A Contribution to Techniques for Building Dependable Operating Systems

Matej Košík[*]
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 3, 842 16 Bratislava, Slovakia
kosik@fiit.stuba.sk

Abstract
Even though development of a new operating system is rarely a good (business) plan, in a small scale it could be a fruitful experiment in software engineering. In this document we describe results of our effort to build a dependable operating system. By adopting and building upon recent advances in the programming language design, we show one possible way how can we reduce development costs of dependable (operating) systems. During the course of our work, we focused primarily on aspects that cannot be added as an afterthought—e.g. dependability of the operating system—rather than those which can be added anytime in the future—e.g. rich services of the operating system.

Categories and Subject Descriptors
D.2.0 [**Software engineering**]: Protection mechanisms; D.4.5 [**Operating systems**]: Reliability—*Fault-tolerance*; D.4.6 [**Operating systems**]: Security and protection—*Security kernels*; F.3.2 [**Logics and meanings of programs**]: Semantics of programming languages

Keywords
software-engineering, dependability,
programming languages, operating systems

1. Introduction
When developing software systems, whose malfunction can cause losses of human lives, a failure of an important mission, or a loss of serious amount of money, then we must ensure that our system has the following properties:

- *Safety*: the damage of external entities is impossible.
- *Confidentiality*: undesired information leaks are impossible.
- *Correctness*: correct services to external entities are provided all the time.
- *Robustness*: a failure of any component must not immediately mean that the whole system will fail.
- *Integrity/Security*: external entities cannot damage our system.
- *Responsiveness*: services of the system are delivered in a timely fashion all the time.
- *Self-healing*: if some part of the system fails, and this failure is "transient", there is a mechanism which is able to restart the failed part.

These properties cannot be added to a software system as another feature in a similar way how modularity cannot be added as an afterthought [38]. They can, however, gradually emerge as we work on the project. In this paper we show what can we do to achieve these properties if we are working on an operating system. We do this by using relevant *tools* to obey relevant *practical principles* to achieve above abstract goals, see Figure 1.

2. Terminology

Conventional. We use terms subject, object, operation, permission and trusted computing base (TCB) in concord with their traditional meaning from computer security literature. *Subjects* are active entities (e.g., UNIX processes) with some behavior. Subjects can designate *objects* and usually try to perform some supported *operations* on them. The set of operations that can be performed on some object depends on its type. The set of existing objects and subjects typically changes over time. *Permissions* is a relation that defines which operations on what objects are permitted for particular subjects. The *trusted computing base* (TCB) of a computer system is the set of all hardware, firmware, and/or software subsystems that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system.

[*]Recommended by thesis supervisor: Prof. Jiří Šafařík
Defended at Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava on September 30, 2011.

Košík, M. A Contribution to techniques for building dependable operating systems. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 3, No. 4 (2011) 1-11

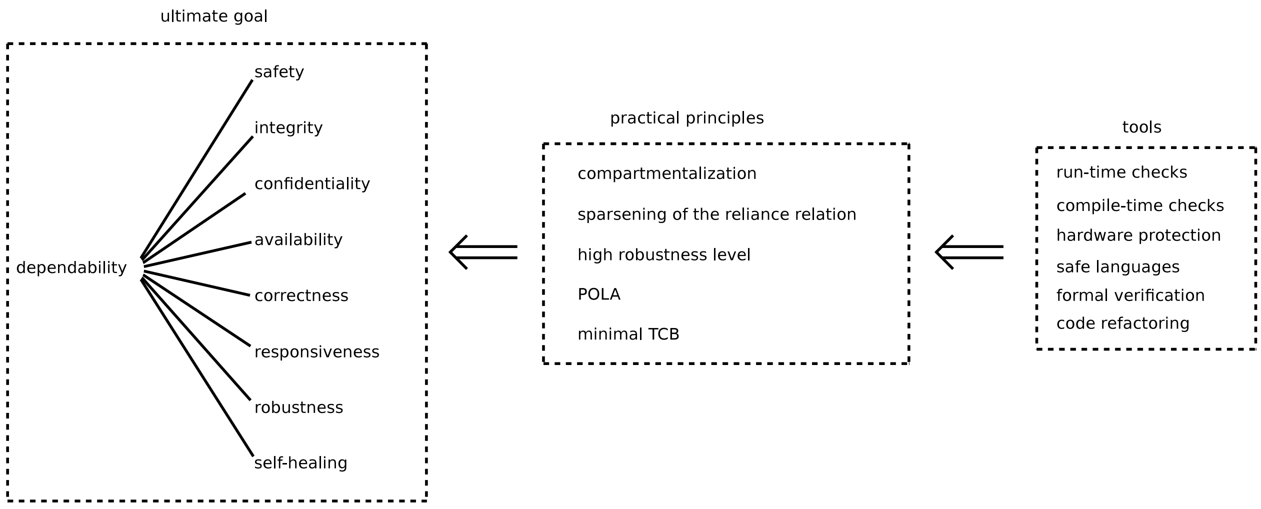


Figure 1: Dependability, as an ultimate goal, can be reached by following a set of practical principles. These principles can be obeyed if we wield with the right tools.

Unconventional. A *capability* is a token that identifies an object and provides its holder with the permission to operate on the object it identifies. Capabilities must either be totally unforgeable[1] or infeasible to forge[2] The *authority* of a subject is the set of all the ways how this subject can affect its environment. The *principle of least authority* (POLA) is a habit, where we ensure that each subject has exactly as much authority, as it needs to have in order to provide all the expected services[3]. If the correct operation of some subject depends on the correct operation of some object, we say that the first subject *relies upon* the given object. In some cases subject S relies on object O because S has a permission to invoke certain set of operations with O and S also tries to invoke those operations. In that case, S assumes that O implements them correctly. *Fragile system* is as reliable as its least reliable subsystem, as secure as its most severe security flaw and as safe as its most malicious subsystem. We say that given system is *defensively consistent*, if misbehaving or malicious client cannot force the server to provide incorrect service for any of its clients. We say that given system is *defensively correct*, if misbehaving or malicious client cannot force the server to stop providing correct service for all its well-behaving clients. *Fragility*, *defensive consistency* and *defensive correctness* are three distinct levels of *robustness* [37].

3. Thesis objectives

We have decided to build an operating system that:

- has smaller *trusted computing base* than any other operating system,
- its components follow *principle of least authority*,
- its *reliance relation* is as sparse as possible,
- and the whole system is *defensively correct.*

We focused on these properties first because they cannot be added later as an afterthought [38], in contrast with various other requirements.

4. Employed practical principles

Dependability, with all its individual aspects, is an abstract goal which can be achieved if we simultaneously follow the following practical principles (see Figure 1):

- *Sparsening of the reliance relation*, *POLA* and *high robustness level* minimize the impact of errors, flaws or malicious behavior inside particular compartments.
- *Compartmentalization* means that the system is split to multiple compartments that can interact only in a controlled way. This makes development process of dependable systems scalable.
- *Minimal TCB* increases our chance to deliver correct TCB. An incorrect TCB may negate our compartmentalization effort.

4.1 Sparsening of the reliance relation

One of the basic ways how can we perceive the structure of the system is to use so called *reliance relation* or *reliance graph.* Individual subsystems $S_1, S_2, \ldots, S_n$ of the whole system are vertices of the graph. Its edges $S_i \to S_j$ mean that subsystem S_i provides correct services as long as subsystem S_j provides correct services.

If *reliance graph* is too similar to a complete graph, that is a sure sign of system's fragility. This is very common and it is typically caused by missing or imperfect *compartmentalization.*

For robust systems, we can ensure that the *reliance graph* is a partial order and we can try to make this graph as

[1]For example, references to objects in Caja, which is a retrofitted version of JavaScript, are capabilities that cannot be forged.
[2]Long URIs sometimes play the role of a capability. They cannot be guessed, but those who have them can access designated objects, e.g., Google Docs, Google Maps, Picasa albums, Doodle schedulers etc.
[3]We avoid a more common term *principle of least privilege (POLP)* because it reinforces a common fallacy that it is sufficient to focus on permissions, which is easy but insufficient, rather than focusing on the authority, which is hard but essential.

sparse as possible. Sparsity has the following positive effects:

- It minifies *reliance set* of individual subsystems. It is formed by all subsystems that must provide correct services before the chosen subsystem can provide correct services.
- It magnifies *non-reliance set* of individual subsystems. It is formed by all subsystems that can provide correct services even though the chosen subsystem provides incorrect services.

4.2 High robustness level

There are various levels of robustness [37]. Let us describe the two most interesting ones.

4.2.1 Defensively consistent processes

We say that given system is *defensively consistent*, if misbehaving or malicious client cannot force the server to provide incorrect service for any of its clients [37]. There are platforms where it is possible to determine possible interaction of processes. One of them is pi-calculus. Each process can interact with other processes only by message-passing over channels designated by its free variables. Then, knowing something about channels used by a process, we can determine how can a given process influence its environment. This is called *authority* of that process.

4.2.2 Defensively correct processes

We say that given system is *defensively correct*, if misbehaving or malicious client cannot force the server to stop providing correct service for all its well-behaving clients [37]. All defensively correct systems are also defensively consistent. Some defensively consistent systems can be retrofitted to become defensively correct.

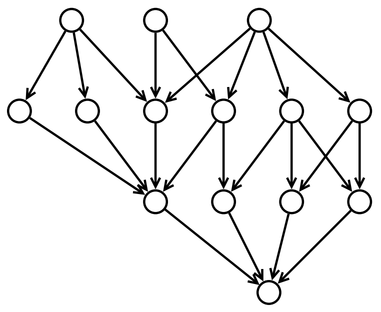
4.3 Principle of Least Authority (POLA)

Each subject typically need some *authority*. The *principle of least authority* postulates that each subject should have minimal authority. Minimal authority enables given subject to provide expected services but it does not enable it to make undesired interactions with its environment. POLA improves both safety and security of individual subsystems as well as of the whole system. The fact that most contemporary platforms do not provide a parsimonious set of versatile mechanisms to obey POLA does not imply that POLA is *in general* a worthless distraction [3, 64].

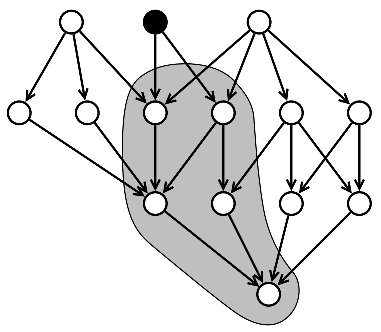
4.4 Minimal trusted computing base

Minimization of the trusted computing base (TCB) represents a realistic stance how to deliver dependable software system despite human fallibility. This engineering principle is not inconsistent with Linus' Law: "given enough eyeballs, all bugs are shallow." It only reinforces its effect. Most of the projects will fail to attract "enough eyeballs" but they can still achieve desired level of dependability by TCB minimization.

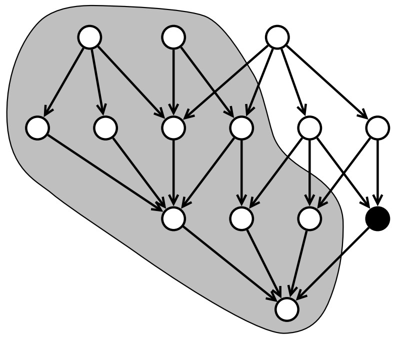
We have examined TCB of two operating systems: Minix and Singularity [17]. Both contains sufficiently strong mechanisms to achieve defensive correctness:



(a) An example a *reliance graph*. It is a partial order so we can use Hasse diagram to define it. $S_i \to S_j$ means that subsystem S_i can provide correct services as long as S_j provides correct services.



(b) A *reliance set* (denoted with gray color) of the chosen subsystem (denoted with black color).



(c) A *non-reliance set* (denoted with gray color) of the chosen subsystem (denoted with black color).

Figure 2: The impact of sparseness of *reliance graph* on the overall robustness of the system.

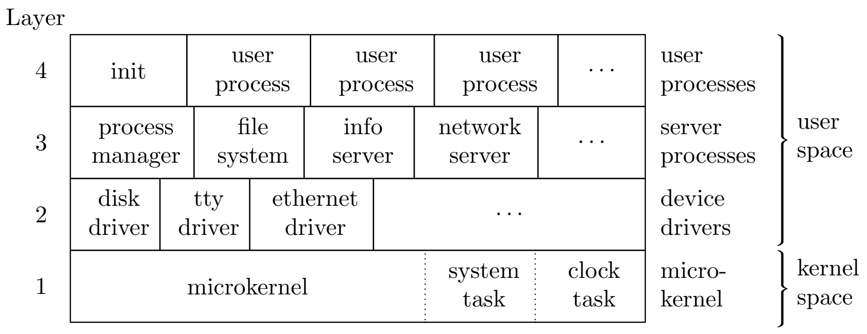


Figure 3: Internal structure of the Minix operating system.

- their subsystems obey POLA,
- we can restrict the memory consumption of their subsystems,
- we can influence CPU priority or CPU cycles that are available for their subsystems.

Minix's trusted computing base has at least 34 404 lines of code[4]. TCB of MS Singularity has at least 150 000 lines of code[5] Due to its large TCB, Singularity can therefore be used [21] as a deceiving argument against using safe programming languages to implement dependable operating systems.

5. Tools

We have a rich set of tools that can be used to obey previously described practical principles. Compartments can be created at least by two different ways. Either by relying on protection mechanisms provided by *hardware* (operating systems with microkernel architecture do this) or by relying on protection mechanisms provided by *safe languages*.

The right set of tools and right goals straight from the beginning are important because safety and security cannot, for complex systems, be achieved as an afterthought [38]. In the next sections we investigate known tools and how can they be used in order to improve dependability.

5.1 Formal verification

Formal verification enables us to prove *correctness*[6] of a system with respect to formal specification. Correctness is one aspect of dependability and this approach is therefore valuable. Novel example of this approach is seL4 microkernel [21, 23]. This microkernel was implemented in Haskell. Its authors then extracted abstract specification from this Haskell implementation. Actual C implementation was then checked against this abstract specification. It is not correct to refer [12, 21] to seL4 as operating system because seL4 is only a microkernel. To create an operating system, this group must implement essential servers such as a file system server, a process manager, device drivers etc. Servers will create usable abstractions of the actual hardware. These abstractions are necessary for development of user space applications. If this group plans to create formally verified device drivers, they will have to create formal specification of relevant hardware devices. They will have to count with the fact that theory (published informal specification of hardware) is often inconsistent with reality [60].

Compilers must be regarded as a part of the trusted computing base. Therefore successful effort to formally prove their correctness enables us to remove them from TCB. Recently, INRIA group created a constructive proof of the correct C compiler [30, 31]. This effort also minimizes the trusted computing base. Instead of the whole C compiler, we can rely only on validity of its formal specification.

Comprehensive overview of projects aiming at formal verification of operating systems can be found in [22]. Comprehensive overview of available proof checkers can be found in [9, 65] and in the foreword of the Coq book [4].

5.2 Hardware-based protection

Hardware protection mechanisms are used by operating systems with microkernel architecture to create compartments. These operating systems typically have a small part that runs in the kernel space (the *microkernel*). Most of the operating system functionality is implemented as a set of separate user space processes.

KeyKOS [5, 11, 29] is the earliest microkernel-based operating system. It was a commercial project originally implemented for System/370 and predates microkernels that later become popular in academia: Mach [6, 47], Minix [59] and L4 [32]. It had interesting properties:

- performance competitive with macrokernel design,
- high levels of security and interval between failures in excess of one year,
- checkpoint at tunable intervals provide system-wide backup,
- fail-over support,
- and system restart times typically less than 30 seconds. On restart, all processes are restored to their exact state at the time of checkpoint, including registers and virtual memory.

[4]It is formed at least by the microkernel, the process manager and libraries linked with the process manager [27].
[5]It is formed by hardware abstraction layer, memory manager, metadata manager, loader, scheduler, channel manager, I/O manager, security manager, MSIL code translators, see Figure 2 in [17].
[6]or in other words *conventional correctness* [37]

KeyKOS inspired several other projects: EROS [52, 53], CapROS [28], Coyotos [51].

Minix3 [59] is a recent attempt to create a dependable operating system. Its compartments rely on hardware protection mechanisms. It is implemented as a set of multiple communicating processes. Only three of those processes run in the kernel space, see Figure 3. Most operating system subsystems are implemented as user space processes. This projects has interesting research goals [58]. In the introduction of the grant proposal Tanenbaum rightly states that:

> *Attacks by viruses and worms are rampant. It is no understatement to say that Windows has very serious reliability and security issues. UNIX derivatives such as Linux are not in the press as much. In part it is because they are slightly more secure, but in reality because hackers tend to focus on hitting Windows, where they can get the most publicity for their attacks.*

This statement is true but Tanenbaum does not describe the whole picture. He states that:

> *Most reliability and security problems ultimately come down to the fact that programmers are human beings and as such are not perfect. While no one disputes this statement, no current systems are designed to deal with the consequences of it. Studies have shown the number of bugs in commercially available software ranges from about 1 bug per 1000 lines of code to 20 bugs per 1000 lines of code (no references are provided in this synopsis but they are present in the full proposal). Large and complex software systems have more bugs per 1000 lines of code due to the larger number of modules and their more complex interactions. Windows XP consists of 50 million lines of code. From this it follows that Windows XP has somewhere between 50,000 and 1 million bugs in it. And Vista is larger still (70 million lines) and probably has even more bugs. The key insight here is: Expect bugs. They are there. Learn to live with them.*

He realizes that we need a better *structure* for operating systems

> *In effect, I am going to make the case that the hundreds of millions of computers in the world are all based on fundamentally flawed system software that cannot be repaired. Future computers need something completely different. This requires a new and different structure of the software. Figuring out how to do this is definitely frontier research.*

but he decided to retain POSIX:

> *Fortunately, this can be done without affecting the user software by simply having the new operating system emulate the existing and long-stable POSIX interface (the UNIX standard), possibly extended somewhat (e.g., with some of the Linux system calls).*

POSIX does not offer satisfactory mechanisms for safe running of untrusted code. Various engineers already realized this and tried to retrofit it, c.f. [14, 40, 49, 50, 63, 66]. Minix cannot avoid a similar fate.

5.3 Language-based protection

In different context the term *safe language* can be interpreted in various ways. In this document we consider given programming language as safe if it has parsimonious formal semantics. That allows us to view programs written in this language as tractable mathematical objects. This excludes "surprising" mismatch between theoretical reasoning about programs and their actual behavior. Semantics of simple programming languages (lambda-calculus [33], sigma-calculus [2], pi-calculus [39] or for example Featherweight Java [44]) can be defined by reduction rules. Rich languages with formal semantics can be easily designed if we can split language constructs into the following three layers:

- *Core language* is a set of orthogonal language constructs with formal semantics.
- *Syntactic sugar* is a set of language constructs that do not extend expressibility of the programming language but improve its usability. Their semantics is defined by mapping them into the core language.
- *Syntactic salt* is a set of language constructs that makes it harder for us to write incorrect programs.

These layers can be found in various contemporary languages: Haskell [16][7], Ocaml [42], Oz [54] etc.

5.3.1 Capability languages

In standard security literature, words: *subject*, *object*, *permission*, *designation*, *capability* have special meaning. In general, *subjects* are active entities that, during their lifetime, may try to perform various (supported) *operations* on *objects* if they are *permitted* to do so. In UNIX, processes are *subjects*, files are *objects*. *Designation*, in this case, is a string that is interpreted as an absolute or a relative pathname. UNIX processes may designate objects (files) and try to perform some supported operations with these objects:

- to open given file for reading,
- to open given file for writing,
- to execute given file as a program.

Somewhere in the system are stored *permissions* that define which attempts will succeed and which attempts will fail.

[7]Formal semantics of the core Haskell does not exist but mostly because nobody bothered to write it down. We include Haskell example bona fide.

A designation of an object, that is indivisibly bundled with permissions to make some operations with the designated object, is called *capability*. File descriptors in UNIX can be regarded as capabilities. Each file descriptor *designates* certain file. The holder of some file descriptor is automatically *permitted* to make certain operations with that file.

Capabilities emerged not only in the context of operating systems but also in safe languages. There they designate objects[8] and are bundled with permission to invoke methods[9] of the designated object. It was constructively proved [37] that in order to enable creation of robust systems out of cooperating untrusted components, the following *capability axioms* must hold:

- Capabilities to objects are not forgeable.
- Subjects can interact only with those objects to which they have a capability.
- A subject can get a capability to an object only by: *initial conditions*, *parenthood*, *introduction* or *endowment*.

Often, when some software system starts, some subjects inherently have access to other objects. This phenomenon is called *connectivity by initial conditions*. It can be usually inferred directly from the source code at compile time simply by taking into consideration statical scoping rules. If one subject creates some object, it is automatically given a capability to it. This phenomenon is called *connectivity by parenthood*. It is closely related to calling constructors of objects. *Connectivity by introduction* refers to a possibility to pass a capability to an object in a message to other object. *Connectivity by endowment* is related to abstractions as they are understood in the context of the lambda-calculus or the pi-calculus. Abstractions can refer (with their free variables) to any object that is visible in the context where we literally construct them and abstractions will continue to refer to those objects regardless of the position from which we ultimately invoke them.

E programming language [36] enforces these axioms by design. Several languages can be, by retrofitting, turned into capability languages: W7 [48] is retrofitted Scheme 48 [20], Joe-E [34, 35] is retrofitted Java, Emily [57] is retrofitted Ocaml, Caja [1] is retrofitted JavaScript. By retrofitting we mean making sure that there are no ways how could one violate capability axioms. Languages like LISP or Erlang can be regarded as safe languages but these cannot, without complete redefinition, be retrofitted in this way.

Capability axioms represent the simplest platform from protecting the rest of the system from untrusted subsystems. There are other more complicated schemes [7, 8, 10, 15, 55].

6. The main result

After analyzing the strengths and the weaknesses of an existing programming language Pict [43, 45, 46, 61] we came up with a definition and an implementation of a new programming language P [26] that enables us to achieve declared goals. We show how can we use it to create an operating system with the following properties:

- It has s small *trusted computing base (TCB)* (1988 lines of code).
- It is composed from several compartmentalized subsystem which follow *principle of least authority (POLA)*.
- Its *reliance relationship* among individual subsystems is as sparse as possible.
- The whole system is defensively correct.

It behaves in the following way:

- It cleans the screen (80x25 characters).
- In one portion of the screen it shows the uptime in `HH:MM:SS` format.
- The terminal subsystem handles user's input and ensures proper update of the screen.
  On `Ctrl+Alt+Del` press, it reboots the computer.

A glimpse of the system is in Figure 4. The source code and the complete technical documentation is available [25].

As you can see, the provided service of the operating system is at the moment pathetic. This property can be fixed without breaking the goals declared in Section 3.

6.1 The trusted computing base

Figure 5 indicates the structure of the system. Its TCB is formed by:

- the underlying hardware,
- assembly code (`stage1.S`) that takes control from the bootloader,
- C code (`stage2.c`) that to which assembly code passes control,
- P programming language runtime (`libp.a`) which is linked with the kernel,
- parts of the kernel (`memory.p + io.p + irq.p + prim.p + main.p`) written in an unsafe variant of P[10].

The unsafe variant of P programming language does not have a formal semantics because it is allowed to inline arbitrary C code. This is essential for defining behavior of primitive processes. System's TCB is defined in 1988 lines of code. Trusted hardware and the trusted compilers are excluded from this account.

[8]or functions or procedures or channels

[9]or to call the designated function or to call the designated procedure or to send/receive messages to/from the designated channel

[10]Inlined C code in `prim.p` module is used to define various harmless primitives (arithmetic operations etc.)

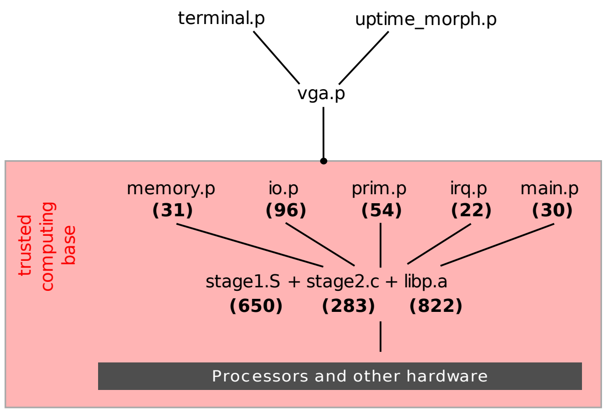


Figure 5: Reliance relationship of the OS defined by a Hasse diagram.

6.2 The reliance relationship

Figure 5 indicates *reliance relationship* of the system. The system is composed from the following subsystems:

- the trusted computing base (TCB);
- `vga`: a VGA driver;
- `uptime_morph`: a subsystem that is able to show uptime in `HH:MM:SS` format on the screen;
- `terminal`: handles user input and updates the screen.

From the figure we can read that:

- `vga` will provide correct services if TCB is correct;
- `terminal` will provide correct services if `vga` and `TCB` are correct;
- `uptime_morph` will provide correct services if `vga` and `TCB` are correct.

The figure is complete; no reliance relationships are concealed. We can therefore also say:

- `vga` cannot force TCB to stop providing the correct service;
- `uptime_morph` cannot force either TCB, or `vga`, or `terminal` to stop providing the correct service;
- `terminal` cannot force either TCB, or `vga`, or `uptime_morph` to stop providing the correct service.

6.3 Compartmentalization

The claims in the previous section are based on the fact that we can run untrusted code so, that unnecessary interaction among individual subsystems are beforehand excluded. This is achieved by relying on the programming language's semantics, particularly:

- statical scoping rules,
- the possibility to enforce memory quotas over untrusted subsystems,
- the possibility to enforce CPU quotas over untrusted subsystems.

For example:

- Even though `memory` module defines and exports functions that can be used for reading or writing to any memory location, none of the untrusted subsystems[11] can, by default, directly call them because these functions are outside their scope.
- Even though `io` module defines and exports functions that can be used for reading or writing any I/O port, none of the untrusted subsystems can, by default, directly call them because these functions are outside their scope.
- Even though `vga` module defines and exports functions that can be used for arbitrary manipulation of the screen, none of the other untrusted subsystems can, by default, directly call them because they are outside their scope.

In other words, each untrusted subsystem is started with a minimal (i.e. harmless) authority.

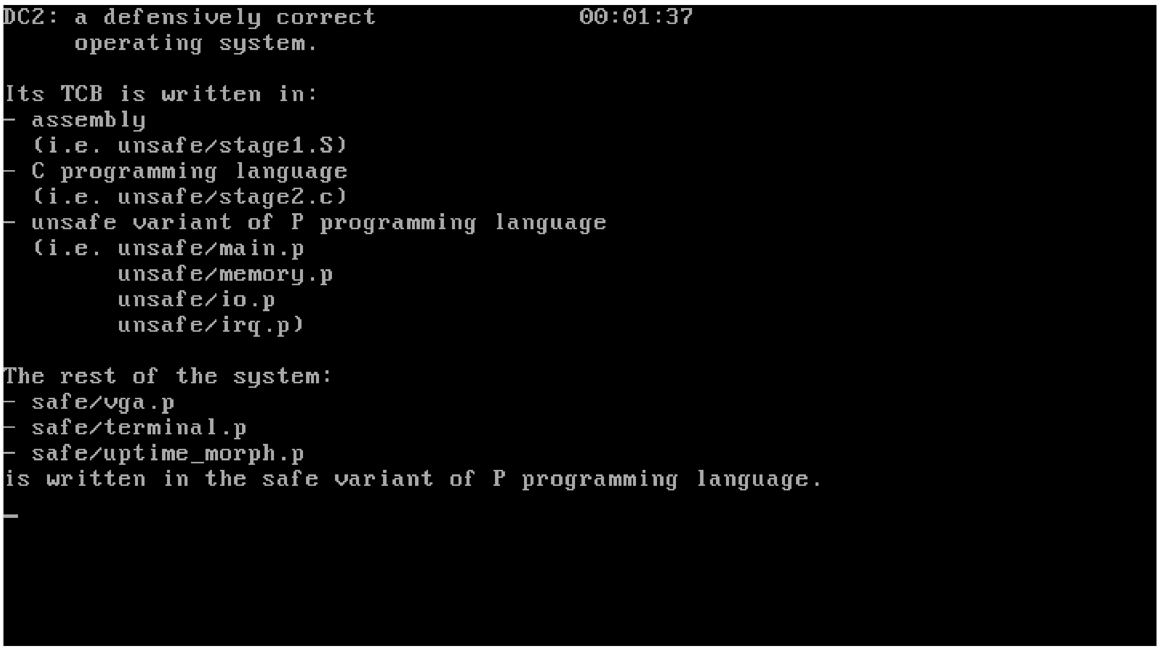
6.4 POLA

Even though none of the untrusted subsystems needs maximal authority[12], each of them nevertheless needs non-minimal authority. E.g., `vga` subsystem needs the permission to read from I/O port `0x3D5`. We do not send it a capability to function that could be used to read all existing I/O ports. Instead, we give it a capability to a proxy which allows `vga` to read that particular I/O port. Similarly, `vga` subsystem needs the permission to write to I/O port `0x3D5`. We therefore send it a capability to a proxy which allows `vga` to write to that one specific I/O port. The definition of P programming language contains closures so these proxies can be define with a single line of code.

With proxies, we can give individual untrusted subsystems access to specific I/O ports, specific regions of memory, register observers for specific IRQs or whatever else is appropriate in a given case. After the system boots, the trusted main program, at our discretion, creates appropriate proxies and redistributes capabilities around in concord with POLA.

The main program also, at our discretion, defines memory and CPU quotas for individual untrusted subsystems. Figure 6 illustrates where in the system we use proxies to provide indirect and attenuated access to underlying services.

[11]I.e., `vga.p`, `timer.p`, `uptime_morph.p`, `life_morph`
[12]Maximal authority is wielded only by TCB.



```
DC2: a defensively correct              00:01:37
     operating system.

Its TCB is written in:
- assembly
  (i.e. unsafe/stage1.S)
- C programming language
  (i.e. unsafe/stage2.c)
- unsafe variant of P programming language
  (i.e. unsafe/main.p
        unsafe/memory.p
        unsafe/io.p
        unsafe/irq.p)

The rest of the system:
- safe/vga.p
- safe/terminal.p
- safe/uptime_morph.p
is written in the safe variant of P programming language.
_
```

Figure 4: A screenshot the OS.

The whole system is defensively correct because all its subsystems
(TCB, `vga`, `terminal`, `uptime_morph`) are defensively correct:

- TCB is defensively correct because none of its clients (there is only one: `vga`) can force it to stop providing the correct service for all its clients:
  - `vga` cannot force TCB to stop providing the correct service for all TCB's clients because `vga` does not have a sufficient authority to do that.
- `vga` is defensively correct because none of its clients (there are two of them: `terminal` and `uptime_morph`) can force it to stop providing the correct service for all its clients:
  - `terminal` cannot force `vga` to stop providing the correct service for all `vga`'s clients because `terminal` does not have a sufficient authority to do that.
  - `uptime_morph` cannot force `vga` to stop providing the correct service for all `vga`'s clients because `uptime_morph` does not have a sufficient authority to do that.
- `terminal` is defensively correct because none of its clients (there are none) can force it to stop providing the correct service for all its clients.
- `uptime_morph` is defensively correct because none of its clients (there are none) can force it to stop providing the correct service for all its clients.

Figure 7: A proof of OS's defensive correctness.

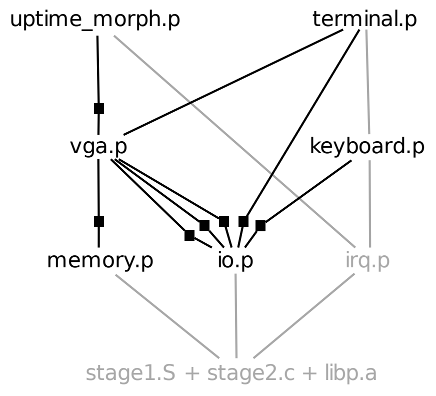


Figure 6: Custom proxies (■) mediate appropriately attenuated services of servers below them to their clients above them. For example, the proxy between `vga` client and `memory` server enables `vga` client to manipulate specific portion of the physical memory. Proxies between `vga` client and `io` server enable `vga` client to read or write specific I/O ports. The proxy between `uptime_morph` client and `vga` server enables `uptime_morph` to draw itself to specific rectangular region of the screen. Etc.

6.5 Defensive correctness

With respect to reliance relationship shown in Figure 5 we can prove that operating system is defensively correct, see Figure 7.

The authority of individual untrusted subsystems is declared in the technical documentation [25]. We establish it informally. Formal treatment of authority is beyond the scope of our thesis. Alfred Spiessens's and Toby Murray's PhD theses, for example, deal with this problem [41, 56] rigorously.

7. Conclusions and future work

We created a defensively consistent operating system using Tamed Pict programming language. TCB of such system is as small as 2000 lines of code. New untrusted subsystems can be added at the top of the TCB. The general Powerbox-sandbox security pattern is used to enforce describable security policies over these untrusted subsystems. The proof of defensive consistency of the whole system is given. These results were published in [24]. Additionally, a small set of students helped with evaluation of comprehensibility and flexibility of the employed techniques [13, 18, 19, 62].

In order to achieve defensive correctness we aimed at the solution with the smallest inflationary impact on the TCB size. Therefore, we proposed P language, as a new version of the Pict programming language. Subsequently, small defensively correct operating system was implemented. TCB of such system is still as small as 2000 lines of code. Again, new untrusted subsystems can be added on top of this TCB and general Powerbox-sandbox security pattern can be used to enforce describable security policies over untrusted subsystems. The proof of defensive correctness of the whole system is given. Description of these results will appear at ECBS EERC 2011 conference [27].

Untrusted subsystems are started in a sandbox with minimal authority with memory and CPU quotas. The ultimate authority is determined by the set of capabilities their receive at run-time. So, individual untrusted subsystems follow principle of least authority.

We consider the goal of the thesis to build an operating system that:

- has a minimal *trusted computing base*,
- its components follow *principle of least authority*,
- and the whole system is *defensively correct*,

achieved.

In the future, with respect to *reliance relationship*, the system will grow from bottom-up. We will continue to follow the client-server architecture [59] although device drivers in our operating system will not necessarily form a flat layer, as in Minix, but a partial order. We will try to consider various Linux's `ioctl` extensions that represent exceptions from the crude character/block/network classes. We will consider a finer distinction expressed by subtyping. The proper operating system interface is also an open (research) question.

Acknowledgments

This work was partially supported by the Scientific Grant Agency of Slovak Republic, grant No. VG1/0508/09. This contribution/publication is also a partial result of the Research&Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] Google Caja.
`http://code.google.com/p/google-caja`.

[2] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1998.

[3] D. J. Bernstein. Some thoughts on security after ten years of `qmail` 1.0. In *CSAW '07: Proceedings of the 2007 ACM workshop on computer security architecture*, pages 1–10, New York, NY, USA, 2007. ACM.

[4] Y. Bertot and P. Casteran. *Interactive theorem proving and program development*. Texts in theoretical computer science. An EATCS series. Springer-Verlag, 2004.

[5] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the workshop on micro-kernels and other kernel architectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association.

[6] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming Under Mach*. Addison-Wesley Professional, 1993.

[7] C. Chu. Introduction to Microsoft .NET security. *IEEE Security & Privacy*, pages 73–78, 11–12 2008.

[8] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[9] H. Geuvers. Proof assistants: History, ideas and future. *SÄĄdhanÄĄ*, 34(1):3–26, 2009.

[10] L. Gong. Java security: Present and near future. *IEEE Micro*, 17(3):14–19, 1997.

[11] N. Hardy. KeyKOS architecture. *SIGOPS operating systems review*, 19(4):8–25, 1985.

[12] G. Heiser. Your system is secure? Prove it! *USENIX login*, 32(6):35–38, 2007.

[13] M. Hečko. Ovládače jednoduchých zariadení v jazyku Pict. Bachelor's thesis, Slovak University of Technology in Bratislava, 2008.

[14] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, SACMAT '07, pages 91–100, New York, NY, USA, 2007. ACM.
`http://doi.acm.org/10.1145/1266840.1266854`.

[15] M. Howard and D. LeBlanc. *Writing secure code*. Microsoft Press, 2 edition, Jan. 2003.

[16] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on history of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[17] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: Challenges and opportunities. In *HOTOS '05: Proceedings of the 10th conference on hot topics in operating systems*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[18] M. Kallo. Powerboxed video card driver. Master's thesis, Slovak University of Technology in Bratislava, 2008.

[19] O. Kallo. Ovládače jednoduchých zariadení v jazyku Pict. Bachelor's thesis, Slovak University of Technology in Bratislava, 2009.

[20] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp Symbolic Computation*, 7(4):315–335, 1994.
`http://bit.ly/gcNxLV`.

[21] G. Klein. Correct OS kernel? Proof? Done! *USENIX login*, 34(6):28–34, 2009.

[22] G. Klein. Operating system verification—an overview. *SÄĄdhanÄĄ*, 34(1):27–69, 2009.

[23] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM symposium on operating systems principles*, Big Sky, MT, USA, Oct 2009. ACM.

[24] M. Košík. Taming of Pict. In V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 2008.

[25] M. Košík. DC2: A defensively correct operating system, 2011.
`http://www2.fiit.stuba.sk/~kosik/dc2.html`.

[26] M. Košík. Implementation of the P compiler and its runtime, 2011.
`http://www2.fiit.stuba.sk/~kosik/p.html`.

[27] M. Košík and J. Šafařík. A contribution to techniques for building dependable software systems. In *Proceedings of 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2011*, Bratislava, Slovakia, Sept. 2011. IEEE Computer Society. To appear.

[28] C. Landau. CapROS: The capability-based reliable operating system.
`http://www.capros.org`.

[29] C. R. Landau. Security in a secure capability-based system. *SIGOPS Oper. Syst. Rev.*, 23(4):2–4, 1989.

[30] X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[31] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[32] J. Liedtke. Improving IPC by kernel design. In *SOSP '93: Proceedings of the fourteenth ACM symposium on operating systems principles*, pages 175–188, New York, NY, USA, 1993. ACM.

[33] B. J. MacLennan. *Functional programming: Practice and theory*. Addison-Wesley, 1990.

[34] A. Mettler and D. Wagner. Class properties for security review in an object-capability subset of Java: (short paper). In *PLAS '10: Proceedings of the 5th ACM SIGPLAN workshop on programming languages and analysis for security*, pages 1–7, New York, NY, USA, 2010. ACM.

[35] A. M. Mettler and D. Wagner. The Joe-E language specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, 3 2006.

[36] M. Miller. E: Open source distributed capabilities.
`http://www.erights.org`.

[37] M. S. Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[38] M. S. Miller, B. Tulloh, and J. S. Shapiro. The structure of authority: Why security is not a separable concern. In *Multiparadigm programming in Mozart/Oz: Extended proceedings second international conference MOZ 2004*, pages 2–20, 2004.

[39] R. Milner. *Communicating and mobile systems: The π-calculus*. Cambridge University Press, 1999.

[40] A. P. Murray and D. A. Grove. PULSE: A pluggable user-space Linux security environment. In *AISC '08: Proceedings of the sixth Australasian Conference on Information Security*, pages 19–25, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[41] T. Murray. *Analysing the security properties of object-capability patterns*. PhD thesis, Oxford University Computing Laboratory, 2010.

[42] S. Owens, G. Peskine, and P. Sewell. A formal specification for OCaml: The core language, 2 2008.
`http://bit.ly/dMiBOL`.

[43] B. C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict.
`http://bit.ly/fD8Ft5`, 1997.

[44] B. C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[45] B. C. Pierce and D. N. Turner. Pict language definition.
`http://bit.ly/fHoRDJ`, 1997.

[46] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, language and interaction: Essays in honour of Robin Milner*. MIT Press, 2000.
`http://bit.ly/lfq70F`.

[47] R. Rashid, R. Baron, R. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 1989 IEEE international conference, COMPCON*, pages 176–178. Press, 1989.

[48] J. A. Rees. A security kernel based on the lambda-calculus. *A. I. Memo 1564, MIT*, 1564, 1996.

[49] J. Rutkowska and R. Wojtczuk. Qubes OS Architecture, 2010.
`http://qubes-os.org/files/doc/arch-spec-0.3.pdf`.

[50] M. Seaborn. Plash.
`http://plash.beasts.org/wiki/`.

[51] J. Shapiro. Programming language challenges in systems codes: Why systems programmers still use C, and what to do about it. In *PLOS '06: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems*, page 9, New York, NY, USA, 2006. ACM.

[52] J. S. Shapiro and N. Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Software*, 19(1):26–33, 2002.

[53] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, 1999.

[54] G. Smolka. The definition of kernel Oz. In *Selected papers from constraint programming*, pages 251–292, London, UK, 1995. Springer-Verlag.

[55] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts, 2006.

[56] A. Spiessens. *Patterns of safe collaboration*. PhD thesis, UniversitÃľ catholique de Louvain, Belgium, Feb. 2007.

[57] M. Stiegler. Emily: A high performance language for enabling secure cooperation. *International conference on creating, connecting and collaborating through computing*, 0:163–169, 2007.

[58] A. S. Tanenbaum. Grant proposal: *Research on really reliable and secure system software (R3S3)*, 2008. `http://bit.ly/cfN9dL`.

[59] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: Design and implementation*. Pearson Prentice Hall, 2006.

[60] L. Torvalds. On Specifications. `http://kerneltrap.org/node/5725`.

[61] D. N. Turner. *The polymorphic pi-calculus: Theory and implementation*. PhD thesis, University of Edinburgh, 1995.

[62] J. Valo. Ovládače jednoduchých zariadení v jazyku Pict. Bachelor's thesis, Slovak University of Technology in Bratislava, 2008.

[63] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. `http://portal.acm.org/citation.cfm?id=1929820.1929824`.

[64] J. Whittaker. Why secure applications are difficult to write. *IEEE Security and Privacy*, 1(2):81–83, 2003.

[65] F. Wiedijk. The seventeen provers of the world.

[66] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, Aug. 2002.

Selected Papers by the Author

M. Košík and J. Šafařík. A contribution to techniques for building dependable software systems. In *Proceedings of 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2011*, Bratislava, Slovakia, Sept. 2011. IEEE Computer Society. To appear.

M. Košík. Prirodzená dedukcia. In *Umelá inteligencia a kognitívna veda II*, pages 125–146. Vydavateľstvo STU v Bratislave, 2010.

M. Košík. Taming of Pict. In V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 610–621. Springer, 2008.

M. Košík. Strategies for memory accounting of cooperating pi-calculus processes. In M. Češka, Z. Kotásek, M. Křetínský, L. Matyska, T. Vojnar, and D. Antoš, editors, *MEMICS*, pages 115–122, 2008.

M. Košík. Concerning untrusted code. In M. Bieliková, editor, *IIT.SRC: Student research conference*, pages 356–361. Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Apr. 2007.

M. Košík. On the composability of concurrent systems. In M. Bieliková, editor, *IIT.SRC 2005: Student research conference*, pages 159–166. Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava, Apr. 2005.