Distributed Execution with Remote Audit

Fabian Monrose
New York University
New York, NY

Peter Wyckoff
Bellcore
Morristown, NJ

Aviel D. Rubin
AT&T Labs - Research
Florham Park, NJ

Abstract

Recently, there has been a rapidly expanding body of work with the vision of seamlessly integrating idle networked computers into virtual computing environments. This is enabled primarily by the success of research efforts promoting parallel and distributed computing on networks of workstations and the wide acceptance of Java. The proliferation of work in this area has provided new Internet-based infrastructures that harness the power of computing bases comprising hundreds of loosely-connected volunteered machines (i.e., hosts). While many of these systems have proposed the use of non-altruistic market-based schemes for promoting large-scale participation, mechanisms for ensuring that hosts participating in collaborative computing environments perform the work assigned to them have been largely ignored. This paper presents our implementation of one framework that layers a remote audit mechanism on top of an existing distributed computing model, and provides efficient methods for verifying, with a tunable level of certainty, whether a remote host performed the task it was assigned.

1 Introduction

The popularity of the World Wide Web and Java [5, 25] has promoted a new model for distributing code — down-loadable active content. This new model has lead to a proliferation of research in *Metacomputing*, that is, the transparent integration of multiple computer platforms, possibly across geographically dispersed areas, into a single virtual computing environment. The distinctive feature of this model is the exploitation of code mobility and mechanisms developed for parallel computing on loosely coupled machines [4].

Platforms that support Metacomputing, such as Atlas [7], Charlotte [8], Javelin [13], and ParaWeb [10] provide programming models that support the single-program-multiple-data (SPMD) computational model in heterogeneous computing environments. These platforms address the issues of scalability and fault tolerance, and for the most part, make efficient use of resources for supporting parallel computation within the Java framework. They support the execution of coarse-grained parallel computations on numerous anonymous machines on the Internet, and rely on Java's security architecture [40] to ensure safety to hosts. However, the issue of detecting misbehavior by hosts has been largely ignored.

Although it is possible to provide simple result checkers for specific SPMD-style applications such as matrix multiplication and factoring, providing result checkers for arbitrary SPMD programs is not easily accomplished. With the exception of redundant computation and voting schemes [35, 28], no additional mechanisms for verifying the work performed by remote hosts participating in coarse-grained parallel computations has yet been proposed.

Though the idea of computing with secrets in public is appealing for addressing this issue, detecting misbehavior through the use of assertion mechanisms that rely on inserting secret keys within the active content, is inadequate — since the content distributed across these platforms must be readable by a potentially large group of hosts, the keys are also readable, and thus can be easily recovered and reused [18]. Furthermore, the availability of efficient decompilation techniques [30, 39], even in light of code obfuscation [22], make these approaches unsatisfactory for deterring hosts from misbehaving.

Recently, Sanders and Tschudin [34, 33] proposed the use of computing with encrypted functions [32, 1] to address the issues pertaining to publicly readable executable content. While the idea of computing with encrypted functions is appealing, numerous obstacles, both theoretical and practical, still need to be overcome before its application to mobile cryptography can be fully realized. An examination of some of the problems associated with computing with encrypted function, as it applies to protecting against malicious hosts, is presented in Section 3.

We have designed and implemented an audit mechanism to detect misbehavior by hosts participating in Metacomputing environments. In particular, we show that a host claiming to perform work that it in fact did not do will be caught with high probability. Our implementation is Java-specific and makes use of the Java execution environment. However, the same principles and techniques could also be applied to other environments such as Safe-Tcl [29, 41].

Section 2 introduces a high-level overview of our model for verifying the work performed by remote hosts. Some preliminary work on mobile agent security, as well as other approaches within different contexts, but with similar desirable goals as ours, is presented in Section 3. Our design and implementation are presented in Section 4. To illustrate proof of concept we present an example application and provide some empirical performance results in Section 5. An analysis of the probability associated with catching *cheating hosts* is presented in Section 6, and limitations of the current design and implementation are examined in Section 7. We conclude with directions for future work.

2 Overview

A high-level overview of our approach to validating whether hosts participating in coarse-grain, SPMD-style parallel computations, perform the work assigned is now presented. We consider the model where a "manager" has a large parallel computation to be performed and there exists a collection of "workers" willing to participate in the computation for some payment. The payment mechanism is outside the scope of this paper. We assume that there is some arrangement between the manager and the workers, and payment is facilitated by some other mechanism (perhaps NetCash [27] or NetBill [36]). Our goal is to audit workers so that the manager can detect misbehavior.

The following terms are used throughout the paper:

- *Computational components* are tasks to be performed and are characterized by executable content containing code and a set of variables $\{M\}$ whose values are to be computed. The running time of a computational component is denoted as T.

There are three entities which participate in our computing model: *managers*, *workers* and *verifiers*:

- *Managers* are processes requiring computing resources.

- *Workers* are processes offering computing resources and represent execution environments. They host computational components and provide support for execution. In its simplest form a worker is a Java-enabled Web browser. The following subtle distinction is made between *cheating* and *malicious* workers:

  – A *cheating* worker is a worker that does not execute a computational component in its entirety; a cheating worker may execute any part(s) of the component that it chooses. For instance, a cheating worker may execute only the first half of a computational component and return partially computed values for $\{M\}$.

    More formally, let $0 \leq P < 1$, then a cheating worker devotes PT execution time to a component and returns an incorrect set of values for $\{M\}$. A cheating worker may use part of the execution time devoted to a component for techniques such as decompilation.

  – A *malicious* worker is a worker which attempts to deliberately subvert an ongoing computation by providing an incorrect set of values for $\{M\}$, **even** at the cost of execution time greater than T.

    More formally, let $P \geq 0$, then a malicious worker devotes PT execution time to a computational component.

  The practical difference between cheating and malicious workers is that a cheating worker's goal is to minimize resource expenditure (possibly to maximize profit), whereas a malicious worker's goal is to subvert the computation. This paper addresses the issue of detecting misbehavior by cheating hosts only.

- *Verifiers* are processes which confirm the remote execution of a computational component. They represent processes working on behalf of *managers*.

Our current verification system is geared towards catching *cheating* workers and is based on instrumenting computational components, with the aid of compiler techniques, to produce proofs of execution. Once a worker completes the work it was assigned, a proof of execution is sent to the verifier, which examines parts of the proof to check (with some desired level of confidence) whether the component was executed correctly. Since our goal is to provide a framework upon which Metacomputing infrastructures can reside, transparency is important in our design.

With regards to the proofs which are generated, there are two important issues that must be addressed. First, the proof must cover the execution of the entire component, not just parts of it. Otherwise a cheating worker could modify the component to execute only the parts that are checkable. Second, since a cheating working will know the exact protocol for producing and verifying the proof *a priori*, the verification protocol cannot present any implications about the security of the system.

In our model, a proof consists of the state of the computational component at various points in its execution. In essence, a computational component is transformed into small subcomponents whose post execution state is the input to the next subcomponent. Each post execution state consists of private data and information from the call stack, which we call a *trace*.

Verification consists of repeatedly picking one of the traces, i, at random, and executing the computational component from the point of the execution that produced that state, to the point in the execution which triggers the next trace. If the state of the call stack in the verifier's virtual machine after executing up to the next state point is the same as the $i+1$ state point that the worker submitted, the worker produced the correct state transition between those two traces.

Figure 1 depicts a high-level view of our system architecture illustrating the *compile-time*, the *run-time*, and the *verification* modules. We describe each of these modules in more detail in Sections 4.1, 4.2.1, and 4.2.2, respectively. As the figure shows, computational components are automatically instrumented (via the compile-time module) with the code needed to produce checkable state points. This augmented component is sent to the worker which executes it, producing the proof of execution and the result(s). Once completed, the worker sends the verifier the results and the sequence of state points that comprise the proof. The verifier then samples a random set of these state points, checking the correctness of the transitions between the state points.

Intuitively, if a computational component is transformed into $N-1$ pieces, then there are N traces (the initialization variables for the component form the extra trace). If the proof contains L incorrect traces, and the verifier chooses to verify K of the traces, then the probability that the verifier finds one of the incorrect traces is greater than $1-((N-L)/N)^K$. In Section 4.1 we describe how computational components are transformed in such a way that producing intermediate states requires that the components are executed.

3 Related work

Although the research on *program checkers*, probabilistic interactive proofs and their spin-offs is quite different from that presented in this paper, it is nonetheless particularly important to our work. The theory and ideas put forth in that research provide a sound formal background for work in this area. The work of Blum *et al.* [9], on program checking is concerned with verifying that a given program returns a correct answer on a given input rather than on all inputs, and, with the adaption of interactive proof systems [19], provides the basic intuition for our work.

Recently, research on validating remote execution has been conducted within the mobile agent framework. Although the distributed SPMD-style computing model which we are interested in is quite different from the focus of classical mobile agent systems, there are some similar desirable security goals in both models. Mobile agents are defined as processes which can autonomously migrate to new hosts while executing their task. Approaches to solve the problem of malicious hosts within these environments have been proposed by Hohl [22], Sanders and Tschudin [34, 33], and Vigna [38].

To address the problem of misbehavior by hosts, Hohl [22] proposes a combination of code mess-up (i.e., obfuscation) and placing time critical restrictions on the mobile code. These restrictions are encapsulated as part of the code, with the intention that nodes which host agents comply with the restrictions placed on the code. Therefore, an agent is invalidated as it migrates from host to host if it is not executed within a certain time window. The timing constraints of the execution window are based on empirical results for the expected de-mangling time of the code (typically on the order of seconds). The goal is to restrict the lifetime of the agent by limiting the execution window to a fraction of the time it would take to de-mangle, understand, and change the code in a deliberate manner. Since the de-mangling time is much less than that of typical coarse-grain parallel computations, this approach is not applicable to our goals.

The approach of Sanders *et al.* [34, 33] for protecting mobile agents against malicious hosts is based on the idea of computing with encrypted functions [32, 1]. The intuition is as follows: given a function $f(x)$ which can be represented by a program P, rather than distribute P, the encrypted function $E(P)$ is distributed instead. The remote host then executes $y = E(P)(x)$ and transmits y back to the originating host, at which point y is decrypted to retrieve $f(x)$. P is expressed as a polynomial or rational function with certain mathe-

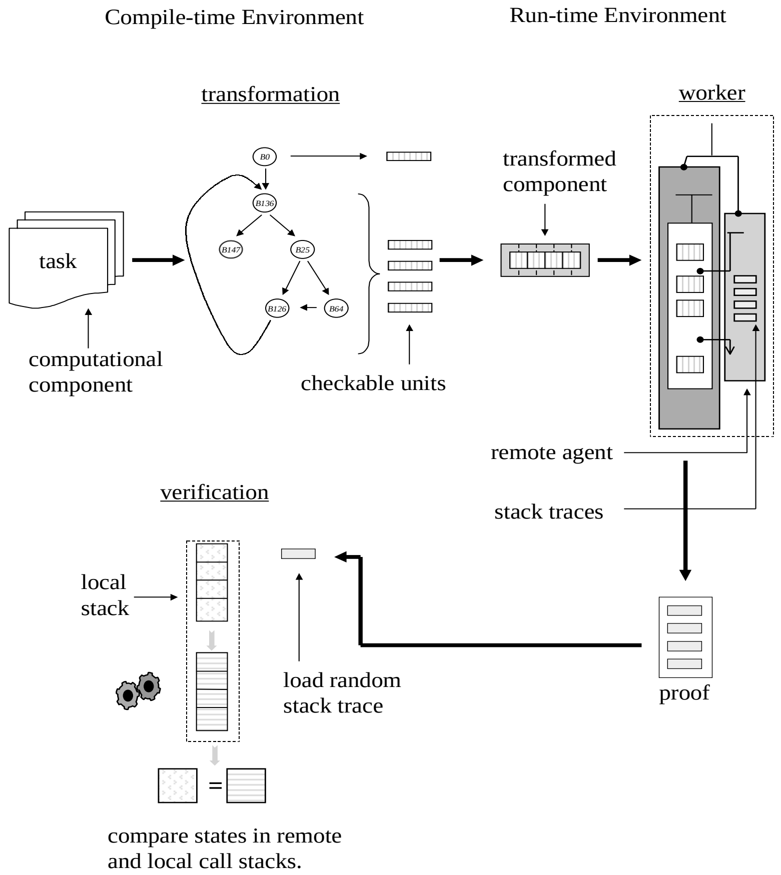

Figure 1: High-level overview of the system design illustrating the compile-time, run-time and verification environments. The compile-time module automatically instruments computational components with the code needed to generate state points. These augmented components are sent to the worker which then executes them in its local interpreter. A remote agent, responsible for handling each component, creates a proof of execution and the results (final state). Once the worker completes the task assigned, the remote agent sends the verifier the results and the sequence of state points that comprise the proof. The verifier then samples a few of these state points checking the correctness of the transitions between the states.

matical properties.

The caveat is that computing with encrypted functions for general programs is an open problem and many strong arguments which largely discourage the idea of computing with encrypted data have already been put forth (see [2, 11, 32]). Furthermore, efficient encryption schemes with the desired homomorphic properties (see [34]) for arbitrary functions are not known and since P is executed in its encrypted form, exactly how this approach will be realized in practice is unclear.

The solution put forth by Vigna [38] is to verify program execution by tracing the operations performed by a mobile agent during its lifetime. In Vigna's model, a roaming agent is composed of code and a state that is determined, at some specified point, by code execution. Every statement executed by the agent is logged and before the agent migrates to a new host, a digitally signed message is sent to the originating host containing a checksum of the program's final state before migrating, a checksum of the execution trace, and a unique identifier. Traces are logs of the operations performed by the agent during its lifetime. The extended form of the execution trace is stored for some limited period of time.

In the event that tampering of an agent's code is suspected, tampering can be proven by verifying the agent program against a supposed history of its execution i.e., simulating the *entire* program locally. The system is implemented in a restricted subset of Safe-Tcl [29] and tracing is accomplished by adding new instructions to the language. It is assumed that the code is static, therefore performance enhancements such as just-in-time compilation are not possible. The main caveat, however, is that the complexity of the validation process is linear with respect to the size of the execution traces, which can be significant. No implementation or empirical results based on system performance were presented.

4 Architecture

Our design and implementation can be roughly separated into a compile-time module and a run-time module. The compile-time module automatically instruments computational components with code to interact with the remote debugger and the run-time module performs the verification. We discuss the design and implementation of these two modules in the following sections.

4.1 Compile-time module

In Section 2 we stated that the remote agent periodically saves information from the call stacks of executing computational components based on conditions specified by the manager at run-time. The mechanism by which these conditions are selected is now elaborated on.

As mentioned earlier, the compile-time module transforms a computational component into *checkable units*, some of which the verifier runs from start to finish on its local machine, to check that the proof of execution provided by a worker corresponds to a correct execution of those units. A proof is represented by a sequence of traces, $t_0, t_1, \ldots, t_n$.

We now define some terms that are useful for describing the design and implementation of the system. A block of code is defined as *execution capturable* if, with high probability, the state of the job after its execution cannot be produced correctly by means other than executing the block. For example, a subcomputation that generates a single bit of information is not execution capturable because an adversary may be able to guess the outcome and generate a correct state with 0.5 probability.

During the transformation of a component, some parts of the program generate single units while loops correspond to multiple units. We must ensure that the multiple units that are generated within a loop have some property that makes them unique. Otherwise, the traces that correspond to their executions might be interchangeable. We define the unit that is generated from a part of the program that does not have a loop boundary (i.e., it is not contained in a loop but it may contain loops) as *distinct.* We define the units generated from a loop where the values computed in the loop depend on *at least* the loop index and the state of the program before the loop as *uniquely identifiable.*

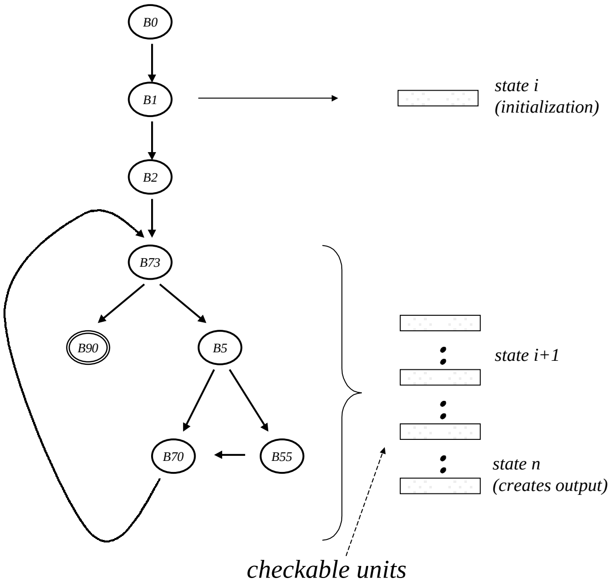

Figure 2: High-level overview of the transformation of the computational component for discrete log example (from the Appendix) into checkable units using its CFG. Given this CFG and its associated component, the compile-time module transforms the code by instrumenting it to trigger breakpoints at specific points. At run-time, when breakpoints are triggered, the *remote agent* associated with the current computational component, collects internal state information on the call stack for the current thread. These traces are used later during verification.

We require that the units that our transformation creates are either distinct or uniquely identifiable. Otherwise, the trace transitions that are defined by the units may not necessarily correspond to only one unit. If that is the case, a cut-and-paste style attack is possible. To guarantee that these criteria are met, we require that at least one variable within a computational component captures past history. That is, there is at least one object (for example, an array) whose values correspond to the computation, and therefore, reflect the computation between traces. This ensures that successive traces include some actual state that was computed.

4.1.1 Code transformation

Our compile-time module uses the control flow graph (CFG) [3] for a given computational component, $\mathcal{P}$, to generate its corresponding checkable units. Intuitively, the key to the instrumentation process is: (1) the execution of the checkable units is captured in the traces and (2) each trace corresponds to the output of exactly one checkable unit. The CFG is produced by recovering high-level structure from the Java bytecode of the untransformed computational component using our own back-end to the Java compiler. Java is used as our platform for implementation because of its portability and heterogeneity. The fixed format of Java class files makes it ideal for recovering high-level structure [30] needed to build the CFGs.

In a manner similar to the work on compiler-assisted checkpointing [24], given the CFG our compile-time module adds the target nodes, i, of any back-edges to a set of breakpoint candidates $\mathcal{S}$. Additionally, every exit node in $\mathcal{P}$ is added to $\mathcal{S}$. For all $i \in S$ a new object $action_{S_i}$, consisting of a single boolean method, is created. This method specifies when a computational component's state should be saved and triggers the remote agent whenever this condition is true. $Action_{S_i}$ is created based on simple data analysis techniques and can be overwritten by the programmer. Work on optimizing Java compilers [15, 12] utilize significantly more powerful analysis techniques than those we currently use, and we hope to borrow some design and implementation from their research to enhance our functionality.

All objects are loaded on-demand across the network by the remote agent, transparently to the worker. Figure 2 depicts a transformation of a computational component into checkable units, each of whose "output" is the input to the next unit. If the entire component were not covered by checkable units, a malicious worker, would be free to corrupt the uncovered parts without any danger of being caught.

Once $\mathcal{S}$ and $action_{S_i}$ are created, for all $i \in S$, the compile-time module inserts calls to $action_{S_i}$ at the location specified by S_i. The manager saves some metadata including $\mathcal{S}$ in persistent storage, and at run-time, the remote agent responsible for collecting traces for the executing component is instructed (via a callback mechanism) where to set breakpoints. These breakpoints alert the remote agent when state must be saved. A trace represents the current state of a computation, including the value of all instance variables within scope, at the point in time when the breakpoint was triggered. Type information is not saved since the verifier knows the type of each object that should appear on its local stack at any given point. We show that even if the condition under which state is saved is known *a priori* it is of no advantage to a cheating worker.

4.2 Run-time module

The runtime component of our system must be interoperable with existing Metacomputing environments. Otherwise, remote monitoring and verification services would not be transparent to the programmer who would need to write stack marshaling and unmarshaling routines for every breakpoint in a computational component. To avoid this, we chose the most common Metacomputing environment implementation language, Java, as our runtime system's implementation platform. While a number of inherent security flaws have been outlined [26, 16, 23], we provide a framework that assumes that these concerns will be resolved as Java matures; however, our verification technique will not be compromised even if a worker tampers with its JVM.

4.2.1 Remote monitoring

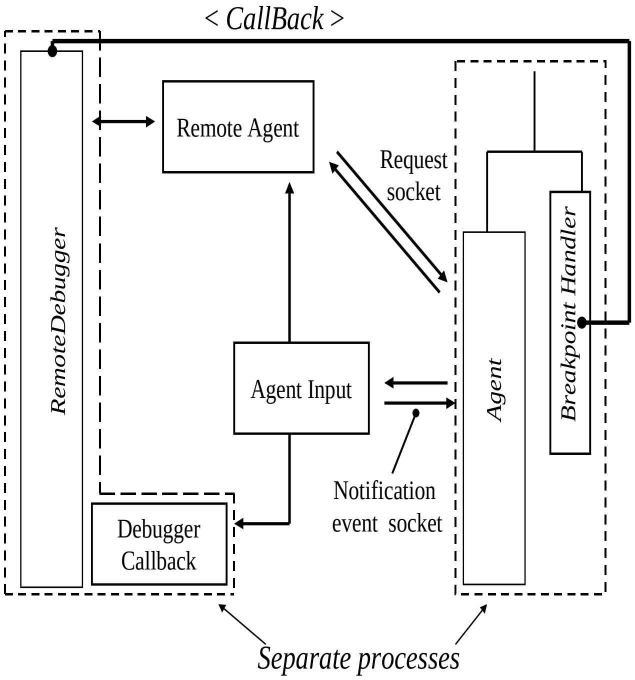

Figure 3: Overview of the remote debugging process in Java. The architecture consists of a debugger client, a debugger server, and a TCP/IP socket based communication protocol. The API allows us to connect and communicate with the JVM of the worker and obtain low-level information about the internal states of executing threads. A *remote agent* handles the actions initiated by the client, while a separate thread handles notification events from the server. The communication and notification events are transparent to the worker.

Our implementation of remote auditing is based primarily on Java's Debugger API. This API is designed around the concept of *remote debugging* which allows the debugger and its target (i.e., debugee) to execute on separate machines. The Java model for remote debugging is depicted in Figure 3.

The model consists of a debugger client, a debugger server, and a TCP/IP socket-based communication protocol. The API allows a process to connect and communicate with the JVM [25] of the target and obtain low-level information about the internal states of executing threads. All communication to and from the debugger server is performed over two socket connections created when the RemoteDebugger class is instantiated. A *remote agent* thread handles synchronous actions initiated by the client and communicated over one of the sockets, while an *agent input* thread handles notification events from the server. These events are asynchronous to the client and are implemented via a callback mechanism. When the Agent Input thread receives a message from the debugger server over the socket, the message is interpreted and the appropriate method within the registered callback is invoked.

For ease of implementation and to reduce communication overhead, the remote agent and the debugger server reside on the target worker machine. This decision has little significance with regards to cheating workers, but does influence the trust relationship between managers and workers. While the current architecture allows for functionality provided by these modules to be easily migrated to the *manager*, network latency overhead would degrade system performance.

The remote agent's task of capturing the state of the computational object in a serialized form suitable for transmission is accomplished through the use of an object serialization mechanism known as *pickling* [31]. The complementary process of *unpickling* is used by the verifier for initializing frames in its local call stack to those transmitted by the remote agent.

4.2.2 Verification

The verifier is depicted in Figure 4. A traditional challenge-response style protocol is used by the verifier to validate the proofs of execution submitted by a remote agent working on behalf of a worker. The protocol proceeds as follows.

The remote agent *commits* to each trace $(t_i, \ldots, t_n)$ by submitting a cryptographic *hash* (SHA-1 in our implementation) of each trace collected. The verifier then randomly chooses $r \in n$ and challenges the remote agent to present the trace for t_r. The remote agent responds with the requested trace, which is cross-checked by the verifier.

To validate trace t_{r+1}, the verifier reconstructs the image of t_r on its stack (checking the values of loop indices for correctness), executes a local copy of $\mathcal{P}$ from the program counter immediately following the point which triggered the commitment of state t_r, until state t_{r+1}. The local execution represents running a fraction of the computational component, which corresponds to the amount of work performed between successive traces. The verifier compares the hash of its call stack with $hash(t_{r+1})$, previously committed by the remote agent. If the hashes are equal, t_r is accepted, otherwise it is rejected. The procedure may be repeated for any number of the remaining traces—depending on the level of assurance required by the manager. We show that the run-time associated with validating traces is significantly less than the time it would take if the manager executed the entire component locally (i.e., spot-checking [35]).

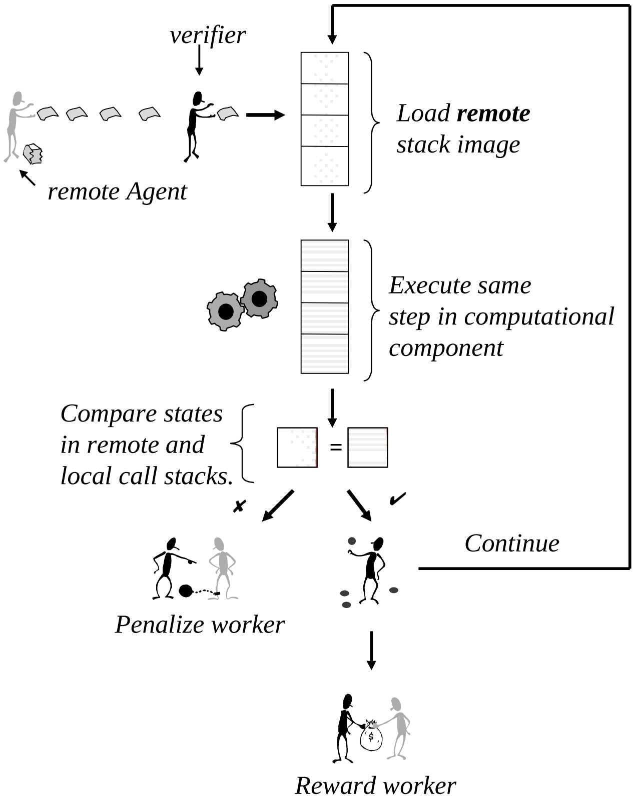
verifier
remote Agent
Load **remote** stack image
Execute same step in computational component
Compare states in remote and local call stacks.
=
✘
✔
Continue
Penalize worker
$
Reward worker

Figure 4: The verification process. The verifier selects a trace, i, at random, and reconstructs its state on its local call stack. The computational component is executed until state $i + 1$ is reached and the final state is compared with that in trace $i + 1$ submitted by the remote agent. If the two states are equal, then the trace is accepted and the worker is rewarded. Otherwise, the trace is rejected. At that point, it might be prudent to penalize the worker or the organization which it represents.

5 Example: computing a discrete logarithm

As proof of concept, we consider an example where a party, Alice, uses the same secret exponent (x) for multiple Diffie-Hellman [17] key exchanges. Mallot, Alice's adversary, has knowledge of this fact, and wants to impersonate Alice. However, he has only limited resources and exhaustively searching for the secret exponent is the only method available. Mallot also knows that Alice uses the relatively small public parameters, $g = 17$ and $n = 9311$. This particular example was chosen because it can be represented by a SPMD-style application that can be efficiently supported by current Metacomputing infrastructures.

To expedite the search for the secret exponent, Mallot creates a SPMD-style *computational component* utilizing a striping technique, and submits the component to a manager, thereafter agreeing on some form of payment for workers. The Java code for the component is given in the Appendix. While it is trivial for Mallot to check that exponents returned by workers are in fact the secret exponent x he is seeking, payment must also be made to workers who pruned the search space requested of them, but did not yield any candidate exponent(s). Furthermore, cheating workers must not receive any payment. For these reasons, the verification service provided by the manager is essential to the task at hand. This example is used as a basis for the performance results presented in the following section.

5.1 Performance

Performance results based on the discrete log example are now presented. Experiments were conducted within the Distributed Systems Laboratory at New York University on twelve 200Mhz Pentium Pro machines running Linux, each with 64MB of main memory. The machines were interconnected by a 100 Mbit/sec Ethernet. All entities resided on separate machines.

Our first set of experiments examined the performance and overhead associated with the remote monitoring environment. In these experiments, the elapsed execution time for exhaustively searching in parallel for the secret exponent, on a series of machines, was compared to the sequential execution of the same task in a non-monitored environment (i.e., the execution was performed on one machine outside the remote monitoring environment).

The sequential execution time was 277 seconds. The elapsed time for performing the search within the remote monitoring environment with only one worker participating in the computation, and the remote agent logging state at *every* step during the computation, was approximately 1452 seconds. This represents a slowdown of more than 5 times over the non-monitored, sequential execution. However, when the remote agent was instructed to save the state at 10% intervals (i.e., the amount of work performed between successive traces corresponds to 10% of the total work to be performed) the elapsed time was 458 *seconds*. This represents a slowdown of only 1.65. The overhead diminishes rapidly as more workers are added to the computation (see Figure 5). In fact, speedups over the sequential execution are achieved with 5 or more workers, regardless of how much state is saved.

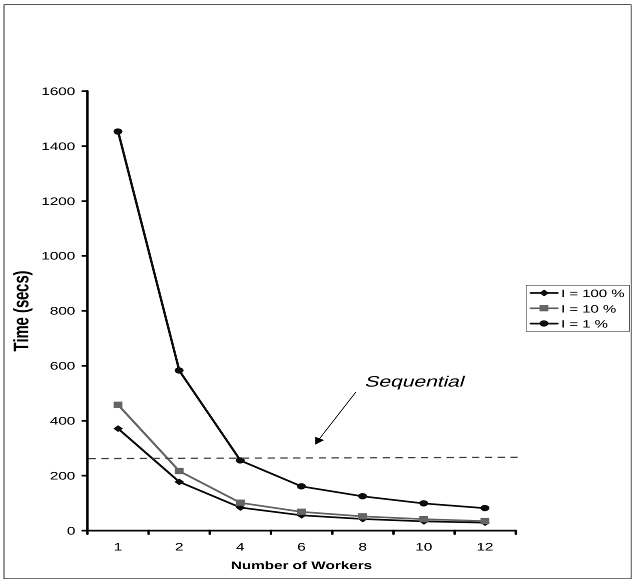

Figure 5: Performance of the auditing environment. I represents the trace interval, that is, the amount of work performed between successive traces. As more workers are added to the computation, speedups over the sequential execution are realized. The relatively low overhead imposed on workers in these distributed settings makes such an auditing environment beneficial to managers, as it provides assurances with regard to a worker's execution of a computational component.

A second set of experiments was conducted to evaluate the performance of the verification system. The experiments involved starting one *manager* that spawned $n = 1, 2, \ldots, 12$ *workers* in successive experiments. For each worker participating in the discrete log computation the verifier picked 10% of the submitted traces at random (in addition to the initialization and final states) and checked the correctness of each state transition against its locally reconstructed image. The results are depicted in Figure 6 — the elapsed time for

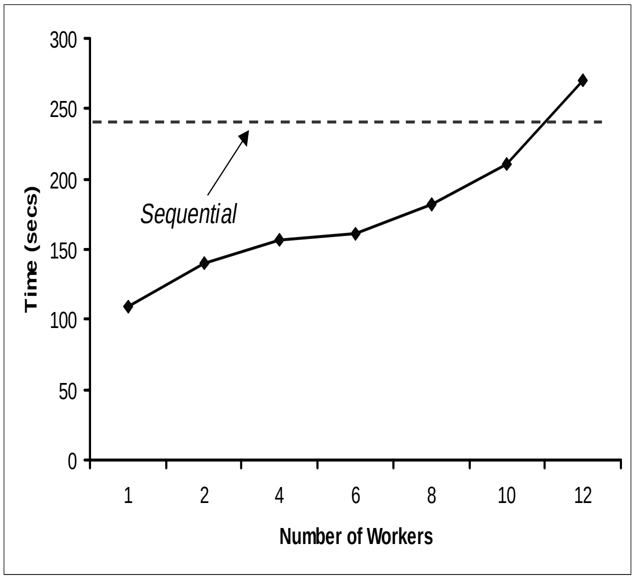

Figure 6: Performance of the verification system. For each worker participating in the computation the verifier picked $I = 10\%$ of the submitted traces at random (in addition to the initialization and final states) and checked the correctness of each state transition against its locally reconstructed image. These results show that our approach represents an efficient method for verifying a worker's supposed execution of a computational component.

starting the remote n workers is not reflected here.

Verifying 10% of the traces of one worker required significantly less time (109.30 seconds) than executing the computational component sequentially. As more workers were added, and hence more traces needed to be verified, the performance of the verification system steadily degraded due to the overhead associated with receiving multiple proofs of execution from remote agents, simultaneously. Nonetheless, these results show that our approach represents an efficient method for catching cheating workers in a distributed environment.

To evaluate where possible improvements in performance could be achieved, we measured the overhead of the various monitoring and verification activities. It was observed that while there was room for enhancement within the verification component whose primary function is in reconstructing and comparing stack states, a significant portion of the overhead, 37.2%, is associated with context switching, especially in time spent pausing and resuming threads within the Java debugging environment. The same is true for the remote monitoring environment.

Although we suspect that performance enhancements within the classes that provide the functionality for the Java debugging environment are unlikely (it has been unsupported since JDK 1.02), implementing a verification system which does not rely on the functionality provided by the remote debugging facilities can be accomplished with little difficulty. We are currently pursuing this approach.

6 Probability of catching a cheating worker

We now present an analysis of the probability associated with catching a cheating worker. Consider the case where the adversary is paid to execute 100 jobs, each of which is transformed into 100 units. Assume that the adversary wants to (or only has the resources to) execute 95% of the total work neccessary to complete the 100 jobs in their entirety. We compare the minimum probability of catching such a cheating worker in our system with spot checking [35]. The minimum probability of catching the worker is computed by finding the probability of catching a worker that is employing an optimal cheating strategy.

For our scheme the probability of catching a cheating worker is multiplicative, and therefore, the optimal cheating strategy is to execute 95 of the units for each job. If the verifier checks a single unit per job, the verifier has a .05 probability on each job of catching the adversary cheating. Over the 100 jobs the probability of detecting misbehavior is approximately .994 $\left(1 - (1 - .05)^{100}\right)$. This probability is associated with the verifier checking only 1% of each job (actually, the probability is slightly higher because when the transition from r_i to r_{i+1} is checked for $1 \leq i \leq n - 1$, if either of r_i and r_{i+1} is incorrect, the verification will catch the error).

If instead as in spot checking, the verifier were to periodically execute a job locally, in its entirety, then the optimal cheating strategy is to execute 95 of the jobs in their entirety, not expending any effort for the other 5 jobs. Then the probability of detecting misbehavior is only .05.

7 Limitations

Our approach to remote auditing of hosts is well suited when the computational components are SPMD-style programs. While our work addresses an important problem within the distributed and parallel research community, our techniques are not directly transferable to other fields, such as classical mobile computing. Some limitations of our current design and implementation, which will be addressed as this research matures, are now outlined:

- The verification techniques described herein assume the existence of good pseudo random number generators; otherwise, if a worker can guess with high probability which traces are examined by the verifier *a priori*, this knowledge significantly decreases its chances of getting caught. Given the current random number generators in Java, such attacks on our system are possible. However,this attack can be subverted through the use of hardware number generators.

- Our model assumes that computational components can be transformed into checkable units that have similar execution times. However, there are certainly applications where this is not the case. Therefore, in choosing which traces to check, it is necessary that the verifier weigh the traces with estimates of their relative execution time. Research on designing efficient weighting algorithms is currently underway.

- Programs which rely on external input or random data can not be handled per se by the method outlined here — however programmer assistance using pragmas or more sophisticated compiler assistance can be used to handled external input and random data.

- Performance enhancements within the remote debugging interface are unlikely as it has been unsupported since JDK Release 1.02. In addition, the functionality provided by the remote debugging environment requires access to privileged system targets beyond the normal confines of the Java sandbox. However, in environments where the principals participating in distributed computations are represented by Java applications, as is the case with JavaPVM [14], object signing is not an issue. Although the security policies which define the behavior of local versus remote code are being designed to be more flexible and extensible [21, 20], if one intends to support this framework entirely within Java enabled browsers then issues pertaining to object signing, need to be taken into consideration.

8 Conclusion and future work

In this paper, we presented a technique for auditing the execution of SPMD tasks in a distributed environment based on transforming tasks into checkable units. Our work is geared towards auditing workers participating in coarse-grain parallel computations, on numerous anonymous machines on the Internet. We presented a framework based on compile- and run-time modules which include a verifier capable of catching cheating workers with high probability, based on checking a small number of the state transitions between these units. Intuitively, this scheme has the advantage that for a worker to cheat, it must successfully corrupt at least one of the checkable units, which is much more difficult than corrupting an entire computation by simply returning an erroneous result.

For our auditing technique to apply, we are restricted to a set of programs that are execution capturable and have uniquely identifiable units generated within loops. Although we show the probability of catching cheating workers given a proper transformation of a computation, we do not show what the probabilities are for programs not in our restricted set—which may be the case if we apply our techniques to non-SPMD programs.

Previous practical work in this area has relied on unreasonably strong assumptions about the adversarial model of cheating or malicious workers. For example, systems which rely on embedding keys within tasks and returning some function of these keys as assertions that the task was correctly executed, assume that the code will not be decompiled. The availability of efficient decompilation techniques has rendered these approaches inadequate. Other approaches have proposed replicating the tasks on domain disjoint machines, and occasionally checking task outputs using a trusted machine. Our approach is in some sense orthogonal to these heuristics and they can therefore be applied to our system; however, the advantage would be minimal. Our contribution is in providing a more efficient execution environment for detecting misbehavior by hosts participating in Metacomputing infrastructures.

A promising approach to improving the guarantees provided by our system involves the application of techniques such as those of Arora and Safra [6] for devising probabilistically checkable proofs (PCP) to the verification of the transitions between checkable units. We plan to pursue this approach by creating redundancy in the traces so that checking a reasonably small number of the augmented pieces of the proof of execution presented by a worker would find errors in the worker's computation with high probability. We are also considering incorporating more complex binary predicates instead of only stack equalities as checks and hope to achieve these goals without too high a cost in performance.

9 Acknowledgments

The authors would like to thank Saugata Basu, Matt Franklin, and Laxmi Parida for stimulating discussions about this research. We also thank the anonymous referees, Geritt Bleumer, Tom Bowen, Ian Jermyn, and Zvi Kedem for their suggestions on improving earlier drafts of the paper.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

APPENDIX

The Java code which implements the discrete log example discussed in Section 5 is now given. The code is augmented with labels that represent the nodes in the control flow graph (CFG) of Figure 2. These labels represent the actual offsets within the Java bytecode of the corresponding *class file*.

```
// Discrete log example with relatively small integers
import java.math.*;
import java.lang.*;
import java.util.*;
public class DLSearch extends Thread {
    private int whoami, window, result=0;
    private final int prime = 9311, g=17, z = 5653;
    private int[] zp; /* variable which captures past computations */

B0: public static void main( String args[] ){
      DLSearch d =
      new DLSearch(Integer.parseInt(args[0]), Integer.parseInt(args[1]));
    }
B1: DLSearch( int who, int slice) {
      this.whoami = who;
      this.window = slice;
      this.setPriority( 6 );
      zp = new int[prime/window];
      this.start();
    }
    /* divide work based on striping technique */
B2: public void run() {
      int i = 0;
B73:  while ( (whoami+(i*window)) < prime ) {
B5:     zp[i] = ( ( (BigInteger.valueOf(g)).pow(whoami+(i*window) ) ).
            mod(BigInteger.valueOf( prime)) ).intValue();
        if ( zp[i] == this.z ) {
B55:      result=whoami+(i*window);
        }
B70:  i++;
      }
B90:  System.exit(0);
    }
}
```

Figure 7: Labels *B0 . . . BN* correspond to the labels in the control flow graph (CFG) in Figure 2. High level structure is recovered from the Java bytecode; labels represent the actual offsets within the Java bytecode of the corresponding DLSearch *class file*.

References

[1] M. Abadi and J. Feigenbaum. Secure Circuit Evaluation. *Journal of Cryptography*, 2(1):1–12, 1990.

[2] N. Ahituv, Y. Lapid, and S. Neumann. Processing Encrypted Data. *Communications of the ACM*, 30(9):777–780, 1987.

[3] Aho, Sethi, and Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb 1995.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[6] S. Arora and S. Safra. Probabilistic Checking of Proofs. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 2–13. IEEE Computer Society Press, October 1992.

[7] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[8] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[9] M. Blum and S. Kannan. Programs That Check Their Work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.

[10] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[11] Ernest F. Brickell and Yacov Yacobi. On Privacy Homomorphisms (Extended Abstract). *Eurocrypt*, 1987. Abstracts: IV-7-IV-14.

[12] Z. Budlimic and K. Kennedy. Optimizing Java: Theory and Practice. *Concurrency - Practice and Experience*, 9(6):445–464, June 1997.

[13] P. Capello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *ACM Workshop on Java for Science and Engineering Computation*, 1997.

[14] Center for Human-Machine System Research. JavaPVM: The Java to PVM Interface, August 1997.

[15] M. Cierniak and W. Li. Optimizing Java Bytecodes. *Concurrency - Practice and Experience*, 9(6):427–444, June 1997.

[16] Drew Dean, Ed Felten, and Dan Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 190–200, 1996.

[17] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov. 1976.

[18] William Farmer, Joshua Guttman, and Vipin Swarup. Security for Mobile Agents: Issues and Requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, 1996.

[19] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *17th ACM Symposium on Theory of Computing*, pages 291–304, 1985.

[20] Li Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Sysposium on Internet Technologies and Systems*, December 1997.

[21] Li Gong and Roland Schemers. Implementing Protection Domains in the Java Development Kit 1.2. In *Proc. Internet Society Symposium on Network and Distributed System Security*, pages 125–134, March 1998.

[22] Fritz Hohl. An Approach to Solve the Problem of Malicious Hosts. Technical Report TR-1997-03, Universität Stuttgart, Fakultät Informatik, Germany, March 1997.

[23] Mark D. Ladue. Java Insecurity. *Computer Society*, Spring 1997.

[24] C. J. Li and W. K. Fuchs. CATCH: Compiler-Assisted Techniques for Checkpointing. In *20th*

International Symposium on Fault Tolerant Computing, pages 74–81, 1990.

[25] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1996.

[26] Gary McGraw and Edward Felten. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons. New York. New York, 1996.

[27] G. Medvinsky and B. C. Neuman. NetCash: A Design for Practical Electronic Currency on the Internet. In *Proceedings of the First ACM Conference on Computer and Communications Security*, Nov. 1993.

[28] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic Support for Fault-Tolerant Distributed Computing. In *Seventh ACM SIGOPS European Workshop*, pages 109–114, Connemara, Ireland, 1996.

[29] J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. Technical report, Sun Microsystems, Nov 1996.

[30] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*, pages 185–197, June 1997.

[31] R. Riggs, J. Waldo, and A. Wollrath. Pickling State in Java. In *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems*, pages 241–250, Toronto, June 1996.

[32] R. Rivest, L. Adleman, and M. Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation*, pages 169–177, 1978.

[33] T. Sanders and C. Tschudin. Protecting Mobile Agents Against Malicious Hosts. *Mobile Agent Security*, 1997.

[34] T. Sanders and C. Tschudin. Toward Mobile Cryptography. *IEEE Symposium on Security and Privacy*, 1998.

[35] Luis F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. In *Proceedings of the 2nd International Conference of World-Wide Computing and its Applications*, 1998.

[36] M. Sirbu and J. D. Tygar. Netbill: An Internet Commerce System Optimized for Network Delivered Service. *IEEE Personal Communications*, pages 34–39, Aug. 1995.

[37] Glenn Vanderburg. *Tricks of the Java Programming Gurus*. Sams Net, 1997.

[38] G. Vigna. Protecting Mobile Agents through Tracing. In *Proceedings of the 3rd Workshop on Mobile Object Systems*, June 1997.

[39] Hanpeter van Vliet. The Mocha Decompiler, 1996.

[40] Frank Yellin. Low Level Security in Java. In *Proceedings of the 4th International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[41] X. N. Zhang. Secure Code Distribution. *Computer*, 30(6):76–79, June 1997.