Computation Certification as a Service in the Cloud*

Safwan Mahmud Khan and Kevin W. Hamlen
Department of Computer Science
The University of Texas at Dallas
Richardson, Texas, USA
{*safwan,hamlen*}*@utdallas.edu*

***Abstract*—This paper proposes a new form of Security as a Service (SECaaS) that allows untrusted, mostly serial computations in untrusted computing environments to be independently and efficiently validated by trusted, commodity clouds. This addresses the longstanding problem of safely executing high assurance computations on untrusted hosts.**

Untrusted computations are instrumented with a checkpointing mechanism that yields a proof of computation integrity as the computation progresses. This proof can be validated by a trusted cloud to ensure that the computation was carried out faithfully. Cloud parallelism and replication is leveraged to validate the proof efficiently even when the original computation is not parallelized. This affords a means of high-assurance, serial computation on cloud-aware, mobile devices that mix resource-rich but untrusted hardware with trusted but comparatively resource-impoverished hardware components. An implementation for Java and Hadoop MapReduce demonstrates that the approach is effective for commodity VMs, clouds, and software.

***Keywords*-result checking; computation integrity; software security; cloud computing; Hadoop MapReduce**

I. Introduction

Protecting remote software from corruption by untrusted or malicious host environments has long been an important challenge for Trustworthy Computing (TwC) paradigms, such as mobile devices that mix trusted and untrusted hardware [1], and trustworthy grids that distribute computations to remote, untrusted hosts [2]. In these contexts, *untrusted environments* are computing platforms (e.g., hardware, OSes, and VMs) that have unfettered access to the distributed computations they receive, including the ability to tamper with the mobile code, its program state, and its results. To achieve high reliability and integrity for computations, secure grids must prevent or detect all such tampering for each computation they distribute.

Many existing platforms therefore aggressively apply *remote attestation* technologies to detect and preclude software tampering in untrusted environments [3], [4]. For example, hardware- and software-based attestation mechanisms evidence the integrity of remote client states through cryptographically signed memory snapshots taken statically or at runtime [5], [6]. However, code integrity alone does not guarantee that a computation result is correct. For instance, an attacker may run the software without any alterations but still return corrupted results to the requester. Code integrity checking must therefore be coupled with result integrity checking, which usually involves embedding a secret subcomputation that is difficult to reverse engineer and that can be checked by the requester [7], or by refactoring distributed computations into function compositions that can be validated cryptographically [8], [9], [10].

Unfortunately, all of these approaches require a significant redesign of most software. For example, typical Android apps are not easily modified to contain inextricable, secret computations or cryptographically verifiable compositions. As a result, few mainstream mobile computing devices have adopted these technologies. Moreover, many of these solutions rely on software obfuscation [11], [12], which does not provide rigorous guarantees, since clever attackers can potentially reverse the obfuscation.

In contrast, clouds [13], [14], [15] are an increasingly popular grid computing paradigm [16] utilized by myriad mobile device architectures [17]. Clouds offer massive parallelism and potentially high integrity assurance through replication. For example, trust management has been used in clouds to ensure that even if some cloud nodes are malicious, computation results are nevertheless correct with high probability [18]. The favorable business model for cloud-powered mobile apps has led to a rapidly growing mobile cloud market that is expected to exceed $9 billion by 2014 [19].

While clouds offer an attractive means for mobile devices to remote their high assurance computations, cloud-assisted devices still face at least two significant limitations in practice: First, most mobile devices do not have perpetual, continuous access to the cloud. Thus, many of their computations must be carried out using purely local resources, pending cloud access. We observe that such devices would benefit from a *trust-but-verify* model in which computations are initially carried out locally using a potentially untrusted environment (e.g., insecure CPU and storage), trusted for a limited time, but then verified once cloud access is available. For example, a password that unlocks a software product could be verified locally, allowing the software it protects to be used for

*This paper contains material supported in part by National Science Foundation grant #1065216 and U.S. Air Force Office of Scientific Research grant FA9550-10-1-0088. All opinions and conclusions expressed are those of the authors and not necessarily of the NSF or AFOSR.

a limited time, after which the tamper-proof portion of the hardware seeks validation of the password verification computation by the cloud.

Second, although clouds offer high parallelism, most everyday app computations are not highly parallelized and therefore derive little or no benefit from such parallelism. Thus, we seek a computation verification strategy that allows mostly serial computations performed in an untrusted environment to be rapidly validated using the massive parallelism offered by the cloud. Such validation empowers clouds with a new form of Security as a Service (SECaaS) that provides high assurance for local, untrusted, mostly-serial computations.

Our answer to this challenge is a Cloud-based COmputation VERifier (CloudCover) that allows untrusted Java computations to yield a *proof of computation integrity* as a side-effect of the computation. The proof can then be validated against the original code and the computation's result to formally verify that the result is correct. Neither the computation nor the proof (nor their origins) are trusted by CloudCover. A (possibly forged) proof either proves that a given computation results from a given code, or it does not. If the former, the result is correct regardless of where the proof came from; if the latter, the computation, the proof, or both are untrustworthy. Thus, CloudCover can be formalized as *proof-carrying computation*, in the spirit of proof-carrying code [20].

CloudCover proofs have the advantageous quality that the task of verifying them can be parallelized almost arbitrarily even when the original computation is not parallelizable. Thus, they derive maximal benefit from massively parallel architectures, like clouds. To demonstrate, we implement CloudCover for Hadoop MapReduce [21], and use it validate non-parallelizable Java computations for message digest generation using SHA-1 [22] and MD5 [23] cryptographic hash functions. Experimental results indicate that CloudCover scales extremely well, with the only practical limit to parallelization stemming from the fixed overhead of dispatching new mappers and reducers.

Our work therefore offers the first computation integrity validation mechanism that

- requires minimal changes to existing software;
- fully leverages the massive parallelism available from commodity data processing clouds, such as MapReduce;
- provides a tunable range of integrity assurances, from rigorous, absolute assurance to probabilistic assurance (with verification overhead scaling linearly with assurance level); and
- is applicable to everyday mobile app computations, such as those that contain mostly serial computations implemented in interpreted bytecode languages like Java.

Section II begins with a presentation of the system details of CloudCover. Our implementation and experimental results are outlined in §III and §IV, respectively. Section V discusses related work. Finally, §VI concludes with a brief discussion of directions for future work.

II. System Overview

A. Architecture and Threat Model

We consider two possible system architectures, one in which trusted and untrusted devices are physically separate, and one in which trusted and untrusted hardware is co-located on a single mobile device. Both architectures are illustrated in Fig. 1. In both cases, a resource-impoverished, *trusted component* wishes to distribute a computation to an *untrusted component* that is comparatively resource-rich.

The trusted component cannot efficiently distribute the computation across the cloud directly because (a) it currently lacks cloud access, (b) the cloud's computing power is based on parallelism, which goes unutilized when the computation is mostly serial, and/or (c) the computation is tentative in the sense that only certain outcomes demand verification, and the computation outcome is not known in advance. The last case arises frequently in grid computing. For example, SETI@home [24] aggressively validates only those computations whose results suggest the existence of extraterrestrial intelligence.

In each scenario, we assume that the trusted component is secure in the sense that no malicious user has access to it, or those that do cannot corrupt its computations, storage, or state. Typically we expect that such components consist of expensive, more secure hardware that is less efficient or more constrained due to its extra security. In contrast, untrusted components consist of insecure hardware and/or remote, untrusted machines that are entirely exposed to malicious users and activities.

The *checker* of our system is a cloud computing platform that may consist of hundreds or thousands of nodes. It is a high assurance, massively parallelized data processing framework whose computation results are trusted, but it is best applied to highly parallelized computations. Assigning it serial computations is prohibitively slow and expensive.

Figure 1 summarizes the system workflow together with the architecture. The trusted component first instruments the mobile code with proof-generation logic. The instrumented code is then distributed to the untrusted component. A cooperative recipient executes the computation faithfully, yielding a computation result and a proof of computation integrity. The code, result, and proof can then be sent to the cloud (when it becomes available) for parallelized verification. A malicious, untrustworthy, or compromised recipient, however, returns an incorrect result. There exists no proof that the original code yields such a result, so cloud validation inevitably fails, irrespective of the proof submitted by the untrusted component. Thus, the incorrect result is rejected.

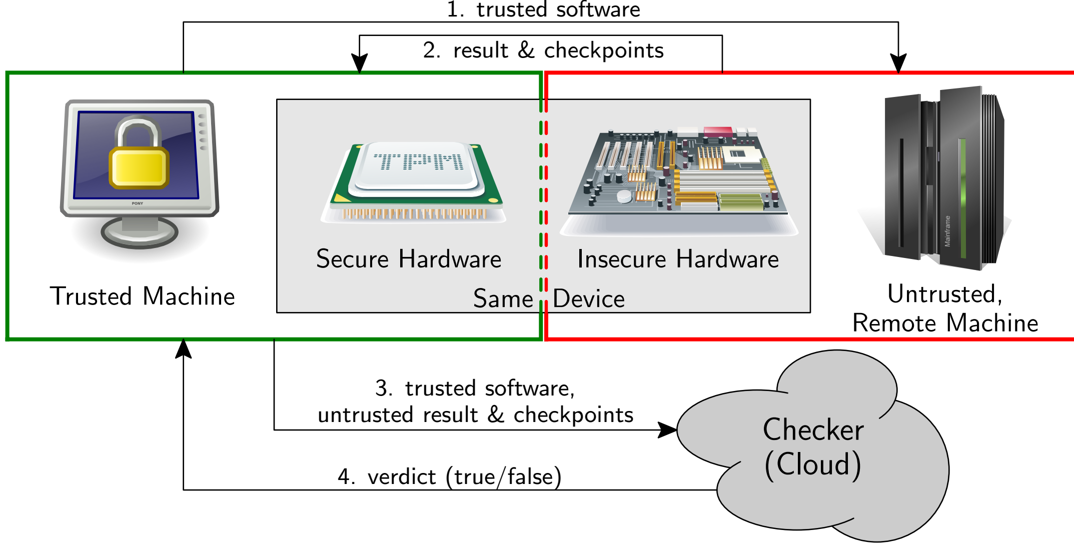

Figure 1. System architecture of CloudCover

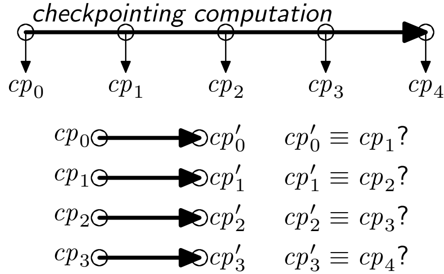

Figure 2. CloudCover checkpointing and validation

B. Computation Integrity Proof Generation and Validation

CloudCover approaches the problem of proof generation through *checkpointing*, as illustrated in Fig. 2. Mobile Java code is instrumented with a checkpoint operation that periodically saves the current program state to disk. Initial checkpoint cp_0 is the program start state and is fully characterized by the code itself, so needn't be generated explicitly. The last checkpoint cp_n characterizes the computation result.

A *chain* of such checkpoints constitutes a proof that a computation whose initial state is cp_0 yields result cp_n. The proof can be validated by recomputing all the segments of the chain in parallel. That is, for each checkpoint $i < n$, cloud node i initializes its JVM to state cp_i and computes until the next checkpoint, yielding state cp'_i. It then decides whether $cp'_i \equiv cp_{i+1}$. If any of these equivalence checks fail, the proof fails and the computation is rejected. This transforms a serial computation into a fully parallelized re-computation that only takes as long as the longest checkpoint interval (plus some time for the checkpoint equivalence check) to validate. The equivalence check can be additionally parallelized, as discussed in §III.

Proof validation through checkpoint chaining engenders a natural trade-off between assurance and computational expense through *spot-checking*. A spot-checking validator recomputes and checks each segment in the checkpoint chain with probability p. This reduces the total computation cost to a fraction p of the total, and detects erroneous computation results with probability p. Thus, clients may tune parameter p in accordance with their desired level of assurance and the expense of cloud computing time.

One naïve way to implement checkpointing for Java programs is to take a system-level snapshot of the JVM process image at fixed time intervals. However, this approach is inadequate for computation certification for at least two reasons: (1) JVM process images vary greatly at the byte level depending on the particular JVM version and the underlying hardware. For example, different JVMs have radically different underlying implementations, including memory allocation strategies, JIT compilation behavior, and a variety of other low-level details. These differences are transparent to Java programs, but they make it difficult or impossible to compare system-level JVM process images for semantic equality. (2) Even when comparing process images from identical JVM versions on identical hardware, semantic equivalence of the JVM state is not an identity function. Many JVM objects have internal fields, such as hash values and clock times, that are irrelevant to semantic object equivalence.

We therefore implement checkpointing at the Java level rather than the system level. Our implementation extends Apache's open source Javaflow library [25]. Javaflow includes a *suspend* operation that generates *continuations*. Each continuation object contains a snapshot of the stack trace, including the call stack, local and global variables, and the program counter. Such continuations can later be resumed (i.e., replayed) by passing the continuation object to the

library's *continueWith* method. As a simple example, a program that prints numbers from 1 to 100 can be suspended immediately after printing 50. The resulting continuation can be resumed later, resulting in the program printing 51, etc., until another suspension is encountered. Continuations can be serialized for mobile execution. We leverage continuations as checkpoints for CloudCover.

Although Javaflow supports suspension and resumption of computations via continuations, it does not support continuation equivalence-checking. This is necessary for the checkpoint equivalence check in Fig. 2. CloudCover therefore extends Javaflow's *Continuation* class with an *equals* method that compares two suspended program states for semantic equivalence. Two states are equivalent if they consist of equal-length stacks whose corresponding slots contain equivalent values and objects. Deciding such semantic equivalence is non-trivial in general; for example, the states may contain objects with private fields to which the continuation object lacks access, or they may include fields whose values are semantically equivalent but non-identical. Fortunately, all Java objects have their own *equals* methods, which encode an object-specific notion of semantic equivalence.

Every Java program therefore carries within itself a *general contract* of object-equality [26], encoded by the collective implementations of all its *equals* methods. This contract can be leveraged to decide semantic equivalence of arbitrary program states. It is this insight that allows CloudCover to validate computations without any significant change to existing Java programs.

C. CloudCover Protocol

CloudCover's protocol is detailed in Algorithm 1. We denote the trusted component, untrusted component, and checker as *TC*, *UC*, and *C* (respectively) in the algorithm. The order of the generated checkpoints is important since the checker accepts a checkpoint as input, resumes it, and matches its result with the immediate next checkpoint. If there are n checkpoints, it therefore organizes $n-1$ pairs. For line 11, we use our customized *equals()* method included in the modified Javaflow library.

D. Attacks and Defenses

The following are some ways in which a malicious untrusted component might attack CloudCover:

- Untrusted components can alter the number and/or positions of checkpoints. This is discovered by the checker with probability p when it compares the first altered checkpoint with the one yielded by the trusted software, since their memory states and/or program counters must differ.
- Untrusted components can modify other parts of the software. This threat is certainly detectable since the checker runs the trusted software received from the trusted component, not from the untrusted one, and thus discovers the difference.
- Untrusted components can leave the code uncorrupted, but tamper with the checkpoints and/or results it sends to the trusted component. This is discovered with probability p when a checkpoint equivalence check fails.

Algorithm 1 CloudCover Protocol

```
1:  TC chooses number n ≥ 2 and placement of checkpoints
2:  TC inserts n − 2 suspend calls into software (the initial
    and final checkpoints are implicit)
3:  TC sends modified (trusted) code c to UC
4:  UC executes c and sends generated checkpoints and
    result to TC
5:  TC sends c, checkpoints, and result to C
6:  C organizes checkpoints into pairs (cp_i, cp_{i+1})
7:  C dispenses (c, cp_i, cp_{i+1}) as input to each computation
    unit (e.g. mappers in Hadoop)
8:  for each checkpoint cp_i where i ∈ [0, n) do
9:    if rand() ≤ p then
10:     cp'_i ← continueWith(cp_i)
11:     if cp'_i ≡ cp_{i+1} then
12:       computation unit returns true
13:     else
14:       computation unit returns false
15:     end if
16:   end if
17: end for
18: return the conjunction of all the unit return values
```

CloudCover trusts the cloud platform. Clouds can attain suitable trustworthiness through trust management, replication, virtualization, and a variety of other technologies (e.g., Hatman [18], AdapTest [27] or RunTest [28]) not typically available to mobile devices and other, stand-alone, cloud-assisted machines.

Privacy preservation of computation results is beyond our scope. For such protection, we refer the reader to numerous related works on that subject, including AnonymousCloud [29], secure multiparty computation [30], and differential privacy [31].

III. Implementation

Implementation of CloudCover for a real-world architecture is a key contribution of our work. We therefore target Java computations, which are the basis for many mobile app domains, and we implement computation validation using a commodity data processing cloud—Hadoop MapReduce. We leverage Javaflow's built-in functionalities for taking checkpoints of software and resuming software from checkpoints. Our custom implementation of *equals()* for Javaflow continuations decides semantic equivalence of checkpoints.

The checker is deployed on a Hadoop [15] cluster consisting of 6 DataNodes and 1 NameNode. Node hardware

is comprised of Intel Pentium IV 2.40, 3.00GHz processors with 2–4GB of memory each, running Ubuntu operating systems. Javaflow was installed and configured on each DataNode in the Hadoop distributed environment, making it available to distributed jobs. We implemented a mechanism for reading and writing checkpoints for mappers in Hadoop in an appropriate file format for equality-checking with Javaflow. LZO compression was applied to all Hadoop file transfers to minimize transfer and storage costs. For trusted and untrusted components of CloudCover, we use standard desktop computers with configurations similar to the individual cloud nodes above.

For experiments, we select two non-parallelizable cryptographic hash functions, SHA-1 [22] and MD5 [23], which yield message digests. These were instrumented with Javaflow checkpointing operations placed within their inner loops. Both algorithms are widely used in TwC, yet not parallelizable beyond fine-grained, instruction-level optimization (cf., [32]). This makes them good subjects for our tests. Both functions take strings of arbitrary length as input, where SHA-1 and MD5 produce 160-bit and 128-bit message digests, respectively. We choose fairly long strings as inputs in our experiments to demonstrate the benefits of fast, parallelized validation of comparatively long, serial computations.

Aside from verifying checkpoint chain segments in parallel, we additionally parallelized the checkpoint equality checking procedure in our implementation. Continuations are stacks that can be partitioned arbitrarily into sub-stacks that can all be checked in parallel for equivalence. We implemented this for Javaflow by introducing a continuation *compare* method. During comparison, instead of equality-checking each pair of objects inside the checkpoints, a mapper can redirect them to other mappers by submitting new jobs in Hadoop. The advantage is that if any individual checkpoint-pair is extremely large (e.g., very large stacks), then the checkpoint equality-checking job can be parallelized to compensate. In our experiments, the stacks are not that large, so this feature went unexercised. (With small stacks, parallelizing the equality-checking task is not worthwhile, since the task of splitting the stacks introduces more overhead than it saves.)

IV. Experimental Results

In all our experiments, any corruption of checkpoints (e.g., moving, modifying, or omitting them) and/or corruption of results provoked rejection by the checker (see §II-D). The remainder of our evaluation therefore focuses on performance.

Our first experiment (Fig. 3) illustrates the superiority of CloudCover over a single machine verifier for SHA-1 applied to a 38KB input string. We produce 391 checkpoints for this experiment and observe that the Hadoop cluster clearly outperforms a single machine—with 6 nodes, it exhibits approximately 23% gain. Although adding nodes reduces the overall validation time, it suffers diminishing returns typical of parallel architectures. The diminishing returns are

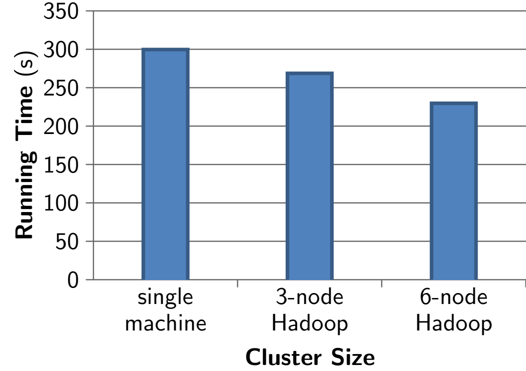

Figure 3. SHA-1 computation verification time

primarily due to additional overhead for file I/O on HDFS (Hadoop's file system) and additional network communication per additional node. Additionally, 391 checkpoints for this experiment spawns a number of parallel mappers that exceeds the total capacity of our Hadoop cluster. It therefore places many mappers in the queue. For this reason, a 3-node cluster takes less than double the time of a 6-node cluster (but still beats the single node performance by 10%). With a larger cluster of hundreds or thousands of machines, we therefore expect even better performance than exhibited by our comparatively small-scale testbed.

Our second experiment (Fig. 4) considers an MD5 algorithm applied to a 38KB input string using the same cluster sizes. It generates 612 checkpoints and achieves a similar performance gain: 3-node and 6-node Hadoop clusters run 12% and 25% faster, respectively, than a single machine setup. The increased number of checkpoints results in longer validation times than the first experiment. This indicates that for any given computation and cluster configuration, there may be an optimal number and distribution of checkpoints. In addition, although the number of checkpoints is 57% greater, the verification time is more than 57% longer than for SHA-1. This is because the MD5 computation's checkpoints are much larger on average than those generated for SHA-1, because the MD5 implementation places more large objects on its stack at checkpoint times. Accordingly, an optimal implementation should select checkpoint positions strategically so as to avoid such overhead when possible.

Figure 5 reports certification times for SHA-1 computations over various input string lengths using 6 checkpoints. For our small cluster, 6 checkpoints was an optimal number (more and fewer checkpoints yielded higher overall runtimes). We expect that with larger clusters the optimal number of checkpoints rises proportionately to the cluster size. With a single machine, the verification time climbs rapidly with the input string length, whereas for Hadoop clusters it climbs much more slowly. For instance, where the performance difference between 3-node Hadoop and a single machine is about 5.5% for a 38KB string, it is about 60% for

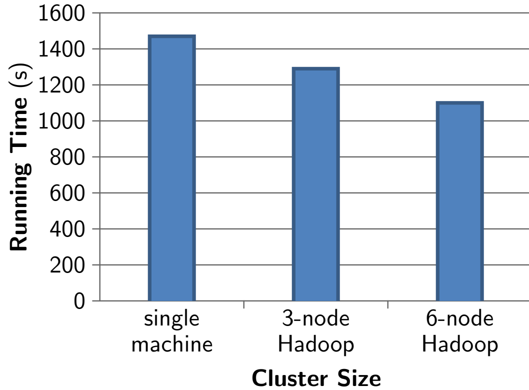

Figure 4. MD5 computation verification time

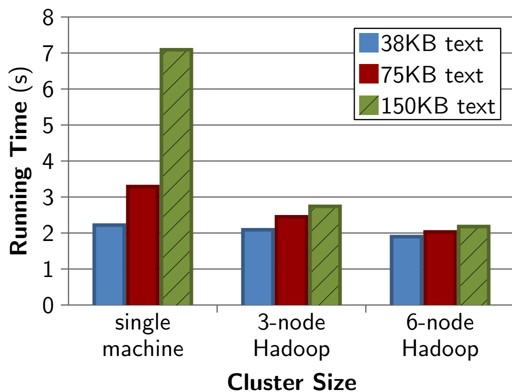

Figure 5. SHA-1 verification times with 6 checkpoints

a 150KB input (4 times as large). Thus, the longer the serial computation, the greater advantage is observed for the parallelized validation architecture.

CloudCover's parallelized verification strategy scales best when applied to certify computations whose time-complexities are greater than their space-complexities. In these cases, CloudCover's division of the original computation's runtime across n nodes introduces speed-ups that rapidly outpace the overhead introduced by checkpoint operations, which typically have time complexity equal to the computation's space-complexity. To illustrate, Fig. 6 compares the certification times for a quadratic-time, linear-space, insertion sort computation against the runtimes of the original, serial computation without any checkpointing. With a 600K-element input array and a 6-node cluster, certification completes 74% faster than the original computation on the same cloud. (Since the original computation is serial, the cloud cannot parallelize it and it runs on only one node.)

V. Related Work

Automatic result checking has been studied in the literature for at least a quarter century. It was first proposed as a means of debugging software [33]. Later work extended the idea to fault tolerance by observing that certain algorithms can

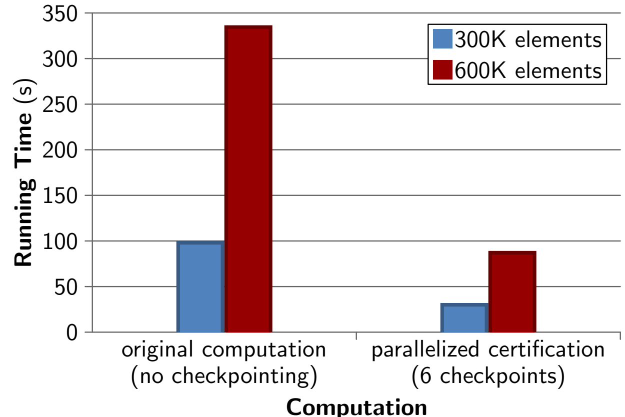

Figure 6. Original computation runtimes vs. verification runtimes for insertion sort on a 6-node Hadoop MapReduce cluster

be reformulated to yield a *certification trail* of data that witnesses the integrity of the algorithm's result [34]. When available, such a trail can be verified independently by a distinct, faster certification algorithm to achieve efficient result checking. This insight has led to recent work in the formal methods community on frameworks for developing trail-producing software and their certifiers [35].

Unfortunately, the addition of certification trails to software is non-trivial in general (justifying the application of formal methods). It typically requires reformulating and reimplementing the algorithm, as well as developing a completely new certification stage that is unique to each algorithm being checked. Applying the technique to most existing, production-level software is therefore a significant challenge.

Homomorphic encryption has been proposed as a way to cryptographically protect computation results on untrusted hosts, making incorrect results arbitrarily difficult for malicious hosts to forge [10]. The disadvantage of these schemes is that homomorphic encryption currently only supports a very limited set of data operations, and therefore cannot yet be applied to a majority of computations.

Computations that are already parallelized and distributed across clouds can be probabilistically checked by simply replicating some or all of the sub-tasks and comparing the results for inconsistencies. Past works have therefore imbued MapReduce architectures with fault tolerance through massive replication with inconsistency resolution via majority voting [36] or distributed trust management [18]. In contrast, CloudCover focuses on certifying the significant class of computations that are not massively parallelized, including those that are inherently serial.

Remote attestation is an alternative to result checking that supplies evidence to a distrustful appraiser that an untrusted, remote target is running authorized software atop permissible hardware [37]. By assuring the integrity of the remote computing environment, its results can be trusted without

additional checking. Remote attestation solutions typically rely upon secure co-processors to attest hardware integrity [3], [4], and software monitors that relay cryptographically protected evidence of software integrity at load-time and in real-time as the remote computation progresses [5], [6], [7], [8].

However, the evidence exhibited by remote attestation solutions is not a proof. A knowledgeable, resourceful, or lucky adversary can potentially forge false evidence to corrupt the computing environment without detection (cf., [38], [39], [40]). This is partly because most software monitors rely upon code obfuscation [11], [12], [41] or blackbox security [42], which does not provide formal guarantees against reverse-engineering and corruption. An attacker with powerful reverse-engineering tools or inside knowledge of the obfuscation strategy can therefore potentially corrupt computations in ways that are not detectable by the appraiser. This invites an arms race in which attackers hone increasingly sophisticated analysis tools while defenders weave ever more complex obfuscations to bewilder them.

VI. Conclusion

CloudCover is a novel approach to SECaaS that allows Java computations executed in untrusted environments to be validated by commodity data processing clouds. Conceptually, it realizes proof-carrying computations as checkpoint chains. Generation and validation of such proofs is possible with relatively minor changes to existing Java software due to the insight that all Java programs already carry a notion of checkpoint equality encoded in their object-equality implementations. This serves as a computation integrity contract that can be validated by a trusted third party. The validation algorithm is massively parallelizable even when the original computation is largely serial, and can be spot-checked for even more efficient validation of probabilistic (yet quantifiable) integrity guarantees.

We demonstrate the feasibility of CloudCover's approach by implementing it and evaluating it on a real-world architecture: Hadoop MapReduce. Experimental results indicate that relatively few modifications to existing Java software are needed to add proof-carrying capabilities, and that validation services have a natural implementation as MapReduce jobs.

Our current implementation instruments Java programs with proof-carrying powers semi-manually. Future work should consider automated, binary-level approaches for doing so. In addition, our preliminary experiments consider only small-scale clouds, simple Java computations, and clients consisting of desktop machines. In the future, we intend to scale our work to larger scenarios and handheld devices, such as smart phones.

Applying our approach to other languages requires a means of generating checkpoints that can be compared for semantic equivalence. Managed, object-oriented, bytecode languages, such as Java, .NET, and ActionScript, facilitate such comparison through built-in class methods that decide object-equality. Native codes that realize checkpoints as process memory images admit such certification only if the images can be made insensitive to low-level hardware details that differ between the untrusted host and the trusted checker. Future work should investigate the feasibility of extending our work to such domains.

References

[1] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: What security properties can my mobile platform give me?" in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, 2012, pp. 159–178.

[2] A. Cooper and A. Martin, "Towards a secure, tamper-proof grid platform," in *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2006, pp. 373–380.

[3] Trusted Computing Group, "TCG attestation PTS protocol: Binding to TNC IF-M, version 1.0, revision 28," August 2011.

[4] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert, "Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform," in *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (TRUST)*, 2010, pp. 1–15.

[5] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 115–124.

[6] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 1–16.

[7] P. Falcarin, R. Scandariato, M. Baldi, and Y. Ofek, "Integrity checking in remote computation," in *Atti del XLIII Congresso Annuale (AICA)*, October 2005.

[8] M. B. MBarka, F. Krief, and O. Ly, "Entrusting remote software executed in an untrusted computation helper," in *Proceedings of the International Conference on Network and Service Security (N2S)*, 2009, pp. 1–5.

[9] T. Sander and C. F. Tschudin, "Towards mobile cryptography," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 1998, pp. 215–224.

[10] ——, "Protecting mobile agents against malicious hosts," in *Proceedings of Mobile Agents and Security*, G. Vigna, Ed., 1997, pp. 44–60.

[11] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Emperical Software Engineering*, pp. 1–35, 2013.

[12] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2010.

[13] Amazon, “Amazon elastic compute cloud,” http://aws.amazon.com/ec2.

[14] Microsoft, “Windows Azure,” http://www.windowsazure.com.

[15] Apache, “Hadoop,” http://hadoop.apache.org.

[16] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, June 2009.

[17] M. Gerla and D. Huang, Eds., *Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing*, SIGCOMM Special Interest Group on Data Communication. ACM, August 2012.

[18] S. M. Khan and K. W. Hamlen, “Hatman: Intra-cloud trust management for Hadoop,” in *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 494–501.

[19] W. Holden, “Mobile cloud applications & services: Monetising enterprise & consumer markets 2009–2014,” Juniper Research, Tech. Rep., January 2010.

[20] G. C. Necula and P. Lee, “Safe, untrusted agents using proof-carrying code,” in *Proceedings of Mobile Agents and Security*, 1998, pp. 61–91.

[21] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications of the ACM (CACM)*, vol. 51, no. 1, pp. 107–113, 2008.

[22] National Institute of Standards and Technology, “Secure hash standard,” April 1995, Federal Information Processing Standard.

[23] R. L. Rivest, “The MD5 message digest algorithm,” April 1992, Internet RFC 1321.

[24] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@home: An experiment in public-resource computing,” *Communications of the ACM (CACM)*, vol. 45, no. 11, pp. 56–61, 2002.

[25] Apache Commons, “Javaflow,” http://commons.apache.org/sandbox/javaflow.

[26] J. Bloch, *Effective Java*, 2nd ed. Sun Microsystems, 2008, ch. 3, Item 8: Obey the general contract when overriding `equals`, pp. 33–44.

[27] J. Du, N. Shah, and X. Gu, “Adaptive data-driven service integrity attestation for multi-tenant cloud systems,” in *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*, 2011, pp. 1–9.

[28] J. Du, W. Wei, X. Gu, and T. Yu, “RunTest: Assuring integrity of dataflow processing in cloud computing infrastructures,” in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010, pp. 293–304.

[29] S. M. Khan and K. W. Hamlen, “AnonymousCloud: A data ownership privacy provider framework in cloud computing,” in *Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, June 2012.

[30] Y. Lindell and B. Pinkas, “Secure multiparty computation for privacy-preserving data mining,” *Journal of Privacy and Confidentiality*, vol. 1, no. 1, pp. 59–98, 2009.

[31] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, “Airavat: Security and privacy for MapReduce,” in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010, pp. 297–312.

[32] J. Nakajima and M. Matsui, “Performance analysis and parallel implementation of dedicated hash functions,” in *Proceedings of the Annual International Conference on Theory and Application of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT)*, 2002, pp. 165–180.

[33] M. Blum and S. Kannan, “Designing programs that check their work,” in *Proceedings of the 21st ACM Symposium on Theory of Computing (STOC)*, 1989, pp. 86–97.

[34] G. F. Sullivan, D. S. Wilson, and G. M. Masson, “Certification of computational results,” *IEEE Transactions on Computers (TC)*, vol. 44, no. 7, pp. 833–847, 1995.

[35] G. Barthe, P. Buiras, and C. Kunz, “A functional framework for result checking,” in *Proceedings of the 9th International Symposium on Functional and Logic Programming*, 2010, pp. 72–86.

[36] M. Moca, G. C. Silaghi, and G. Fedak, “Distributed results checking for MapReduce in volunteer computing,” in *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPSW)*, 2011, pp. 1847–1854.

[37] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O’Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, “Principles of remote attestation,” in *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*, vol. 10, no. 2, 2011, pp. 63–81.

[38] S. Delaune, S. Kremer, M. D. Ryan, and G. Steel, “A formal analysis of authentication in the TPM,” in *Proceedings of the 7th International Conference on Formal Aspects of Security and Trust (FAST)*, 2010, pp. 111–125.

[39] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, “On the difficulty of software-based attestation of embedded devices,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009, pp. 400–409.

[40] A. Perrig and L. van Doorn, “Refutation of ‘On the difficulty of software-based attestation of embedded devices’,” http://sparrow.ece.cmu.edu/group/pub/perrig-vandoorn-refutation.pdf, 2010.

[41] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai, “Analysis of program obfuscation schemes with variable encoding technique,” *IEICE Transactions: Special Section on Cryptography and Information Security*, vol. 91-A, no. 1, pp. 316–329, January 2008.

[42] F. Hohl, “Time limited blackbox security: Protecting mobile agents from malicious hosts,” in *Mobile Agents and Security*, 1998, pp. 92–113.