How To Withstand Mobile Virus Attacks

EXTENDED ABSTRACT

Rafail Ostrovsky* Moti Yung†

Abstract

We initiate a study of distributed adversarial model of computation in which faults are non-stationary and can move through the network, analogous to a spread of a virus or a worm. We show how local computations (at each processor) and global computations can be made *robust* using a constant factor resilience and a polynomial factor redundancy in the computation.

1 Introduction

Computer viruses pose one of the central problems in distributed computing today. In this work, we initiate the study of "mobile viruses" (or computer network viruses) — intruders which try to compromise or destroy the system. Our machine model is a synchronous distributed architecture in which a malicious, infinitely-powerful adversary injects/distributes computer viruses at a certain rate at every round. We assume that the detection (of infected sites) can proceed with the same rate as the infection. We note that in practice, this is indeed a reasonable assumption to make [KW]. That is, we allow up to a constant fraction of the machines to be infected at any *single* round, and allow *all the machines* to be eventually infected at *different* rounds. When the virus is detected, the machine is *rebooted*.

* MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge MA 02139; Supported by IBM Graduate Fellowship. Part of this work was done while visiting IBM T.J. Watson Research Center. E-mail: raf@theory.lcs.mit.edu

† IBM Research, T.J. Watson Center, Yorktown, NY 10598. E-mail: moti@watson.ibm.com

We contrast our model with the model of *secure fault-tolerant* synchronous distributed computation, pioneered by [GMW2], where the faults are "stationary". In the model of [GMW2], an adversary can corrupt (during the entire life-time of the protocol) only at most a constant fraction of the processors [GMW2, GHY, BGW, CCD, BB, RB]. In this work, we consider a strictly stronger model, in which we allow our adversary not only to be infinitely-powerful and to corrupt a constant fraction of processors at any time during execution of the protocol, but also, allow him to "move" faults from processor to processor in a dynamic fashion and thus to corrupt *all the processors* during the course of the protocol or the life-time of the system (which can be very large). We show that for this, strictly stronger notion of adversary, the information-theoretic security can be maintained using a constant factor resilience and a polynomial factor redundancy in the computations. Moreover, we show how to maintain our global computation correct and secure in an on-line fashion. We stress that we do not make any assumptions on the nature of the virus — once the machine is infected, it is completely corrupted by the adversary. Moreover, we consider only information-theoretic notions of protection and security.

We allow our adversary to request for any machine and for any round to execute an arbitrary (and potentially malicious) program. That is, in addition to global distributed computation, we allow each machine to request an execution of its local, user-specified (or adversary-specified) program. Of course, such programs are not trusted — measures are taken to make sure that such programs can not do any harm. We show how to execute all such programs in a distributed, secure and reversible manner. (We call this *sequential computation protection*.) Thus, our overall system is self correcting/securing.

In summary, in this work, we make the first step towards a concrete theory of computer viruses by presenting a model which captures some folklore notions of a virus and by showing how to defeat viruses in that model. We model viruses as a certain (malicious) types of faults, which we believe covers a variety of "real" computer virus behaviors. This includes virus activities such as mobility, spread, and the malicious actions the virus itself can perform, including damage to memory, performance of damaging computations, and the compromise of privacy. We show how to cope with all these attacks.

1.1 Motivation and previous work

We start from a very pragmatic scenario. The basic assumption is that in order to combat viruses, the underlying environment can be a distributed network. Indeed, in many places local area networks (as well as larger networks) are available to the organization. Moreover, the current trend is that communication is becoming less and less costly (using the soon-to-be-available high capacity fiber-optics cables.) Indeed, in many cases the right assumption is that communication is less costly than the computation itself (e.g., [CGK]). However, the ever-increasing reliance on computer-networks increases already-existing threat of computer-viruses. In this work, we wish to use this weakness to our advantage — that is, we wish to exploit the distributed nature of the system in order to combat viruses.

Despite the fact that computer viruses are here today, and are considered to be the major threat to computing, there was no attempt to treat them as a general computational phenomenon. The only work we are aware of is Adleman's [A] characterization of computer viruses using recursive theoretic notions and getting impossibility results. On the other hand, there are several practical approaches for fighting viruses. These include: examination and certification of the software (including cryptographic authentication), and virus protection at run-time. In this paper, we take the latter approach. For run-time protection, two different approaches were suggested in practice. One is isolation [Co, A], where the system is closed to the rest of the world. However, this is not realistic for most computer networks. Another approach is fault-tolerance during execution time [De, Ka, Pi, JA]. We take the latter approach here, and show that a very weak detection capability is sufficient to make the computation robust against virus attacks.

Another important point must be emphasized: our solution requires processors to constantly exchange messages in order to defeat the adversary. Being inactive even for a constant number of rounds enables the adversary to compromise the security of the system.

Remark: A natural question might be raised at this point: how come the additional communication messages do not help in spreading the viruses, instead of curbing their spread? The reason is that the additional messages that the processors must exchange are (by virtue of our construction) of a very special form and are *guaranteed* to be virus-free.

A required ingredient in our proof is the necessity of every machine to be able to *erase* parts of its memory. Indeed, without such capability, the protection against mobile viruses is impossible. To the best of our knowledge, this is the first distributed non-cryptographic protocol where the security of the system depends on the erasure capabilities of the machines. On the other hand, for certain recovery procedures we need to *memorize* the history of the computation. In order to resolve these two (at a first glance contradictory) goals, we develop a new mechanism which erases locally (at the machine level) and at the same time remembers globally (at the network level).

1.2 The adversarial setting

We assume that the spread rate is limited and the detection rate competes well with it. That is, we require that at any clock tick, only a fraction of the nodes is infected (i.e. dishonest), but we do not put any restriction on the schedule of the adversary as to which node to corrupt and for how long. We note that this is the weakest possible requirement one must assume – otherwise the network will be completely overtaken at a very few steps by the bad processors.

Jumping ahead, we will show the protection mechanism for an *on-line global computation*. Thus, our model is an application of secure distributed computing to real-time control in the presence of virus attacks. That is, the global memory is "maintained correct" throughout. On the other hand, since local memories may be corrupted, we show how to maintain local reversible memory for each machine, so "local computations" which may be untrusted are recoverable as well.

We allow our faults to be Byzantine. That is, once a node is infected, we assume that it is both a *Trojan-horse virus* (which tries to compromise security) and a *destructive virus* (which tries to divert the computation arbitrarily). Once the processor is infected,

the processor's memory becomes known to the adversary; this is a model of a Trojan horse which transfers information out of the machine (and we take the worst case assumption that the entire local memory is compromised). Moreover, once infected, the node is completely corrupted (i.e. does not have to follow the prescribed protocol.) This models any misbehavior which the virus can create like stopping, message flooding, memory erasing and so on. We assume that an infinitely-powerful, malicious adversary controls the virus behavior and spread. (In practice, measures are being developed to cope with both the Trojan horse viruses [Ka, SG], and the destructive ones [De, Pi].)

Thus, our adversary is allowed to be "mobile", with the following restrictions: First, we assume that we have virus detection capabilities (such as in [De]), in order to allow us to keep the number of infected cites to a constant fraction of the total number of processors. Once the machine is found to be infected, a complete (cold-start) reboot is performed in order to bring a completely fresh version of the program from ROM to memory. Second, we require that the future coin-flips of the machine to be unpredictable prior to the reboot to the adversary, even if he is allowed to inspect the state of the machine just before the reboot. That is, we either require that each coin-flip is generated on-line (which is the practical assumption on generating randomness from physical devices or noise), or, more abstractly, that the entire random tape of the machine is replaced with a new one during reboot. Third, we assume that the rate of the infection spread versus the recovery rate is at most equal, in order to make the system meaningful (otherwise eventually the entire system is infected, which is a hopeless case). Fourth, we require the minimum amount of trusted hardware for I/O to make the interaction of the system with the external world possible. (For example, we must assume that it is possible to reboot a corrupted machine.)

In practice, the network viruses are usually noticeable (i.e. detectable) once they reside long enough and act at a node, and in many systems can be backed-up to an uncorrupted stage. Moreover, in a recent practically-motivated modeling and study of virus spread [KW] it was noticed that after an initial distribution phase with an exponential growth rate, the number of infected cites stabilizes around some fraction of the total number of nodes, thus our assumption is supported by experimental studies and analysis.

1.3 Results

We maintain security when the system runs for polynomially-many rounds (and not just a constant), sequentially executing many (perhaps not even known in advance) protocols. We do not rely on any cryptographic assumptions and exhibit how to achieve our results for the non-trivial case when *all the processors* (during the life-span of the distributed system) become infected at different times. In particular, we establish the following:

- There exists an ϵ, such that if we allow an ϵ fraction of all the machines to be corrupted at each stage, an information-theoretically secure database can be maintained in an on-line fashion with polynomial overhead.
- With polynomial overhead, the network can perform local untrusted updates in a trusted and reversible manner which stabilizes once the machine is cleaned and not infected again.
- The network can efficiently perform secure distributed computations (of global and trusted protocol) in an on-line, correct and secure fashion.

Thus, by extending results developed in [BGW, CCD, BB, RB] to be resilient against a more powerful adversary, we provide a computation technique robust against mobile viruses, using a constant factor resilience and a polynomial factor redundancy in the computations. We note that this is the first application which uses the full power of the results for secure distributed computing (beside direct application for a concrete problem such as secure election) as a building block in our solution.

1.4 Organization of the paper

In the next section we explain the basic definitions, model, and background. We describe the basic tools used in the construction in section 3, in particular a reduction of a distributed data-base system environment to one withstanding mobile virus attacks. Section 4 gives a general review of the system computation. Section 5 shows a reduction of a local computation to a self-stabilized distributed procedure robust against mobile viruses, while section 6 describes our general reduction of a non-robust distributed protocol to a protocol withstanding virus attacks.

2 Preliminaries

2.1 The model

We consider a distributed network in which an information-theoretically secure, distributed database must be (securely) maintained and updated. Our model of computation is a complete synchronous network of n processors. Every processor is a *probabilistic* interactive Turing Machine (as defined in [GMR]). Every pair of processors has a private communication channel and an access to a broadcast channel. The I/O to such a database is done through the hardware at each node (we assume that a hardware device cannot be corrupted) or from some trusted component (e.g., we use a reliable "reboot" mechanism). That is, each processor has a read-only tape which corresponds to a ROM non-volatile memory and a trusted I/O device.

2.2 The adversary

We assume that a fraction of all the nodes in the network is (maliciously) corrupted by an infinitely-powerful adversary at every step of the computation. More specifically, the adversary has t "pebbles" which, at the beginning of every round, he is free to place at any t subset of processors, given all the information coming from "pebbled" nodes. (Placing a pebble on a processor corresponds to corrupting that processor.) When the processor is corrupted, its work tape and random tape can be arbitrarily changed by the adversary. When the pebble is removed from a processor, at the next round the processor is put into a pre-specified (reboot) state and supplied with a new random tape.

Definition 1 *For any $\epsilon < 1$,* **mobile ϵ-adversary** *is an infinitely-powerful machine with $t = \epsilon \cdot n$* pebbles *which operates on an n-processor network.*

In other words, an adversary is allowed to corrupt a constant fraction of processors, (as in [GMW2, BGW, CCD, RB]), and also, after each round, to move pebbles (i.e. faults) from node to node. The node from which the pebble was removed (by an adversary) is the one where a "virus" has been detected, and the node has been "rebooted". That is, the contents of its work tapes are erased, a "fresh" version of its software is loaded (from a read-only tape) and it is put into an initial "reboot" state. In addition, the random tape of the rebooted process is replaced by a new random tape. (This is a fine point worth emphasizing: If the random tape is left unchanged, then, to the adversary, the player becomes deterministic.) We stress that in our model we allow the adversary to decide where the corruption will be detected. For example, if the adversary does not move a pebble from some node during the entire protocol — the node will never be detected to be corrupted. In addition, we allow *rushing*, that is, the adversary can see the messages sent to processors under its control in the current round before it issues messages to be sent by controlled processors in the round.

2.3 The Computation

As a requirement specification of our protocol, we are given a security parameter k, so that $\frac{1}{2^k}$ is negligible and use a probabilistic notion of security, analogous to [BGW, CCD, RB, MiRo]. Intuitively, the correct computation of a protocol/program means that the result computed at the processor while the adversary was active and the program followed the protocol/program in our system, is with very high probability the result of a computation of the program/protocol specification without the presence of the adversary.

We note that the formalization of the above notion is a delicate task, even in the case when the adversary is not allowed to move faults around and especially in protocols which are based on encryption and cryptographic assumptions e.g. [MiRo] (which is not our case – in this work we deal with the information-theoretic notions of correctness and security.) We defer formal definitions and proofs to the full version of the paper.

Definition 2 *We call a function μ: $\mathbf{N} \mapsto \mathbf{N}$* negligible *if for every constant $c > 0$ there exists a N_c such that for all $n > N_c$, $\mu(n) < \frac{1}{n^c}$.*

If M is a probabilistic algorithm, we denote by $M[x]$ the probability distribution on the outputs, given x as an input. (The probability is taken over M's coin tosses.)

Definition 3 *Let $A[x]$ and $B[x]$ be two distributions of strings. $A[x]$ and $B[x]$ are* statistically close *if for any subset of strings S,*

$$|\sum_{y \in S} \mathbf{Prob}_{A[x]}(y) - \sum_{y \in S} \mathbf{Prob}_{B[x]}(y)| < \frac{1}{q(|x|)}$$

for all polynomials q and for sufficiently large x.

We define the view

Definition 4 $\mathbf{VIEW}_A^k(P_x)$ *of the adversary is the probability space assigned to the sequence of messages and memory snapshots of pebbled nodes during the execution of protocol P on input x.*

Informally, we say that the protocol P is *secure* against ϵ-adversary if for all ϵ adversaries A there exists a probabilistic polynomial time algorithm M,

such that the probability distributions $M^{P(x)}(1^k)$ and $\mathbf{VIEW}_A^k(P_x)$ are statistically indistinguishable, where $P(x)$ are the outputs of P, provided to the simulator.

Our protocol starts by the processors committing themselves to the computation by an *input distribution procedure.*

Informally, we say that a protocol P for computing a vector of functions F is *correct*, if it enables with high probability to detect processors which are not committed to the computation (by not following the input distribution procedure), and given the set of processors which are committed, the outputs computed by the protocol are with very high probability the outputs computed by a machine implementing the function F directly with access to the correct processor's inputs. P has to be polynomial in the length of the description of F, and the security parameter.

Our transformations of the program/protocol instruction stream will provide self-securing and self-correcting system. We elaborate on this further in the full version of the paper.

3 Basic Tools

In this section we discuss tools necessary for the solution. In particular, we show how to reduce a distributed data-base system to one withstanding mobile virus attacks. Notice that in our case, the faults are not stationary. Nevertheless, we can still define "faulty" and "non-faulty" processors for a time interval from T_n to T_m $(n < m)$:

Definition 5 *The processor P_i is* faulty *at rounds T_n through T_m if for any j, $n \leq j \leq m$ a pebble is placed by the adversary on P_i.*

We adopt the implementations of *weak secret sharing* **(WSS)** and the *verifiable secret sharing* **(VSS)** as in [RB]. In particular, we assume that there are n players and $t = \epsilon n$ faulty players. Let $s \in Z_p$ be our secret, for some prime number $p > n$. We fix n distinct points $\alpha_1, \ldots, \alpha_n \in Z_p$ known to all players. We recall the definition of a *verified secret s* of [RB]:

Definition 6 *A group of n players holds a* **verified secret (data)** *s, shared using the polynomial $f(x)$, so that $f(0) = s$, and satisfying the conditions of* **VSS** *if:*

1. *The polynomial $f(x)$ is of degree t.*
2. *Each player P_i holds a share of the secret $\beta_i = f(\alpha_i)$*
3. *Every piece β_i was shared by P_i using* **WSS**.

We use the following result of [RB]:

Theorem 1 *(T. Rabin and M. Ben-Or [RB]): Secure distributed circuit evaluation can be performed on verified secrets when the majority of the players are honest, given a broadcast channel. Moreover, the number of rounds is proportional to the depth of the circuit.*

We now introduce a new notion of a *randomized secret s*. The goal is to make randomization of the shares of the shares:

Definition 7 *A group of n players holds a* **randomized secret** *s, if the following conditions are satisfied:*

1. *With s a polynomial f of degree t is associated, such that $f(0) = s$. (f is hidden from all the players)*
2. *With every player P_i a share $\beta'_j = f(\alpha_i)$ is associated (β'_j is hidden from all the players, including P_i.)*
3. *Every β'_j $(1 \leq j \leq n)$ is distributed among n players as a* **verified secret** *β'_j*

Suppose a bit b is a *randomized secret.* Our first goal is to establish that b can be securely maintained in the presence of mobile viruses, despite the fact that we have to re-supply nodes which claim to be "just rebooted" with (presumably lost) data. We achieve this without security breach and show:

Theorem 2 *There exists (an absolute constant) c, such that a distributed data base can be maintained correctly and securely in the presence of mobile $\frac{1}{c}$-adversary.*

Proof Outline: With every β'_j, for every player P_i a share $\beta_{j,i}$ is associated. We call it a *fragment.* If these fragments are going to be kept for long enough time, the adversary will move around and will get enough pieces so to reconstruct β'_j.

We add self-securing procedure to prevent this. The community executes secure circuit evaluation protocol where the inputs (from each player) are its *fragments* (of current round) and a (polynomial number of) random bits: (1) The community draws a random polynomial f' of degree t so that $f(0) = 0$; (2) Computes new shares (secret from every player) $\beta''_i = f(\alpha_i) + f'(\alpha_i)$ for every P_i; (3) Distributes new shares β''_i as new *verified secrets.* Note that the above protocol can be achieved in constant δ number of rounds. After the distribution is done, every (honest) player is required to erase the old fragments, and keep only the new ones. (Since all the currently-honest players actually

do erase, the remaining shares of the bad guys are useless — the secret can not be reconstructed from the remaining old shares.)

More formally, we construct the simulator which, not having the value of the secret can nevertheless construct the view of the adversary, which is statistically close to the actual view of the adversary. Hence, the fragments for the *new shares* β_i'' do not reveal any information about the old shares.

Since there are at most ϵn infected processors at each round and the entire procedure takes less than δ rounds, a polynomial of degree *deg* greater than $\epsilon\delta n$ can be kept information theoretically secure in the process. Hence, we can keep a value alive and secret in the system, while fragments become independent of their old values. ■

The above protocol allows us to implement (local) erasing without (global) forgetting. However, we already assumed that the input is already somehow distributed as a *randomized secret*. There are a few possible assumptions about the input model, the easy one assumes that it is done by a special trusted device or at a starting state before the faults start, or that when it is done correctly all processor can agree to this fact. The more complex way is when the processors do not know at any point that the input is correct. In this case, however, the process can be made *self-stabilized* [Dij] and the system will eventually start from correct data items. In the full version of the paper we will elaborate on the actual possibilities of input process of the initial values and show that in the worst scenario we can nevertheless achieve self-stabilizing correct computation in our model. The *self-stabilized* protocol will be correct in executions in which eventually all faults are eliminated (even though this state may not be recognizable as in [Dij]).

4 Processor Computation: a global view

Next we describe the global view of the operations that every uninfected (i.e. *honest*) machine must follow:

1. Regular operation — at each clock tick each node participates in:
   - Maintenance of the global database.
   - Taking part in a network distributed computation which implements all the individual machines' RAM programs (which we make sure are reversible since the software itself is not trusted). These are called *local untrusted computation.*
   - Taking part in global secure distributed computation protocols These protocols are maintained correct and secure at all times in an on-line fashion. We call such a computation *globally trusted computation*, since the software is reliable.
2. When the virus is detected, (by an auditing system as was modeled in [De], and perhaps by a diligent work of a system manager, or even by the system itself — we leave this act out of the current scope), the node is *rebooted*, which is a necessary basic procedure required for the vitality of the system.

Thus, when a node participates in distributed computation, it fulfills two separate tasks: One is the maintenance and execution of what we call a "global operating system" which maintains the secure network as a whole. The other task of each node is the execution of (untrusted) programs, requested by single processors. While "global operating system" is initially trusted not to have any viruses, the second type of (user-provided) software (requested at a site) can potentially be infected. Hence, the (distributed) execution of untrusted software is made "reversible" in case we need to roll it back when virus is detected.

5 Self-Stabilizing Local Program Execution

In this section we show how to reduce a "local sequential virus" to a "network virus" (or worms, like the Internet Worm [ER]) which propagates in the network. That is, we show how to protect each machine against a local virus attack, provided that one can protect against a global, mobile virus attack. We do so by making computation running at each individual machine *reversible* and then running computations of each individual machine in a distributed fashion.

For these *local untrusted computations* we use a *monotone memory*. Self-stabilizing computation is performed on it (in executions in which the machine with the correct program stops being faulty, the correct program is eventually evaluated– only the machine with the program "knows" about the correctness). In particular, we have a memory array as a monotone memory in which its history is accessible, (along the lines of a fully persistent Data structure as in Driscoll, Sarnak, Sleater, and Tarjan [DSST]). The operation on the memory are refined to local operations on this data structure which enables any refresh

of the memory (as in section 3) and monotonically adding values as the computation progresses, no actual erasing is allowed by the hardware (or program) controlling this memory tape.

Upon detection of faulty (local) computation we perform a "virtual rollback" of the computation:

- Before any local computation starts, we record the (initial) state of every machine. With every processor p we associate a local variable i_p, which is globally (and securely) maintained.
- Every time we start a new local computation of processor p, at time t' we reset i_p to zero.
- If virus is detected at time t, we roll back the local computation to the time $\max\{t - i_p, t'\}$ and set i_p to i_{p+1}.

Thus, the sequential computation of each individual machine can be corrupted and then cleaned: our system can tolerate repetitive infection of the same machine.

Theorem 3 *There exists an (absolute constant) c, such that a local computation request i issued by an untrusted processor can be performed in a secure, correct, non-destructive, and self-stabilized fashion, assuming at most $\frac{1}{c}$ fraction of the machines are infected at each stage.*

Proof Outline: The self-stabilizing computation goes as follows. Given any round r in the computation, the system can produce a state of any machine's memory. The machine can than start from there as the new state (without erasing the history which has declared "bad" since the declaration may have been produced by a virus) The computation is as follows:

- When a machine has to execute a command on its memory it broadcasts it.
- If the command is "reboot" which means that the machine declares an existence of a virus the system performs "virtual rollback" by copying the current memory of the round to which we are rolling back as explained above, and then the community provides a new, global randomization of it.
- If the command is a RAM instruction — a protocol is executed by the community to simulate this command and the result is randomized and stored in a new place (this community computation stage is explained in the next section).

Eventually, when there is a period in which the machine is not attacked for long enough time (but enough to complete the computation), the computation stabilizes on the result (though no one but the machine itself can tell it), the final value is distributed in the global data-base and is maintained as in section 3. (The security and correctness of global computation steps are implied by the result of the coming section.) ■

6 Self-Correcting and Self-Securing Distributed Computation

In this section we cope with "mobile viruses" spreading and attacking global distributed computation. These computations are global protocols known in advance to all machines (unlike "sequential computation" which are results of activating software of a specific, possibly infected, machine.) These computations are maintained correct and secure by the distributed network as a whole. Our goal is to maintain them on-line without the need of backtracking.

In order to perform *globally trusted computations* we use fast information theoretically secure computations of small circuits of [RB, BGW, CCD, BB] and self correcting-securing database maintenance (as explained in section 3).

6.1 Rebooting

First, we consider the issue of *rebooting* of machines. The computation has to take care of rebooting requests agreed upon. Notice that when the processor is re-booted, it comes up in a neutral state. The information that it has "lost" must be re-delivered to him — if this is not done, then after a while (when all processors are rebooted, for example) all the information is lost from the network. On the other hand, the "rebooting" processor should not get any "additional" information even if it is a corrupted processor which just pretends to be re-booting. (Such an additional information, is, for example, old shares.) This two goals seem contradictory at a first glance. To circumvent the problem we first define a *legal* state (i.e. finite state control and contents of work tapes) of the processor at a current state with the computation to be the state of the processor which always follows the (prescribed) protocol. We then claim that the above two goals are not contradictory if what is delivered to the "re-booting" processor is his current *legal* state.

That is, if current *legal* state is given to each processor at each state, the adversary does not gain any additional information it would of not gained without such re-booting information:

Lemma 1 *There exists an (absolute constant) c, such that on-line reboot mechanism is possible, tolerating $\frac{1}{c}$-adversary.*

Proof Outline: Consider two scenarios, the one in which the adversary corrupts the node and then asks for "rebooting" information, and the other in which the adversary moves a pebble to a given node in order to learn its current legal state. In both cases in order to learn the same information, the adversary must put (or keep) a pebble at a current node at a current step. Since every consecutive step (for each processor) is randomized, and since the adversary can look only at $\frac{n}{c}$ nodes at any instance. This allows us to construct the simulator which produces conversations which are statistically close to the real ones. ■

To summarize, the community (as a whole) keeps the current *legal* state of each machine in its "global" (and shared) memory. We must stress here that by "keeping" some values by the community, we mean keeping them in the distributed (i.e. shared) and constantly randomizing fashion, as explained in the previous section.

6.2 Global computation step

Next, we explore how the community can make a single, global computation step:

Theorem 4 *There exists a (absolute constant) c, such that on-line, robust global poly-time computations are possible in our model withstanding $\frac{n}{c}$ mobile viruses.*

Proof Outline: The community must perform a secure distributed circuit evaluation, simulating the computation of each machine (in parallel) with "randomization". This, however, requires to do oblivious circuit evaluation, which brings us to a crucial point in our solution: we do simulation of each step of each machine (with randomization) gate by gate (similar to [RB,BB,BGW,CCD], but also *randomizing after each level* (i.e. gate) of the circuit. Notice that randomization happens on two (different) levels – randomization for each (consecutive in the computation) "legal" (but randomized) states of each machine is the "first" level which we compute gate by gate using another level of randomization (in-between gates.) Are we running in circles? The answer is *no* — since the last (second) level can be done in constant rounds, reducing the solution to the simple case (i.e. if the total number of pebbles $\frac{n}{c}$ the adversary can have times the (constant) number of rounds of our (second level) protocol is less then $\frac{n}{2}$ we are done, by work of [RB].)

We note that in addition to keeping the current, legal and randomized states of every node, the goal of the community is to maintain and update the distributed database according to a trusted global program. The trusted program is performed in a similar fashion as above, maintaining the data-base, computing new gates and re-randomizing the representation. ■

We remark that in the absence of a broadcast channel we can still carry out the secure computation in the presence of mobile viruses. The problem, though, is that fast agreement and collective agreement protocols are only *expected constant time*. This may lead to unexpected delays. In the full paper we show what adaptations are needed in order to perform our computations in this situation.

Our results have certain generalizations which we do not describe here. One is the extension to more general topologies [RB, DDWY], and the other is improvements in efficiency, building on the works of [BE] and [FY].

7 Conclusions and Open Problems

As we stated, this is an initial direction of attacking the "mobile virus" problem. Many questions are left open, regarding efficiency, refinements and improvements of the model, improving our protocols and our results and their adaptation to practical systems, as well as better characterization (generalization or specialization) of the nature of viruses.

Acknowledgments

We thank Jeff Kephart, Matt Franklin and Steve White for helpful discussions and comments.

References

[A] L. Adelman *Abstract Theory of Computer Viruses* CRYPTO 88.

[BB] Bar Ilan J. and D. Beaver, *Non-Cryptographic Secure Computation in Constant Rounds* PODC 1989, ACM.

[BMR] Beaver D., S. Micali and P. Rogaway, *The Round Complexity of Secure Protocols*, STOC 1990, ACM, pp. 503-513.

[BG] Beaver D., S. Goldwasser *Multiparty Computation with Faulty Majority* FOCS 1989, IEEE, pp. 468-473.

[BGW] Ben-Or M., S. Goldwasser and A. Wigderson, *Completeness Theorem for Noncryptographic Fault-tolerant Distributed Computing*, STOC 1988, ACM, pp. 1-10.

[BE] Ben-Or M. and G. El-Yaniv. , Manuscript.

[CCD] D. Chaum, C. Crepeau and I. Damgard, *Multiparty Unconditionally Secure Protocols*, STOC 1988, ACM, pp. 11-19.

[Co] F. Cohen, *Computer Viruses*, Ph.D. dissertation, UCS, 1986.

[CGK] I. Cidon, I. Goapl, and S. Kutten, *New Models and Algorithms for Future Networks*, 7-th ACM PODC, pp. 75-89.

[De] D. Denning, *An Intrusion-Detection Model*, IEEE Sym. on Security and Privacy, 1986, pp. 118-131.

[Dij] E. W. Dijkstra, *Self-Stabilizing Systems in spite of Distributed Control, CACM*,17, 1974, pp. 643-644.

[DDWY] D. Dolev, C. Dwork, O. Waarts and M. Yung, *Perfectly Secure Message Transmission*, FOCS 1990, IEEE.

[DSST] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan, *Making Data Structure Persistent*, STOC 1986, ACM.

[ER] M. Eichin and J. Rochlis, *With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*, IEEE Sym. on Security and Privacy, 1989, pp. 326-343.

[FM] P. Feldman and S. Micali, *An Optimal Algorithms For Synchronous Byzantine Agreement*, STOC 1988, ACM, pp. 148-161.

[F] P. Feldman *Optimal Algorithms for Byzantine Agreement*, MIT Ph.D. Thesis, 1988.

[FY] M. Franklin and M. Yung, *Parallel Secure Distributed Computing*, manuscript.

[GHY] Z. Galil, S. Haber and M. Yung, *Cryptographic Computations and the Public-Key Model*, The 7-th Crypto 1987, Springer-Verlag, pp. 135-155.

[GMW1] O. Goldreich, S. Micali and A. Wigderson, *Proofs that Yield Nothing But their Validity*, FOCS 1986, IEEE, pp. 174-187.

[GMW2] O. Goldreich, S. Micali and A. Wigderson, *How to Play any Mental Poker* , STOC 1987, ACM, pp. 218-229.

[GMR] S. Goldwasser, S. Micali and C. Rackoff, *The Knowledge Complexity of Interactive Proof-Systems*, STOC 1985, ACM, pp. 291-304.

[JA] M. Joseph, and A. Avizienis, *A Fault-Tolerance Approach to Computer Viruses*, IEEE Sym. on Security and Privacy, 1988, pp. 52-58.

[Ka] P.A. Kager, *Limiting the Damage Potential of Discretionary Trojan Horses*, IEEE Sym. on Security and Privacy, 1987, pp. 32-37.

[KW] Jeff Kephart and Steve White, *Directed-Graph Epidemiological Models of Computer Viruses*, IBM Research Report, also in IEEE Sym. on Security and Privacy, 1991.

[MiRo] S. Micali and P. Rogaway *Secure Computation*, a manuscript.

[MR] S. Micali and T. Rabin. *Collective Coin Tossing without Assumptions nor Broadcast*, Crypto-90.

[Pi] J. Piccioto, *The Design of an Effective Auditing Subsystem*, IEEE Sym. on Security and Privacy, 1987, pp. 13-22.

[PSL] M. Pease, R. Shostak, and L. Lamport *Reaching agreement in the presence of faults*, JACM, 27(2), 1980.

[SG] S-P. Shieh, and V. Gligor, *Auditing the Use of Covert Channels in Secure Systems*, IEEE Sym. on Security and Privacy, 1990, pp. 285-295.

[RB] T. Rabin and M. Ben-Or, *Verifiable Secret Sharing and Multiparty Protocols with Honest Majority*, STOC 1989, ACM, pp. 73-85.

[W] Steve White, Personal Communication.