

STABILIZING TRUST and REPUTATION for Self-Stabilizing Efficient Hosts in Spite of Byzantine Guests*

(Extended Abstract)

Shlomi Dolev and Reuven Yagel

Department of Computer Science,
Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
dolev,yagel@cs.bgu.ac.il

Abstract. This work presents a general and complete method to protect a system against possible malicious programs. We provide concepts for building a system that can automatically recover from an arbitrary state including even one in which a Byzantine execution of one or more programs repeatedly attempts to corrupt the system state. Preservation of a guest execution is guaranteed as long as the guest respects a predefined contract, while efficiency is improved by using stabilizing reputation. We augment a provable self-stabilizing host operating system implementation with a contract-enforcement framework example.

Keywords: self-stabilization, security, host systems, Byzantine programs, trust and reputation.

1 Introduction

“Guests, like fish, begin to smell after three days” (Benjamin Franklin). A typical computer system today is composed of several self-contained components which in many cases should be isolated from one another, while sharing some of the system’s resources. Some examples are processes in operating systems, Java applets executing in browsers, and several guest operating systems above virtual machine monitors (VMM). Apart from performance challenges, those settings pose security considerations. The *host* should protect not only its various *guests* from other possibly Byzantine guests [16,18,36], e.g. viruses, but also must protect its own integrity in order to allow correct and continuous operation of the system [43]. Many infrastructures today are constructed with self-healing properties, or even built to be self-stabilizing. A system is *self-stabilizing* [13,14] if it can be started in any possible state, and subsequently it converges to a desired behavior. A *state* of a system is an assignment of arbitrary values to the system’s variables.

* Partially supported by the Lynne and William Frankel Center for Computer Sciences and the Rita Altura trust chair in Computer Sciences.

Recovery with no utility. The fact that the system regains consistency automatically, does not guarantee that a Byzantine guest will not repeatedly drive the system to an inconsistent state from which the recovery process should be restarted. In this work we expand earlier self-stabilizing efforts for guaranteeing that eventually, some of the host's critical code will be executed. This ensures that eventually the host has the opportunity to execute a monitor which can enforce its correctness in spite of the possibly existing Byzantine guests. In particular the host forces the Byzantine code not to influence the other programs' state. Finally, non-Byzantine programs will be able to get executed by the operating system, and provide their services.

Soft errors and eventual Byzantine programs. Even if we run a closed system in which all applications are examined in advance (and during runtime), still problems like soft-errors [38] or bugs that are revealed in rare cases (due to rare *i/o* sequence of the environment that was not tested/considered), might lead to a situation in which a program enters an unplanned state. The execution that starts from such an unplanned state may cause corruption to other programs or to the host system itself. This emphasizes the importance of the self-stabilization property that recovers in the presence of (temporarily or constantly) Byzantine guests. Otherwise, a single temporal violation may stop the host or the guests from functioning as required.

Host-guest enforced contract. The host guarantees preservation of the guest execution as long as the guest respects the predefined rules of a contract. The host cannot thoroughly check the guest for possible Byzantine behaviors (this is equivalent to checking whether the guest halts or not). Therefor the host will force a contract, that is sufficient for achieving useful processing for itself and the guests. The rules enforced by the host can be restrictive, e.g., never write to code segments, and allocate resources only through leases.

Stabilizing trust and reputation. Upon detecting a Byzantine behavior of a guest during run time (namely, sanity checks detect a contract violation) we can not prevent the guest from being executed, since the Byzantine behavior might be caused by a transient fault. Does this mean that we must execute all guests, including the Byzantine ones, with the same amount of resources? Furthermore, when we accumulate behavior history to conclude that a guest is Byzantine, the accumulated data maybe corrupted due to a single transient fault, thus we can not totally count on the data used to accumulate the history. Instead we continuously refresh our impression on the behavior of a guest while continuing executing all guests with different amount of resources. Details of violations are continuously gathered, and the impression depends more on recent behavior history. Such a trust and reputation function rates the guests with a suspicious level, and determines the amount of resources a guest will be granted. In this calculation, recent events are given higher weight, as past event are slowly forgotten. This approach copes with corruptions in the reputation data itself, since wrong reputation fades over time.

Table 1. Byzantine Threats

Mechanism	Examples	Byzantine Threats
Privileges, Address Space separation(MMU)	Commodity OSes	(<i>i/o</i>) Resources tampering, Separation algo. corruption
Type checking	JVM	Resource sharing Self modifying code
Emulation and Dynamic translator	Bochs, Qemu	Compiled code corruption
Hypervisor	Xen, VmWare	Rootkits, Privileged guest corruption

Byzantine guest examples. We review here some systems with their protection mechanisms and possible ways for Byzantine guests to attack. Commodity operating systems use standard protection mechanisms [49] such as several privilege levels and address space separation enforced by hardware, e.g., an MMU. A Byzantine guest can be executed with high privilege (by an unaware user), and corrupt the system's state. Additionally the lack of hardware *i/o* addresses separation in today's common processor architectures enables even kernel data corruption by, say, a faulty device driver. Managed environments like Java or the .NET CLR [22] use various methods of type checking, resource access control and also sandboxing [52]. These mechanisms rely on the correctness of the runtime loaders and interpreters and are also sensitive to self modifying code (see e.g., [12,23]).

Recently, there is a growing interest in virtualization techniques through virtual machine monitors. VMMS may form full emulators like Bochs [8] and Qemu [3] which interpret (almost) all of the guest's instructions (and thus can even ensure correct memory addressing). Other VMMS, like Xen [4], let the guest perform directly most of the processor instructions, especially the non-privileged ones (in [1,23,44] there is a classification of the various VMMS types). Many VMMS rely on one of the guests for performing complex operations such as *i/o*, and thus are vulnerable to Byzantine behavior of this special guest. Some studies, e.g., [44], show other problems in implementing virtualization above the x86 architecture [31], including some privileged instructions which are not trappable in user mode, and direct physical memory access through DMA. Recently, vendors augmented this architecture with partial corrections [40], but still not completely [24].

“Guests” that become super-hosts. Virtualized rootkits [34,43] were recently discussed. They load the original operating system as a virtual machine, thereby enabling the rootkit to even intercept all hardware calls that are made by the guest OS. They demonstrate the relative simplicity of a Byzantine program to take full control of the host. Table 1 summarizes some different mechanisms and their weaknesses.

Related research towards robust hosts. Various protection mechanisms were mentioned in the previous section. Some which emphasize separation and protection are detailed in the following. In [6], a Java virtual machine is

enhanced with operating system and garbage collection mechanisms (type safety and write barriers) in order to prevent, cases like, “a Java applet can generate excessive amounts of garbage and cause a Web browser to spend all of its time collecting it”. Virtual machine emulators have become tools to analyze malicious code [23]. Lately, several studies detailed ways of preventing malicious code from recognizing that it is executing as a guest [23,24,34,40,43,44] (see also [27] for VMM usage for security, and [43] which argues against relying on a full operating system kernel as a protection and monitoring base). In addition, well known hardware manufacturers intend to introduce soon IO-MMUs with isolation capability [1,10]. Operating system based emulators or hypervisors such as UML [15] or KVM [33] are used also to analyze suspected programs. [41] uses virtualization techniques to contain errors, especially in drivers, in realtime. Methods that are based on secure boot (e.g., [2,45,51]) are important in order to make sure that the host gets the chance to load first, and prevent *rootkits* from fooling it by actually running the host as a guest [34]. In [42], cryptography techniques are used in order to ensure that only authorized code can be executed.

Sandboxing techniques were presented in [52] (see also [26]). Sandboxing techniques make sure that code branches are in segment (a distinct memory section which exclusively belongs to a process), and also rely on different segments for code and data. For every computed address they add validation code that traps the system, or just masks addresses to be in the segment (this is actually a sandbox). They count on dedicated registers which hold correct addresses. Overview of trust and reputation can be found, e.g., in [25,37]. In [7] a Bayesian based approach with exponential decay is proposed.

The need for address space separation, the use of capabilities, minimal trusted base and other protection mechanisms were introduced in well known works [9,11,35,39,46,49]. Singularity [29,28] achieves process isolation in software by relying on type safety, and also prevents dynamic code. Self-modifying code certification is presented in [12].

Generally, extensive theoretical research has been done towards self-stabilizing systems [13,14,48] and autonomic - computing/ disaster - recovery/ reliability - availability - serviceability [30,32,50]. However, none of the above suggest a design for a host system that can automatically recover from an arbitrary state, even in the presence of Byzantine guests that repeatedly try to corrupt the system state.

Our contribution. (a) Identifying the need of combined self-stabilization, and techniques for enforcing a contract over the operations of a guest. We show that only such a combination will allow (useful) recovery. (b) The introduction of stabilizing trust and reputation and the use of the level of trust as a criteria for granting resources while continuing to evaluate the trust of the guests. (c) Concepts and a proof for designing hosts and contracts. (d) A running example.

Paper organization. Next, in Section 2, we briefly review results from previous works on self-stabilizing operating systems, which form our basis for a protected host system. Section 3 details the system settings and requirements. This is

followed by Section 4 which presents a general framework for protecting against Byzantine programs. Section 5 presents an example of a simple Byzantine guest followed by the way a provable host implementation copes with such a Byzantine guest. As a result of paper length limitation, proofs and some technical and implementation details are omitted from this extended abstract.

2 Self-stabilizing Operating Systems – Foundations Overview

In previous works [19,20,21], we presented new concepts and directions for building a self stabilizing operating system kernel. A self-stabilizing algorithm/system makes the obvious assumption that it is executed. This assumption is not simple to achieve since both the microprocessor and the operating system should be self-stabilizing, ensuring that eventually the (self-stabilizing) applications/programs are executed. An elegant composition technique of self-stabilizing algorithms [14] is used to show that once the underlying microprocessor stabilizes the self-stabilizing operating system (which can be started in any arbitrary state) stabilizes, then the self-stabilizing applications that implement the algorithms stabilize. This work considers the important layer of the operating system.

One approach in designing a self-stabilizing operating system is to consider an existing operating system (e.g., Microsoft Windows, Linux) as a black-box and add components to monitor its activity and take actions accordingly, such that automatic recovery is achieved. We called this approach the black-box based approach. The other extreme approach is to write a self-stabilizing operating system from scratch. We called this approach the tailored solution approach. We have presented several design solutions in the scale of the black-box to the tailored solutions. The first simplest technique for the automatic recovery of an operating system is based on repeatedly reinstalling the operating system and then re-executing. The second technique is to repeatedly reinstall only the executable portion, monitoring the state of the operating system and assigning a legitimate state whenever required. Alternatively, the operating system code can be “tailored” to be self-stabilizing. In this case the operating system takes care of its own consistency. This approach may obviously lead to more efficient self-stabilizing operating systems, since it allows the use of more involved techniques.

Tailored Approach. An operating system kernel usually contains basic mechanisms for managing hardware resources. The classical Von-Neumann machine includes a processor, a memory device and external I/O devices. The tailored operating system is built (like many other systems) as a kernel that manages these three main resources. The usual efficiency concerns which operating systems must address, are augmented with stabilization requirements.

- **Process Scheduling.** The system is composed of various processes which are executing each in turn. The process loading, executing and scheduling part of the

operating system usually forms the lowest and the most basic level. Two main requirements of the scheduler are fairness and stabilization preservation. Fairness means that in every infinite execution every running process is guaranteed to get a chance to run. Stabilization preservation means ensuring that the scheduler preserves the self-stabilization property of a process in spite of the fact that other processes are executed as well (e.g., the scheduler ensures that one process will not corrupt the variables of another process).

- **Memory Management.** We deal with two important requirements to the tasks of memory management. The first requirement is the *eventual memory hierarchy consistency*. Memory hierarchies and caching are key ideas in memory management. The memory manager must provide eventual consistency of the various memory levels. The second requirement is the *stabilization preservation* requirement. It means that stabilization proof for a single process p is automatically carried to the case of multiprocessing in spite the fact that context switches occur and the fact that the memory is actually shared. Namely, the actions of other processes will not damage the stabilization property of the process p .

- **I/O Device Drivers.** Device drivers are programs which are practically an essential part of any operating system. They serve as an adaptation layer by managing the various operation and communication details of I/O devices. They also serve as a translation layer providing consistent and more abstract interface for other programs and the hardware device resources (and sometimes they also add extra services not provided by the hardware devices). Device drivers are known to be a major cause of operating system failures [41].

In [21] we define two requirements which should be satisfied in order for the protocol between the operating system and an I/O device to be self-stabilizing. The first requirement (the ping-pong requirement) states that in an infinite system execution, in which there are infinitely many I/O requests, the OS driver and the device controller are infinitely often exchanging requests and replies. The second requirement is about progress and it states that eventually every I/O request is executed completely and correctly according to some protocol specification (e.g., the ATA protocol for storage devices). A device driver and device controller can be viewed as a master and a slave working together according to some protocol to achieve their mission. Thus, the device driver acting as a master can check that the slave is following, e.g. the ATA protocol, correctly.

The usage and usefulness of such a system in critical and remote systems cannot be over emphasized. For example entire years of work maybe lost when the operating system of an expensive complicated device (e.g., an autonomous spaceship) may reach an arbitrary state (say, due to soft errors) and be lost forever (say, on Mars). The controllers of a critical facility (e.g., a nuclear reactor or even a car) may experience an unexpected fault (e.g., an electrical spike) that will cause it to reach an unexpected state, from which the system will never recover, therein leading to harmful results. Our proofs and prototypes show that it is possible to design a self-stabilizing operating system kernel.

3 Settings and the Requirements

Definitions. We briefly define the system states and state transitions (see [19,20] for details concerning processor executions, interrupt, registers, read-only memories, a watchdog and additional settings). A *state* of the system is an assignment to its various memory components (including the program counter register). A *clock tick* triggers the microprocessor to *execute a processor step* $ps_j = (s, i, s', o)$, where the inputs i and the current state of the processor s are used for defining the next processor state s' , and the outputs o . The *inputs and outputs* of the processor are the values of its *i/o connectors* whenever a clock tick occurs. The processor uses the *i/o connectors* values for communicating with other devices, mainly with the memory, via its data lines. In fact, the processor can be viewed as a transition function defined by e.g., [31]. A *processor execution* $PE = ps_1, ps_2, \dots$ is a sequence of processor steps such that for every two successive steps in PE , $ps_j = (s, i, s', o)$ and $ps_{j+1} = (\bar{s}, \bar{i}, \bar{s}', \bar{o})$ it holds that $\bar{s} = s'$.

Error model – arbitrary transient and Byzantine faults. The system state, including the program counter, system data structures and also the program code and data in RAM, may become arbitrarily corrupted, namely, assigned any possible value. This model is an extension of a previous one ([19]). The main feature of the extension is the removal of the assumption that all programs are self-stabilizing (or restartable [4]) so they might exhibit Byzantine behavior forever.

Requirements. We now define the requirements which should be satisfied for a host system to be self-stabilizing in spite of a Byzantine behavior.

(r1) Guest stabilization preservation. The fact that the host system may start in an arbitrary state, and execute code of Byzantine guests, will not falsify the stabilization property of each of the non-Byzantine guests in the system.

(r2) Efficiency guarantee. Non-Byzantine guests will eventually get the needed resources in order to supply their intended services.

Note that both (r1) and (r2) implicitly require that a program that shares resources with others, will not block or will be blocked, outside of acceptable limits, due to this sharing (although in the worst case, due to the use of leases combined with a reputation system, resource will eventually be granted).

4 Concepts for Fighting the Byzantines

By combining techniques like secure booting, contract verification and enforcement together with self-stabilization we can protect a system against Byzantine guests in a provable way.

- **Secure booting** ensures that there is a minimal trusted computing base which runs programs and monitors.
- **Offline Byzantine behavior detectors** use code verification techniques, analyzing a program offline and looking for possible breaks of contracts.

- **Runtime anti-Byzantine enforcers** insert additional instructions in the executable for online sanity checks to enforce contract properties during a program execution.
- **Stabilizing trust and reputation** for determining the amount of resources a guest will be granted.
- **Self-stabilization** of these mechanisms and their composition [5,14] ensures that the system is eventually protected and functioning.

Secure booting is achieved through standard hardware based mechanisms (e.g., [2,45,51]). These are essential in order to guarantee that a Byzantine guest is not loaded first.

The system should be augmented with a detector framework which executes one or more upfront offline Byzantine detector plug-ins. A detector is built to enforce some aspect of a contract with a guest, and must be provable to perform its action completely and within acceptable time limits. These detectors scan the program code in advance for particular violations of the contract that are easy to check, and in case the scan reveals a Byzantine guest, this guest will not be loaded at all.

A program that passes the first check is augmented with sanity checks and access restrictions in sensitive code parts, where execution might do harm. The augmented code does not change the program semantics (up to stuttering) as long as the guest respect the contract. Upon detection of a violation in runtime, an enforcer can reload the program code and also update the trust and reputation level. An example for such an enforcer is one that enforces that segments used by the program are not changeable (meaning that self-modifying code is forbidden according to a contract). Runtime sanity checks, look for possible instruction sequences to make sure they do not violate the contract. Note that due to transient faults (that are rare by nature), a target address may change right after a sanity check, causing the system later to start a convergence stage as a self-stabilizing system should. In the case of a Byzantine program, the harm is prevented, although the detection and reloading will occur again and again (the trust and reputation record of a guest will limit the amount of processing used for this particular guest). In case the program is not Byzantine, the reload-of-code procedure will ensure correct behavior after which the trust and reputation will reach the maximal possible level.

Stabilizing trust and reputation can be achieved by using methods which favor recent events over past events. One example is [7] which combines a Bayesian approach with exponential decay. In such ways, trusted guests get more resources overtime, while suspected guests are not totally blocked and get chance to “shun evil and do good”. Such approaches also cope with transient (fault) corruptions in the reputation data, since wrong reputation fades over time.

Theorem 1. *There exists a self-stabilizing host that can fulfill (r1) stabilization preservation and (r2) efficiency guarantees for guests.*

Sketch of proof: We list the mechanisms we use and the properties we establish by them. (a) The host is built above a self-stabilizing hardware ([17]) which

guarantees eventually correct operation of the hardware from any state. (b) A self-stabilizing host operating system [19] which is guaranteed to periodically run some boot-code loaded in a secure way [2,45,51], without being subverted ([43]) (c) This trusted operating system guarantees eventual execution of all runnable processes including the contract offline detectors. (d) Code is being refreshed ([19,20]) periodically, so Byzantine or wrong behavior caused by transient faults to code segments are eventually fixed. (e) Contract properties are asserted by online enforcers. (f) Self-stabilizing programs might be supplied with a list of “initial” safe states. In such a case when recognizing Byzantine behavior, apart from preventing this behavior and refreshing the code, the closest state (using Hamming distance or some other metric) can be applied to the program. (g) All resource allocations are granted using leases with a self-stabilizing manager, as demonstrated in a previous work on dynamic memory [20], ensuring that resource allocations are eventually fair. The contract detectors and enforcers check also for behavior which violates the leasing rules. Resource are leased to a guest according to its trust and reputation level. (h) System calls and traps are also leased (again, according to the trust and reputation level), so a Byzantine guest is limited in the number of times it can cause long delay due to system calls. (i) Non-Byzantine programs are stabilizing in spite of faults and Byzantine behavior. (j) The interaction between those programs and other programs or devices is stabilizing too ([21]). (k) The stabilization process of one program only affects the state of this program and does not affect other programs. Thus, stabilization preservation and efficiency guarantee is achieved for guests. \square

The implementations presented next, add sanity checks to branches and memory accesses, ensure correct use of leased resources, and enforce allowed patterns of out of memory accesses.

5 Host Implementation Example

In previous works we demonstrated the construction of a self-stabilizing operating system (SOS) [19,20,21,47]. Guest separation was achieved by using the segmentation mechanism of the Pentium processor [31], without MMU hardware protection. Additionally, we assumed that the code of the programs is hardwired and correct, thus a program does not contain instructions which affect the state of the other programs (including the system). When we introduce programs with arbitrary code, other programs, even the host/operating system itself, may be corrupted. In the current work we have implemented a prototype of a simple host that satisfies requirements (r1) and (r2) above the mentioned system.

To demonstrate the possible corruption of the system designed, we show an example of a threat in a program that accesses the operating system’s segment and changes the scheduler state. The scheduler state is changed so that this program will be scheduled (again

1	<code>mov ax, 0x8010</code>
2	<code>mov ds, ax</code>
3	<code>mov word [0x292], 3</code>

Fig. 1. Byzantine Code

and again) instead of other guests. Figure 1 shows an example of such a 16-bit x86 assembly code. Lines 1-2 change the data segment pointer to the system's segment. Then, line 3 changes the process pointer contents to a value which will cause re-scheduling of this program.

One can argue that address-space separation, like found in commodity operating system kernels, can prevent this behavior. But if a Byzantine program manages to operate in a privileged mode, even once due to a transient fault, the separation algorithm itself might be subverted, followed by the above malicious behavior. Next we will describe our settings in order to show a provable solution.

To demonstrate these ideas we show: (a) an example containing added code that enforces memory access within a program's data segments (sandboxing). (b) Accessing shared resources through leases. (c) A prototype of a detector that performs offline verification that out of segment accesses are according to a list of known patterns allowed by a contract. (d) An example of stabilizing trust and reputation evaluation according to online sanity checks.

The suggested solution uses an architecture in which some code is read-only (Harvard model). A non-maskable interrupt (NMI) is generated by a simple self-stabilizing watchdog. Thus, the hardware triggers a periodic execution of the host monitoring (detectors) code. This architecture also guarantees that the monitoring code gets enough time to complete. A detector searches the code of every program to make sure it does not contain code that changes segments outside the scope of the program. Computed addresses are enforced to be within limits, by inserting sanity checks. The correct use of leased resources is also enforced during runtime. Additionally, from time to time the host refreshes the code of all guests (including sanity checks insertions), say from a CD-ROM, to allow self-stabilization of programs following a code refresh.

(a) Figure 2 line 1 demonstrates calculation of a segment selector value, as opposed to Figure 1 in which the address is fixed. Then, lines 2-5 are a sanity check added by the runtime anti-Byzantine enforcer. First the calculated address is validated to be in range (in this example it must have some fixed value), in case of a violation detection (line 3) the Increase-Bad-Reputation procedure is called to record the violation (see (d) below). Then in line 5, the correct address is enforced. Alternatively, a monitor could start actions of reloading code and data in case of detecting such a wrong access.

```

1  mov ax, <<computed address>>
   // added sanity check
2  xor ax, SEGMENT_MASK
3  jz AfterSanityCheck
4  call Increase-Bad-Reputation
5  mov ax, FIXED_SEGMENT
AfterSanityCheck:
...
```

Fig. 2. Memory Access Enforcer

(b) Figure 3 presents the way a program uses a shared resource, in this case the dynamic memory heap. The contract is that all accesses to segments in this memory area must happen only through the segment selector register **fs**. Additionally, in order for this access to be leased, a program is not allowed to load a value in this register, but instead asks the system for a leased allocation (line 1). After this allocation request, and before every dynamic memory access, the program must check that the lease is still in effect, by checking the value in **fs** (lines 2-3). In case the allocation failed or expired, the value will be 0. Detectors and enforcers check that the use of shared resources is done according to this contract [20] and penalize the program in case of irregular use detection.

(c) The Pentium's operation code for moving a value into one of the 16-bit segment selector registers, is **8e**. Thus, the offline detector searches for commands starting with this code (assuming for simplicity that this is the only possible way). Note that the Pentium has variable length operations so in order to detect beginnings of operations we need to use disassembly techniques. Additionally, the mentioned operation code is sometimes used

```

1  call MM_Alloc
After_MM_Alloc:
2  cmp fs, 0
3  jz TryLater
...
```

Fig. 3. Shared Resource Access

in legitimate ways, e.g. for accessing dynamic data segments. A possible solution is allowing known fixed patterns of access to other segments. An example pattern appears in Figure 4, where the program is accessing the video segment, which is needed for screen output. The **es** segment register is loaded in line 2 by the allowed value that is computed in line 1. In this case the **8e** op-code is preceded by the sequence **b8 00 b8** which is considered valid. Figure 5 presents the algorithm of the segment access detector. This detector is executed before loading the guest program. It scans a program's code segment for the **8e** code. When found, it verifies that it is preceded by one of the allowed patterns, otherwise the program is considered as one that does not respect the contract and therefore an upfront Byzantine program.

(d) Upon finding an online contract violation through performing the enforced sanity checks, the violation is recorded in the reputation history record (Figure 6). This record maybe kept as part of a *process entry* in the system's process table. Every predefined period of time (or steps) the system updates this record, as seen in in the Decay-Reputation procedure in Figure 6. The entries in the record are shifted in a way that the oldest entry is removed, thus implementing the needed decay and stabilization. This updated record is used for evaluating

```

1  mov ax, VIDEO_SEGMENT
2  mov es, ax
```

Fig. 4. An Allowed Pattern

```

BYZANTINE-DETECTOR(process_entry, legal_patterns)
1  for each instruction_code(ic) in process_entry.code_segment
2  do if ic starts_with "8e"
3      then for each pattern in leagal_patterns
4          do if pattern precedes ic
5              then continue_main_loop
6          process_entry.byzantine  $\leftarrow$  true
7  return

```

Fig. 5. Out of Segment Access Detector

```

INCREASE-BAD-REPUTATION(process_entry)
1  process_entry.reputation[0]  $\&=$  BAD_REPUTATION_BIT
2  process_entry.reputation[0]  $\ll$  1  $\triangleright$  Shift left.
3  return

DECAY-REPUTATION(process_entry)
1  for i in (MAX_HISTORY - 1) .. 1
2  do process_entry.reputation[i]  $\leftarrow$  process_entry.reputation[i - 1]
3  return process_entry.reputation

```

Fig. 6. Update Trust and Reputation – Increase and Decay

the trust and reputation level of the relevant guest and granting resources in accordance.

Performance issues. The timing of the execution of code refreshing (and offline detectors) can be tuned according to the expected rate of soft-error corruptions to code. This processes do not have a great impact on the program execution performance, since the frequency of the checks may be balanced against the desired recovery speed.

One could suggest a performance gain by having an auxiliary processor (one core of a multi-core) for performing a repeated contract verification on the loaded code. However, a Byzantine guest might fool the auxiliary processor, say, by changing the sanity checks to be correct whenever the auxiliary processor is checking them. Still we can use such a processor for most of the cases to speed the indication on soft errors and to trigger code refreshing.

6 Concluding Remarks

In this work we presented an approach to use self-stabilizing reputation in order to gain efficient performance. We believe that self-stabilizing host systems that use stabilizing reputation are a key technology which can cope with Byzantine behavior in critical computing system. Source code examples can be found in [47].

References

1. Adams, K., Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization. In: ASPLOS. Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, CA (2006)
2. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of 1997 IEEE Symposium on Computer Security and Privacy, IEEE Computer Society Press, Los Alamitos (1997)
3. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: Proc. of USENIX Annual Technical Conference. FREENIX Track (2005)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing, NY, USA (2003)
5. Brukman, O., Dolev, S., Haviv, Y., Yagel, R.: Self-Stabilization as a Foundation for Autonomic Computing. In: FOFDC. Proceedings of the Second International Conference on Availability, Reliability and Security, Workshop on Foundations of Fault-tolerant Distributed Computing, Vienna, Austria (April 2007)
6. Back, G., Hsieh, W.H., Lepreau, J.: Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In: OSDI. Proc. 4th Symposium on Operating Systems Design and Implementation, San Diego, CA (2000)
7. Buchegger, S., Le Boudec, J.-Y.: A Robust Reputation System for Mobile Ad-hoc Networks. Technical Report IC/2003/50, EPFL-IC-LCA (2003)
8. Bochs IA-32 Emulator Project. <http://bochs.sourceforge.net/>
9. Bershad, B.N., Savage, S., Pardyak, P., Sizer, E.G., Fiuchynski, M., Becker, D., Eggers, S., Chambers, C.: Extensibility, Safety, and Performance in the SPIN Operating System. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, Colorado, December (1995)
10. Ben-Yehuda, M., Xenidis, J., Mostrows, M., Rister, K., Bruemmer, A., Van Doorn, L.: The Price of Safety: Evaluating IOMMU Performance. In: OLS. The 2007 Ottawa Linux Symposium (2007)
11. Chase, J.S., Levy, H.M., Feeley, M.J., Lazowska, E.D.: Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems* 12(4) (November 1994)
12. Cai, H., Shao, Z., Vaynberg, A.: Certified Self-Modifying Code. In: Proceedings of PLDI 2007, CA (2007)
13. Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM* 17(11), 643–644 (1974)
14. Dolev, S.: Self-Stabilization. The MIT Press, Cambridge (2000)
15. Dike, J.: A User-mode Port of the Linux Kernel. In: 5th Annual Linux Showcase and Conference, Oakland, California (2001)
16. Dalot, A., Dolev, D.: Self-stabilizing Byzantine Agreement. In: PODC 2006. Proc. of Twenty-fifth ACM Symposium on Principles of Distributed Computing, Colorado (2006)
17. Dolev, S., Haviv, Y.: Stabilization Enabling Technology. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 1–15. Springer, Heidelberg (2006)
18. Dolev, S., Welch, J.: Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. In: UNLV. Proc. of the 2nd Workshop on Self-Stabilizing Systems (1995). *Journal of the ACM*, Vol. 51, No. 5, pp. 780-799, September 2004.

19. Dolev, S., Yagel, R.: Toward Self-Stabilizing Operating Systems. In: SAACS04, DEXA. Proceedings of the 15th International Conference on Database and Expert Systems Applications, 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems, Zaragoza, Spain, pp. 684–688 (August 2004)
20. Dolev, S., Yagel, R.: Memory Management for Self-Stabilizing Operating Systems. In: Proceedings of the 7th Symposium on Self-Stabilizing Systems, Barcelona, Spain (2005). also in *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 2006.
21. Dolev, S., Yagel, R.: Self-Stabilizing Device Drivers. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 276–289. Springer, Heidelberg (2006)
22. ECMA International. ECMA-335 Common Language Infrastructure (CLI), 4th Edition, Technical Report (2006)
23. Ferrie, P.: Attacks on Virtual Machine Emulators. Symantec Advanced Threat Research, http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf
24. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility Is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems, San Diego, CA (2007)
25. Guha, R., Kumar, R., Raghavani, P., Tomkins, A.: Propagation of trust and distrust. In: WWW. Proceedings of the 13th International World Wide Web conference (2004)
26. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (1997)
27. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A virtual machine-based platform for trusted computing. In: Proceedings of SOSP 2003 (2003)
28. Hunt, G., Larus, J.: Singularity: Rethinking the Software Stack. *Operating Systems Review* 41(2) (April 2007)
29. Hunt, G., Aiken, M., Fhndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D., Wobber, T.: Sealing OS Processes to Improve Dependability and Safety. In: Proceedings of EuroSys2007, Lisbon, Portugal (March 2007)
30. Intel Corporation. Reliability, Availability, and Serviceability for the Always-on Enterprise, The Enhanced RAS Capabilities of Intel Processor-based Server Platforms Simplify 24 x7 Business Solutions, Technology@Intel Magazine (August 2005), http://www.intel.com/technology/magazine/Computing/Intel_RAS_WP_0805.pdf
31. Intel Corporation. The IA-32 Intel Architecture Software Developer's Manual (2006), <http://developer.intel.com/products/processor/manuals/index.htm>
32. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. *IEEE Computer*, 41–50 (January 2003), See also <http://www.research.ibm.com/autonomic>
33. *KVM: Kernel-based Virtual Machine for Linux*, <http://kvm.qumranet.com/>
34. King, S.T., Chen, P.M., Wang, Y., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: IEEE Symposium on Security and Privacy (May 2006)
35. Lamport, B.W.: Protection. In: Proceedings of the 5th Princeton Symposium on Information Sciences and Systems, Princeton University (March 1971). Reprinted in *ACM Operating Systems Review* (January 1974)

36. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems* 4(3), 382–401 (1982)
37. Mui, L.: Computational Models of Trust and Reputation: Agents, Evolutionary Games, and Social Networks. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (2002)
38. Mastipuram, R., Wee, E.C.: Soft errors' impact on system reliability. *Voice of Electronics Engineer* (2004), <http://www.edn.com/article/CA454636.html>
39. Neumann, P.G.: *Computer-Related Risks*. Addison-Wesley, Reading (1995)
40. Neiger, G., Santony, A., Leung, F., Rogers, D., Uhlig, R.: Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal* 10(3) (August 2006)
41. Swift, M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. In: *SOSP 2003. Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY (October 2003). See also: M. Swift. *Improving the Reliability of Commodity Operating Systems*, Ph.D. Dissertation, University of Washington (2005)
42. Sharma, A., Welch, S.: Preserving the integrity of enterprise platforms via an Assured eXecution Environment (AxE). In: *OSDI. A poster at the 7th Symposium on Operating Systems Design and Implementation* (2006)
43. Rutkowska, J.: "Subvirting Vista Kernel For Fun and Profit — Part II Blue Pill", see also (2006), http://www.whiteacid.org/misc/bh2006/070_Rutkowska.pdf, <http://www.whiteacid.org/papers/redpill.html>
44. Robin, J., Irvine, C.: Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor. In: *Usenix annual technical conference* (2000)
45. Ray, E., Schultz, E.E.: An early look at Windows Vista security. *Computer Fraud & Security* 2007(1) (2007)
46. Schroeder, M.D.: Cooperation of Mutually Suspicious Subsystems in a Computer Utility. Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA (September 1972)
47. SOS download page. <http://www.cs.bgu.ac.il/~yagel/sos>, 2007
48. <http://www.selfstabilization.org>
49. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. *Proceedings of the IEEE* 63(9), 1268–1308 (1975)
50. Sun Microsystems, Inc. 'Predictive Self-Healing in the Solaris™ 10 Operating System', White paper (September 2004), http://www.sun.com/software/solaris/ds/self_healing.pdf
51. Tygar, J.D., Yee, B.: Dyad: A system for using physically secure coprocessors. In: *Proceedings of IP Workshop* (1994)
52. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-based fault isolation. In: *Proceedings of the Sym. On Operating System Principles* (1993)