user and every enclave in the job; $k_{in,u}$ is known to every enclave and to user u; k_{inter} and k_{prf} are known to every enclave in the job, but not to any strict subset of users. *VC3* does currently not implement a multi-user key exchange but we plan to support it in the future.

B. Lightweight Key Exchange

The in-band key exchange protocol (§V-C) implemented in our *VC3* prototype works well with existing Hadoop installations, but requires executing a full (though lightweight) MapReduce job just for exchanging keys.

In case the user is willing to put extended trust into a special *Support Enclave* (SE), the necessity for a separate key exchange job can be avoided while maintaining compatibility with existing Hadoop installations. We briefly describe a corresponding protocol in the following.

As described in §V-A, we generally foresee the cloud provider to deploy a Cloud QE to each of its SGX-enabled nodes that is functionally equivalent to the standard SGX QE. Here, a special *Support Enclave* (SE) with extended functionality is deployed *instead* of such a Cloud QE. Instead of an EPID private key, each SE creates and manages a node-specific long-term RSA key pair that is permanently sealed to local storage. Each SE acquires a quote from the regular local SGX QE for its RSA public key. The cloud provider makes the quoted public keys for all SEs available to the user. For each SE, the user verifies the quotes and sends

$$\mathsf{Enc}_{k_e}[C_{j,u}]\{k_{code} \mid \mathbf{k}\} \mid \mathsf{PKEnc}_{pk_{SE}}\{k_e\}$$

where k_e is a fresh job-specific ephemeral symmetric key. Each SE decrypts k_e and then verifies $C_{j,u}$ and decrypts $k_{code} \mid \mathbf{k}$. Subsequently, on each node, a new enclave containing $C_{j,u}$ is started. A mutually authenticated secure local channel between E^+ (running in the newly created enclave) and SE is created using local attestation (see §II-B). The SE passes $k_{code} \mid \mathbf{k}$ over this channel to E^+. Finally, E^+ decrypts E^- and the enclave is able to process job data afterwards.

In this protocol variant, the user needs to trust the SE deployed by the cloud provider as it handles k_{code} and $\mathbf{k}$ in the clear. In order to facilitate the establishment of this trust, the cloud provider should make the source code of the SE publicly available.