Verified Secure Kernels and Hypervisors for the Cloud

Matthieu Lemerre[1], Nikolai Kosmatov[2], and Céline Alec[2]

[1] CEA, LIST, Embedded Real-Time System Laboratory, PC 172, 91191 Gif-sur-Yvette, France.

[2] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France.

firstname.lastname@cea.fr

Abstract. Cloud-computing systems share hardware resources (CPU time and memory) between mutually untrusted applications. As such, they must provide isolation barriers between these applications, but must also secure resource sharing such that applications cannot stop, slow down, or provoke incorrect resource accounting of other applications. These isolation barriers are implemented by a trusted operating system kernel, which must be built according to security principles.
Confidence in the system can be further increased with the help of tools for formal verification and proof of programs. Using these tools is eased because thorough application of security principles leads to a small system, but still poses many challenges for formal verification, like concurrency and the need to model the behavior of the hardware.
This paper shows how modern tools for proof of programs can be applied to verification of secure kernels and hypervisors for the Cloud. We illustrate this approach on a critical module of a prototype Cloud hypervisor, called Anaxagoros, using the Frama-C software verification platform.

1 Introduction

Secure systems are traditionally built according to design principles that decompose the system into *isolated* components with minimal *rights*, and with communication between components tightly controlled.

This isolation is very important for cloud computing, that executes applications from mutually untrusted users: a failure or attack from an application of a user must not interfere with applications of other users (or other applications of the same user).

But cloud systems have another requirement, which is to mutualize the resources of the system on which they are built, so as to maximize efficiency. Resources of the system must be shared *securely* between the tasks, for instance, in order to make denials of service impossible (i.e. to prevent an application to slow down or stop another one, for instance by requesting all resources), or to correctly account for the memory and CPU time spent executing the applications, and thus, to correctly bill the clients. Secure resource sharing requires to

complement the traditional behavioral security principles [1] with new *resource security* principles.

The Anaxagoros [2,3] system has been designed to maximize both traditional and resource security. It is composed of:

- A *microkernel*, that implements the isolation between tasks, controls access to the services; it is trusted by all tasks in the system;
- Some *services*, each sharing one resource (memory, network...) securely between the tasks, and being trusted only by the tasks that need the resource;
- *Libraries*, helping to use the resources in the system (e.g. the C library), and being trusted only by the task that uses them.

The amount of system code that must work properly to correctly execute a program is called its *trusted computing base (TCB)*. The implementation of Anaxagoros tries to put the code first into libraries then into shared services, and then into the kernel, leading to minimizing the TCB of each task of the system. Critical tasks can minimize the amount of shared services used and avoid using the provided libraries, thereby having a minimal TCB.

To minimize TCB, the kernel and services present a low-level interface (like exokernels [4]): they consist in a thin layer that only refuses or authorizes operations requested by user applications. This low-level interface also allows efficient virtualization of operating systems [5], such as Linux, to provide security and isolation between existing systems.

In particular, the Anaxagoros kernel is globally trusted, even by critical tasks, which makes it the most critical component of the system. The fact that it is minimized (approximately 3500 lines of code) decreases the likeliness of bugs and helps in code reviews. But its minimality and criticality make it a good target to further increase confidence in the system, by providing a formal proof that the kernel offers the required security functions.

Verification of such system software represents an interesting and challenging target for software verification because of its critical and complex functions, such as concurrency, the use of assembly code, and the need to model interactions with the hardware.

This paper is organized as follows. Sec. 2 presents a short tutorial on proof of programs with the JESSIE tool. Sec. 3 introduces basic system security rules, describes Anaxagoros microkernel and in particular its virtual memory system. Sec. 4 shows how this system it can be formally verified. Sec. 5 and 6 provide related work and conclusion.

2 A short tutorial on proof of programs

In this section we show how a program can be formally verified using automatic tools for proof of programs.

We use FRAMA-C [6,7], an open-source platform dedicated to analysis of C programs, developed at CEA LIST. FRAMA-C has a plugin-oriented architecture that allows the user to run analyzers already available in the platform

```
1 /*@ ensures \result >= a && \result >= b &&
2         ( \result == a || \result == b );
3     assigns \nothing;
4 */
5 int max(int a, int b){
6   if( a > b )
7     return a;
8   else
9     return b;
10 }
```

Fig. 1. File `max.c` with a function returning the maximum of its inputs `a` and `b`

as well as to develop new plugins. FRAMA-C offers various analyzers, many of them are open-source. They implement a wide range of modern software analysis techniques, such as abstract interpretation, impact analysis, dependency analysis, program slicing, pointer analysis, weakest precondition, etc. The structural test generation tool PATHCRAWLER [8,9] is also available as a FRAMA-C plugin. FRAMA-C contains two plugins for proof of programs, JESSIE and WP.

Proof of programs, also known as *theorem proving,* is a powerful technique of program verification that provides a formal mathematical proof that the program meets its specification. While the theoretical foundations [10,11] of this approach were developed in the 1970s, its automation became possible only during the last decade thanks to the spectacular progress made by the developers of modern program verification tools.

Among them, automatic theorem provers, like Simplify [12], ALT-ERGO [13,14] and Z3 [15], are capable to perform simple proofs automatically. On the other hand, interactive provers, such as COQ [16,17,18], ISABELLE [19,20], and HOL [21,22], allow the engineer to indicate proof steps that are then checked by the tool. Interactive provers may be suitable for more complex properties but require human interventions.

In this approach, to prove a program p, one proceeds in the following way.

- First, each function f in the program source code must be *annotated,* or *specified,* by inserting into the code special clauses, or *annotations*, indicating the hypotheses h and conclusions c. They describe the state of program variables at different execution points and program actions. Typically, the hypotheses describe the state before the function is called, while the conclusions may specify how this state is modified by the function, and the return values. In this manner, each function receives a *specification,* or a *contract.* Basically, the verification task is then reduced to the proof of the implication $h \Rightarrow c$.
- Next, these annotations are used to compute *proof obligations*, i.e. propositions that must be proved to ensure that if the hypotheses h are verified, then the conclusions c hold. The proof obligations can be very complex since they should take into account all program instructions including variable assignments, conditionals, loops, other function calls, etc.
- Finally, a theorem prover is called to prove the proof obligations.

```
1  /*@ requires l >= 0;
2      requires \valid(a + (0..(l-1)));
3      requires \forall integer i, j; (0 <= i <= j < l ==> a[i] <= a[j]);
4
5      assigns \nothing;
6
7      behavior present:
8        assumes \exists integer i; (0 <= i < l && a[i] == x);
9        ensures 0 <= \result < l;
10       ensures a[\result] == x;
11
12     behavior absent:
13       assumes \forall integer i; (0 <= i < l ==> a[i] != x);
14       ensures \result == -1;
15 */
16 int searchInArray(int* a, int l, int x){
17   int k;
18
19   /*@ loop invariant 0 <= k <= l &&
20         \forall integer i; 0 <= i < k ==> a[i] < x;
21       loop assigns \nothing;
22       loop variant l-k;
23   */
24   for(k = 0; k < l; k++){
25     if(a[k] == x)
26       return k;
27     else if(x < a[k])
28       return -1;
29   }
30   return -1;
31 }
```

Fig. 2. Function `searchInArray` looks for element `x` in sorted array `a` of length `l` and returns the index of this element in `a` if it is found, or -1 otherwise

Frama-C analyzers share a common specification language called ACSL (ANSI/ISO C Specification Language) [23]. Let us show how a C program can be specified in ACSL and proved in the JESSIE plugin of Frama-C.

Example 1. Fig. 1 shows a simple example of function `max` returning the maximum of its two inputs, specified in ACSL. The `ensures` clause provides a postcondition. Here it states that the return value `\result` must be not less than both inputs, and equal to at least one of them. The `assigns` clause specifies the (non local) variables that the function is allowed to modify. All other variables accessible outside the current function call cannot be affected by the function. Thus this clause also expresses a postcondition. In Fig. 1, this clause (line 3) states that the function should not modify any non local variable. Running the proof of this program in JESSIE with the command `frama-c -jessie max.c` proves the program. In other words, it finds a formal mathematical proof that the program meets its specification.

Example 2. Fig. 2 shows the function `searchInArray` specified in ACSL using behaviors. *Behaviors* provide a very convenient notation when it is necessary to specify the function separately in several cases. Common precondition and postcondition can be provided and must be true for all cases, and a default behavior can be used to specify the default case (see [23] for more detail). A behavior

applies to the situation when its `assumes` clause is true, and in this case the postcondition (`ensures` and `assigns` clauses) must be verified. Notice that a behavior's postcondition should be true in addition to the common postcondition.

In Fig. 2, the common precondition for all behaviors includes lines 1–3. Line 1 states that the array size `l` is not negative. Line 2 specifies that the memory locations `a[0]`, ..., `a[l-1]` are valid, that is, the program can access them. Line 3 specifies that the array `a` is sorted. The common `assigns` clause at line 5 states that the function should not modify any non local variable.

Two behaviors describe separately the case when the element `x` is present in `a` (lines 7–10) and the case it is absent (lines 12–14). Behavior `present` describes the presence case defined by line 8. In this case the postconditions of lines 9–10 must be true. Similarly, the second behavior `absent` describes the absence case defined by line 13. In this case the postcondition of line 14 must be satisfied.

In order to help JESSIE to prove programs in presence of loops, specific clauses for loops may be used, such as `loop invariant`, `loop variant` and `loops assigns`. Here, lines 19–20 indicate a *loop invariant* that must be true before the loop and after any loop iteration. Line 21 indicates variables that can be modified after a number of iterations. Line 22 provides a *loop variant* V, that is, a positive, integer expression strictly decreasing after any loop iteration. It allows the tool to prove loop termination (since the positive integer V cannot decrease infinitely).

3 The Anaxagoros microkernel and hypervisor

3.1 Introduction to systems security

The basic design principles to build a secure operating system have been put forth by Saltzer and Schroeder in the famous paper "The Protection of Information in Computer Systems" [1]. The idea is to build the system as a set of small components called *domains*, isolated from one another, and with tightly-controlled communication between the components. This design brings several benefits:

- inspecting the security in one domain is easier (because it is smaller),
- it is easy to inspect the impact of the loss of one domain on the security of the system,
- the loss of one domain does not weaken the security of other domains.

A good analogy would be the design of a medieval castle: if you only have one level of fence around your large castle, then a single breach looses the entire castle to the opponent. A well-designed castle has several level of fences, with isolated small regions in the castle, and tightly-controlled pathways between the regions, to ease resisting to an attacker.

The eight design principles for designing secure systems are:

Separation of privilege: The idea is to build the system as a set of separate domains with different *privileges*, or *rights*. The domain that may have the

right to read from the network, while a second will have the right to write to a secret file; both domains must be involved to compromise the file with a remote attack.

Least common mechanisms: We call the *trusted computing base* (TCB) of a domain the set of code on which the domain depends to operate properly. The goal of the *least common mechanism* principle is to minimize the TCB of each task in the system. Indeed, a shared domain is an opportunity to breach isolation between several domains; furthermore, an error in shared code may affect several, or even all domains.

There are two main ways to interpret this principle:

- *Microkernel* systems split base *services*, such as network or hard drive access, into isolated domains; so that failure or compromission of a base service affect only the domains that need it. This decrease the TCB for each task.
- *Exokernel* systems and hypervisors decrease the amount of code in base services to put it into the upper layer code. This globally decrease the size of shared system code.

These two ways are not incompatible; for instance, Anaxagoros follows both ways.

Complete mediation: Every access to every resource or object is a privileged operation, and this privilege must be checked, i.e. there should be access control for all objects. For instance, writing to a file, reading a network packet, receiving keyboard inputs, displaying data on the screen, are all privileged operations, and the system must check that the domain that tries to perform these operations has sufficient privileges. A hidden benefit of this principle is to indentify all privileged operations, and the domain that may perform them.

Principle of least privilege states that domains should be given only the privilege of the actions they need to accomplish. The program that displays PDFs on the screen need not access the network, and so is not given this right; thus its eventual compromission does not lead to divulge secret files on the Internet. This principle limits the impact of a compromission of a domain.

Fail-safe defaults: It basically states that "whatever is not explicitly allowed, is forbidden". Forgetting to give enough privilege to a program will only prevent it to work, a problem that will be quickly found; forgetting to remove a privilege leads to a potential security breach that can remain unnoticed.

Economy of mechanism: It states that the design of the system should be as simple as possible. This simplifies reviews. Especially the mechanism for access control should be simple, as the security of the whole system relies on it. As a result, there are two main systems of access control:

- *Access control lists*, where to each resource is associated a list of the domains that can access this resource;
- *Capabilities*, where to each domain is associated the list of accessible resources.

Open design opposes "security through obscurity". The security of the system should not rely on the fact that some features are known only by the team that designed it.

Psychological acceptability states that if the security rules in a system are too complex to use or understand, they are likely to be not applied.

3.2 Anaxagoros

Anaxagoros [2,3] is a secure microkernel that is also capable of virtualizing pre-existing operating systems, for example Linux virtual machines. It is capable of executing hard real-time tasks or operating systems, for instance the PharOS real-time system [24], securely with non real-time tasks, on a single chip.

This goal has required to put a strong emphasis on security in the design of the system, and not only on traditional "behavioral" security (isolation and access control to protect confidentiality and integrity, as presented in the previous section) but also on availability (being able to slow down or steal resources from another task is considered a breach in security).

As it is a microkernel, Anaxagoros is the only piece of code that requires to run in the privileged mode of the CPU in an Anaxagoros-based system. Every piece of code that can be moved out of the kernel is placed in a separated user-level *service*, with limited rights.

This approach contributes to TCB minimization in two ways:

- first, the kernel, which is the only globally trusted piece of code, is minimized,
- second, as services are isolated, their faults do not affect applications that do not require them. For instance, a bug in a network stack would not affect a task that does not use the network.

For safety and concurrency reasons [2], the interface of the kernel and the main user services is low-level, close to the hardware (this is contrary to other microkernel approaches which attempt to provide a generic interface that abstracts the hardware). This approach also allows to classify Anaxagoros as an exokernel [4] or as an hypervisor.

This approach also contributes to TCB minimization: as the interface provides no abstraction, the code of the kernel and services becomes much simpler, as it only has to check that the required hardware operations are permitted.

The kernel generally strictly enforces Saltzer and Schroeder behavioral security principles [1]. In addition to minimizing TCB, the kernel provides protection domains using the machine's virtual memory mechanisms and controls access to shared services using capabilities.

The kernel and services are designed to prevent availability attacks, which are a problem often ignored in conventional system design. In particular the *denial of resources* attack can be made when a task can issue requests that make the kernel or a service allocate a resource (e.g. memory): by issuing a sufficient number of requests, the system can run out of memory. New resource security mechanisms and principles have been built in Anaxagoros to avoid this kind of attack (for instance the kernel does not allocate any memory, while still allowing dynamic creation of new tasks and virtual machines).

3.3 The virtual memory system of Anaxagoros

A critical component to ensure security in Anaxagoros is its *virtual memory system* [3]. The x86 processor (and many other high-end hardware architecture) provide a mechanism for *virtual memory translation*, that translates the address manipulated by a program to real address. One of the goals of this mechanism is to help organizing the program address space, for instance to allow a program to access big contiguous memory regions.

The other goal is to control the memory that a program can access; we will be focusing on that part. The physical memory is split into same-sized region, called *pages* (pages are of size 4kb on standard x86 configurations).

Anaxagoros does not decide what is written to pages; rather, it allows tasks to perform any operations on pages, provided that this does not affect the security of the kernel itself, and of the other tasks in the system.

To do that, it ensures only two simple properties. The first is that a program can only change the page that it "owns". We will not explain here how ownership is represented or checked, and rather concentrate on the second property stating that pages are used according to their types.

Indeed, the hardware mechanism works as follows: a page p is accessible if a special register b points to a page p_d, that points to a page p_t, that points to p (pointing to x means containing a pointer to x in a special format, that we call a *mapping* to p). If a page p_d is pointed by b, we say that p_d is used as a *page directory;* if p_t is pointed by a page directory, we say that it is used as a *page table*. If p is accessible, we say that it is used as a *data page*. The program can write directly to any accessible pages. To write to the other pages, it sends a request to the virtual memory algorithm in the kernel, that checks the request and performs the writing operation if it is allowed.

The key point here is that the hardware does not prevent a page table or page directory to be also used as a data frame. Thus if nothing is done, a task can change the mappings in any page table or page directory it owns. By doing the right modifications, it can access (and write to) any page, including those that it does not own.

3.4 The memory system algorithm: an overview

The goal of the algorithm we are presenting (and verifying) is to prevent these unauthorized modifications. It works by recording:

- The type of the page (the basic types are `zero`, `data`, `pagetable` and `pagedirectory`).
- The number of times the page is being used as a data page, page table, or page directory.

The types are used to ensure the following rules:

Rule 1 Only pages of type `pagedirectory` can be used as page directories;
Rule 2 Only pages of type `pagetable` can be used as page tables;
Rule 3 Only pages of type `data` can be used as data pages.

The kernel ensures these rules by checking requests for changes of page table and page directory entries, so that page directories can only point to pages of type `pagetable`, and pagetables can only point to pages of type `data`. Other requests are denied.

In other words, the algorithm ensures that pages can be used only according to their role. With these rules, we know that page directories cannot be used as data pages, and thus cannot be changed directly by the program, preventing unauthorized modifications.

Now, we allow dynamic reuse of memory, meaning that a page once used as a data page can later be used as page directory. To allow that, the type of the page has to change. But the kernel cannot allow arbitrary change of type, otherwise several types of attacks are possible, and that would lead to the first three rules becoming false.

For instance, a program can access any page p (including those it does not own) by changing a data page p' to contain a mapping to p, change the type of p' to `pagetable`, then use p' as a page table. To prevent this attack, the page p' must be cleaned by the kernel before changing the type. This is the role of the type `zero`: when the kernel receive a request to change the type of a page to `zero`, it first cleans up the contents of that page.

Rule 4 Pages can change their types only from, and to the type `zero`
Rule 5 Pages of type `zero` are filled with zeros, and thus do not point to other pages.

These rules are not sufficient, and other kinds of attacks to access p are possible, by having a page p' used simultaneously as a data page and as a page table:

- Either p' is used as a data page (of type `data`), then cleaned and changed to type `zero`, to type `pagetable`, and used as pagetable;
- Or it is used as a page table (of type `pagetable`), then cleaned and changed to type `zero`, to type `data`, and used as data page.

After both situations, even if p' has been cleaned, nothing prevents the attacker to directly add mappings to p in p'. To prevent these attacks we use a "number of mappings" counter for each page:

Rule 6 The "number of mappings" counter of a page of type `data` is equal to the number of mappings to this page in pages of type `pagetable`
Rule 7 The "number of mappings" counter of a page of type `pagetable` is equal to the number of mappings to this page in pages of type `pagedirectory`
Rule 8 The "number of mappings" counter of a page of type `pagedirectory` is equal to the number of mappings to this page in the b register (at most one for monoprocessor systems).
Rule 9 A page of type `zero` is not used as data page, page table, or page directory, i.e. its number of mappings is 0.

The kernel enforces these rules by denying requests to clean pages to type zero when their "number of mappings" counter is not zero, and by adjusting the "number of mappings" counter every time a pointer to a page is added or removed.

When these checks are present, all the above rules hold whatever the requests from the tasks. Formal proof that these rules are fulfilled by the algorithm is illustrated in Section 4.

3.5 The actual algorithm

The algorithm presented above is simplified: the actual algorithm present additional cases. In particular:

1. There are two kinds of mappings: read-only and writable. Only writable mappings need to be accounted for in this algorithm, but we also have a "number of readable mappings" counter that has other uses. For instance ensuring that there are no more mappings to a page when it changes its owner ensures absence of communication between the previous and the new owner.
2. Pages of type `pagedirectory` and `pagetable` can be used as data pages, but only in read-only mappings. Also, pages of type `pagedirectory` can be used as page directories and as page tables (i.e. they can be pointed by other pages of type `pagedirectory`, or by themselves). The "number of mappings" meaning for pages of type `pagedirectory` is changed: it is equal of the number of times it is pointed by a b register plus the number of times it is pointed by pages of type `pagedirectory`.
3. Cleaning a page is a long operation that can be interrupted. If a page cleanup is interrupted, the page's type is set to a special "partially cleaned" type, and the number of entries cleaned up so far is recorded. Types of page `data` must have a "number of mappings" counter equal to zero at the beginning of the cleanup, otherwise the page contents could be changed during the cleanup. But pages of type `pagedirectory` and `pagetable` can be cleaned up while they are still being used, because the kernel can refuse any concurrent modification. If the "number of mappings" of a page table or a page directory is non zero at the end of a cleanup, the page stays with a "partially cleaned" type, so that there are no active mappings to pages of type `zero`.

4 Anaxagoros verification

The purpose of our ongoing work is the formal verification of the Anaxagoros hypervisor. Our approach is based on the specification of the code in ACSL [23] and proving it using the Frama-C plugins jessie and wp for proof of programs. We are currently working on the proof of the virtual memory management module, one of the most critical modules.

In this section we show how critical system C code can be specified and formally verified using proof of programs. We illustrate it on a partial simplified version of Anaxogoros virtual memory module given in the Appendix. This extract focuses on cleaning `data` pages with interruptions and resuming at the right place. It takes into account the "number of mappings" counter for `data` pages (cf Sec. 3.3, 3.4).

The types `DToZero`, `PTToZero` and `PDToZero` (line 4) are assigned respectively to pages of types `Data`, `Pagetable` and `Pagedirectory` in cleanup, i.e. for which cleaning has started but has not yet terminated (it can be in progress or interrupted, cf Sec. 3.5.3). In this simplified version, the size and maximal number of pages are limited (lines 1–2), and their contents and attributes are represented by arrays (lines 5–10). `Cleaning[p]` specifies if the page `p` is being cleaned now, and `Cleaned[p]` indicates how many elements from the beginning of the page have been already cleaned if the cleaning has been interrupted.

Lines 13–37 define some predicates that will be used in the specification. For instance, `R9` (lines 26–27) states that there are no mappings to a page of type `Zero` (cf Rule 9), while `R5` (lines 28–29) states that a page of type `Zero` is filled with zeros (cf Rule 5). The predicate `ToZeroPagesStartWithZeros` (lines 30–36) specifies that the first `Cleaned[p]` elements of a page `p` in cleanup are filled with zeros, with no valid mappings. Line 37 defines the global invariant.

Page cleaning can be started by `CleanData` (lines 97–107), or resumed by `CleanPartiallyCleanedData` (lines 129–137). After necessary checks, they call `putOnePageToZero` (lines 57–76) that cleans the page from a given position. At each iteration, it checks for interruptions (line 67) modeled in this simplified version by a global variable (line 11).

This example shows that formal program specification takes more than 80% in the resulting specified C code, and writing specification represents very significant effort. Function contracts are the most important part of it, but proving the program with JESSIE may require writing additional clauses, for instance, for loop iterations (cf lines 59–64), or intermediate assertions that help the prover (about 15 lines not presented here).

100% of the 489 proof obligations generated for the code of the Appendix are proved by JESSIE (using JESSIE version 2.29, FRAMA-C Carbon version, and the provers Alt-Ergo version 0.93 and/or Simplify version 1.5.4).

5 Related work and discussion

A recent work [25] presented formal verification for the OS microkernel seL4, allowing devices running seL4 to achieve the EAL7 evaluation level of the Common Criteria [26]. Another formal verification of a microkernel was presented in [27]. In both cases, the verification used interactive, machine-assisted and machine-checked proof with the theorem prover Isabelle/HOL. Although interactive theorem proving requires human intervention to construct and guide the proof, it has the benefit to serve a general range of properties and is not limited

to specific properties treatable by more automated methods of verification as static analysis or model checking.

The formal verification of a simple hypervisor [28] used VCC [29], an automatic first-order logic based verifier for C. The underlying system architecture was precisely modeled and represented in VCC, where the mixed-language system software was then proved correct. Unlike [25] and [27], this technique was based on automated methods.

[30] reports on verification of TLB (translation lookaside buffer) virtualization, a core component of modern hypervisors. Because devices run in parallel with software, they necessitate concurrent program reasoning even for single-threaded software. The authors give a general methodology for verifying virtual device implementations, and demonstrate the verification of TLB virtualization code in VCC.

Formal verification nowadays remains rather costly. According to [31], the cost of the verification of the seL4 microkernel was around 25 person-years, and required highly qualified experts. seL4 contains only about 10,000 lines of C code, and verification cost is about $700 per line of code.

6 Conclusion and future work

Recent advances in formal verification and security engineering have shown that it is now possible to perform a formal machine-checked proof of a complete microkernel. In this paper, we presented a short tutorial on proof of programs with Frama-C, described the Anaxagoros hypervisor with its security principles, and illustrated on the Anaxagoros virtual memory system how critical system code can be specified and formally verified using modern verification tools.

There are still many interesting challenges to be addressed. An important future work perspective is the verification of a concurrent hypervisor. For instance, the verification in [25,28] was carried out for a sequential version. This research direction is extremely important for an OS or a hypervisor since concurrency naturally appears both for parallel execution on a multi-core architecture and for non-deterministic interleaving via threads on a unique processor. We expect that such verification may require the development of new algorithms and specifications, adapted for the proof of a concurrent version, in particular for the execution on multi-core processors.

Future work also includes an extension of the verification to complex mixed software and hardware designs in order to avoid that a hardware failure alters the expected behavior of a verified hypervisor.

Whatever particular verification approach is used, formal verification of a microkernel or a hypervisor represents a great effort and remains valid only for a particular version being verified. Therefore, any evolution of the software requires new verification. To allow industrial usage of formally verified system software in a real-life environment, the verification of a new version should require only a limited effort, without performing a new specification and proof of the whole system. Another important future work direction is developing formal verification

methodologies for modular proof such that any evolution has a limited impact on the verification.

References

1. Saltzer, Schroeder: The protection of information in computer systems. Communication of the ACM **7** (1974)
2. Lemerre, M., David, V., Vidal-Naquet, G.: A communication mechanism for resource isolation. In: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems. IIES'09, New York, NY, USA, ACM (2009) 1–6
3. Lemerre, M., David, V., Vidal-Naquet, G.: A dependable kernel design for resource isolation and protection. In: IIDS '10: Proceedings of the First Workshop on Isolation and Integration in Dependable Systems, ACM (2010) 1–6
4. Engler, D.R., Kaashoek, M.F., J. O'Toole, J.: Exokernel: an operating system architecture for application-level resource management. In: Proceedings of SOSP '95. (1995) 251–266
5. Legout, V., Lemerre, M.: Paravirtualizing linux in a real-time hypervisor. In: Embed with Linux (EWili) 2012. (2012)
6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A program analysis perspective. In: Proc. of the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012), Springer (2012) 233–247
7. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C User Manual. (October 2011) `http://frama-c.com`.
8. Kosmatov, N.: PathCrawler online (2010–2012) http://pathcrawler-online.com/.
9. Kosmatov, N., Williams, N., Botella, B., Roger, M., Chebaro, O.: A lesson on structural testing with pathcrawler-online.com. In: TAP'2012. Volume 7305 of LNCS., Prague, Czech Republic, Springer (May 2012) 169–175
10. Hoare, C.A.R.: An axiomatic basis for computer programmation. Communication of the ACM **12**(10) (1969) 576–580
11. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. CACM **18**(8) (1975) 453–457
12. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52** (May 2005) 365–473
13. The Alt-Ergo theorem prover. `http://ergo.lri.fr/`
14. Conchon, S., Contejean, E., Kannig, J., Lescuyer, S.: Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In Rushby, J., Shankar, N., eds.: AFM, New York, NY, USA, ACM (2007) 55–59
15. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Volume 4963 of LNCS., Springer (2008) 337–340
16. The Coq proof assistant. `http://coq.inria.fr/`
17. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. SpringerVerlag (2004)
18. Bertot, Y.: A short presentation of coq. In: TPHOLs. (2008) 12–16
19. The Isabelle proof assistant. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
20. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
21. The HOL4 proof assistant. `http://hol.sourceforge.net/`

22. Gordon, M.J.C., Melham, T.F., eds.: Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press (1993)
23. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (February 2011) http://frama-c.cea.fr/acsl.html.
24. Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., Jacques, M.B.: Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In: Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems. (2011)
25. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09), ACM (2009) 207–220
26. The Common Criteria Recognition Arrangement. http://www.commoncriteriaportal.org
27. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive Verification of an OS Microkernel. In Leavens, G., O'Hearn, P., Rajamani, S., eds.: Verified Software: Theories, Tools, Experiments. Volume 6217 of LNCS. Springer Berlin / Heidelberg (2010) 71–85
28. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: Verified software: theories, tools, experiments (VSTTE), Springer (2010) 40–54
29. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., , Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics (TPHOLs). Volume 5674 of LNCS. (2009) 23–42
30. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012), Philadelphia, PA, USA, Springer (January 2012) 209–224
31. Klein, G.: From a verified kernel towards verified systems. In: the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010), Shanghai, China, Springer (November 2010) 21–33

Appendix. A simplified version of data page cleaning

```
1  #define MaxNumPages 100
2  #define PageSize 10
3  int NumPages; // number of pages
4  enum pageType {Zero,Data,Pagetable,Pagedirectory,DToZero,PTToZero,PDToZero};
5  unsigned int Contents[MaxNumPages * PageSize];
6  unsigned char Valid[MaxNumPages * PageSize];
7  enum pageType Type[MaxNumPages];
8  unsigned int Mappings[MaxNumPages];
9  unsigned char Cleaning[MaxNumPages];
10 unsigned int Cleaned[MaxNumPages];
11 int pending_preemption; // interruption iff non zero value
```

```
/*@
  predicate zero_Contents{L}(integer pIndex) =
    \forall integer k; 0<=k<PageSize ==> Contents[pIndex*PageSize+k]==0;
  predicate structures_validity = 0<=NumPages<=MaxNumPages &&
    \valid( Contents + (0.. (NumPages*PageSize -1) ) ) &&
    \valid( Valid + (0.. (NumPages*PageSize -1) ) ) &&
    \valid(Type + (0..NumPages-1)) && \valid(Mappings + (0..NumPages-1)) &&
    \valid(Cleaning + (0..NumPages-1)) && \valid(Cleaned + (0..NumPages-1)) &&
    ( \forall integer j; 0<=j<NumPages ==> (Type[j]==Zero || Type[j]==Data ||
      Type[j]==Pagetable || Type[j]==Pagedirectory || Type[j]==DToZero ||
      Type[j]==PTToZero || Type[j]==PDToZero) ) &&
    ( \forall integer j; 0<=j<NumPages ==> 0<=Cleaned[j]<=PageSize ) &&
    ( \forall integer j,l; 0<=j<NumPages && 0<=l<PageSize ==>
      0<=Valid[j*PageSize+l]<=1 );
  predicate R9 =
    \forall integer j;  ( 0<=j<NumPages && Type[j]==Zero ) ==> Mappings[j]==0 ;
  predicate R5 =
    \forall integer j;  ( 0<=j<NumPages && Type[j]==Zero ) ==> zero_Contents(j) ;
  predicate ToZeroPagesStartWithZeros =
    ( \forall integer j;  ( 0<=j<NumPages && (Type[j]==DToZero ||
      Type[j]==PTToZero || Type[j]==PDToZero) ) ==>
      (\forall integer i; 0<=i<Cleaned[j] ==> Contents[j*PageSize+i]==0) ) &&
    ( \forall integer j;  ( 0<=j<NumPages && (Type[j]==DToZero ||
      Type[j]==PTToZero || Type[j]==PDToZero) ) ==>
      (\forall integer i; 0<=i<Cleaned[j] ==> Valid[j*PageSize+i]==0) );
  predicate Inv = structures_validity && R9 && ToZeroPagesStartWithZeros && R5;
*/

/*@
  requires Inv && 0<=pIndex<NumPages && 0<=startIndex<=PageSize &&
    ( \forall integer k; 0<=k<startIndex ==> Contents[pIndex*PageSize+k]==0 ) &&
    Cleaning[pIndex]==1 && Mappings[pIndex]==0 && Type[pIndex] == DToZero;
  behavior finished:
    assumes pending_preemption == 0;
    ensures Inv && Type[pIndex]==Zero && Mappings[pIndex]==0 &&
      Cleaning[pIndex]==0 && Cleaned[pIndex]==0;
    assigns Contents[(pIndex*PageSize+startIndex) .. (pIndex*PageSize +
      PageSize-1)], Type[pIndex], Cleaning[pIndex], Cleaned[pIndex];
  behavior interrupted:
    assumes pending_preemption != 0;
    ensures Inv && Type[pIndex]==DToZero && Mappings[pIndex]==0 &&
       Cleaning[pIndex]==0 && startIndex < Cleaned[pIndex] <= PageSize;
    assigns Contents[(pIndex*PageSize+startIndex) .. (pIndex*PageSize +
      PageSize-1)], Type[pIndex], Cleaning[pIndex], Cleaned[pIndex];
*/
void putOnePageToZero(int pIndex, int startIndex){
  int l;
  /*@ loop invariant startIndex <= l <= PageSize && Inv &&
        ( \forall integer k; 0<=k<l ==> Contents[pIndex*PageSize+k]==0 );
      loop assigns Contents[(pIndex*PageSize+startIndex) .. (pIndex*PageSize +
        PageSize-1)];
      loop variant PageSize - l;
  */
  for(l=startIndex; l<PageSize; l++){
    Contents[pIndex*PageSize + l] = 0;
    if(pending_preemption){
      Cleaned[pIndex]=l+1;
      Cleaning[pIndex]=0;
      return;
    }
  }
  Type[pIndex] = Zero;
  Cleaning[pIndex]=0;
  Cleaned[pIndex]=0;
}
```

```
77
78 /*@
79   requires Inv && 0<=pIndex<NumPages;
80   behavior finished:
81     assumes pending_preemption == 0 && Type[pIndex]==Data && Mappings[pIndex]==0;
82     ensures Inv && \result==0 && Type[pIndex]==Zero &&
83       Cleaning[pIndex]==0 && Cleaned[pIndex]==0;
84     assigns Contents[(pIndex*PageSize) .. (pIndex*PageSize + PageSize-1)],
85       Type[pIndex], Cleaning[pIndex], Cleaned[pIndex];
86   behavior interrupted:
87     assumes pending_preemption != 0 && Type[pIndex]==Data && Mappings[pIndex]==0;
88     ensures Inv && \result==0 && Type[pIndex]==DToZero &&
89        Cleaning[pIndex]==0 && 0 < Cleaned[pIndex] <= PageSize;
90     assigns Contents[(pIndex*PageSize) .. (pIndex*PageSize + PageSize-1)],
91       Type[pIndex], Cleaning[pIndex], Cleaned[pIndex];
92   behavior failure:
93     assumes Type[pIndex]!=Data || Mappings[pIndex]!=0;
94     ensures Inv && \result==1;
95     assigns \nothing;
96 */
97 int cleanData (int pIndex){
98   if(Type[pIndex] != Data)
99     return 1;
100  if(Mappings[pIndex] != 0)
101    return 1;
102  Cleaning[pIndex]=1;
103  Cleaned[pIndex]=0;
104  Type[pIndex] = DToZero;
105  putOnePageToZero(pIndex,0);
106  return 0;
107 }
108
109 /*@
110   requires Inv && 0<=pIndex<NumPages;
111   behavior finished:
112     assumes pending_preemption == 0 && Type[pIndex]==DToZero && Mappings[pIndex]==0;
113     ensures Inv && \result==0 && Type[pIndex]==Zero &&
114       Cleaning[pIndex]==0 && Cleaned[pIndex]==0;
115     assigns Contents[(pIndex*PageSize+Cleaned[pIndex]) .. (pIndex*PageSize +
116       PageSize -1)],Type[pIndex],Cleaning[pIndex],Cleaned[pIndex];
117   behavior interrupted:
118     assumes pending_preemption != 0 && Type[pIndex]==DToZero && Mappings[pIndex]==0 &&
119       Cleaning[pIndex]==0;
120     ensures Inv && \result==0 && Type[pIndex]==DToZero &&
121       Cleaning[pIndex]==0 && \old(Cleaned[pIndex]) < Cleaned[pIndex] <= PageSize;
122     assigns Contents[(pIndex*PageSize+Cleaned[pIndex]) .. (pIndex*PageSize +
123       PageSize -1)],Type[pIndex],Cleaning[pIndex],Cleaned[pIndex];
124   behavior failure:
125     assumes Type[pIndex]!=DToZero || Mappings[pIndex]!=0;
126     ensures Inv && \result==1;
127     assigns \nothing;
128 */
129 int cleanPartiallyCleanedData (int pIndex){
130   if(Type[pIndex] != DToZero)
131     return 1;
132   if(Mappings[pIndex] != 0)
133     return 1;
134   Cleaning[pIndex] = 1;
135   putOnePageToZero(pIndex,Cleaned[pIndex]);
136   return 0;
137 }
```