

A Contribution to Techniques for Building Dependable Software Systems

Matej Košík

Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
kosik@fiit.stuba.sk

Jiří Šafařík

Faculty of Applied Sciences
University of West Bohemia
Plzeň, Czech Republic
safarikj@kiv.zcu.cz

Abstract—Dependability is an essential property of critical systems and it also contributes to the quality of non-critical systems. There already exists an agreement *what* this term means, but we are still in the process of searching for an answer *how* to create dependable software systems in a cost-effective way. For different contexts, the *how* question may have different correct answers. In this paper, we present how an object-capability programming language P, which we have developed, can positively influence the following attributes of dependability: *safety, confidentiality, correctness, and robustness*. This is demonstrated using P language to build a simple operating system.

Keywords—software engineering; dependability; programming languages, operating systems

I. 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.

In this paper we define tools that enable us to build dependable software systems with a minimal trusted computing base (TCB). That means that:

- we describe major properties of a new domain-specific programming language designed precisely for the purpose of TCB reduction;
- we describe major properties of a small dependable operating system that was implemented to prove our point concerning TCB minimization;

- we describe how the proposed programming language enables us to achieve *safety, confidentiality, correctness* and a high *robustness* level.
- we recount how the choice of this programming language affected TCB size.

The syntax, semantics, type system and implementation of the proposed programming language (P) is described in the technical documentation [1]. The complete information concerning the described dependable operating system (DC2) can be found in its technical documentation [2].

We achieve these goals by extending the work of Pierce and Turner [3]. The proposed extensions of the programming language's semantics simplify proofs of so called *defensive correctness*, i.e. of the highest defined level of robustness. This is our original contribution.

II. TERMINOLOGY

Conventional

We use the terms *subject, object, operation, permission*, and *trusted computing base* 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 dependability of the entire system.

Unconventional

A *capability* identifies some specific object and provides its holder with the permission to operate on the object it identifies. Capabilities must either be totally unforgeable or infeasible to forge. For example, references to objects in Caja, which is a retrofitted version of JavaScript, are capabilities that cannot be forged. 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. 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. 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. 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 of O and S also tries to invoke those operations. In that case, S assumes that O implements them correctly. *Reliance set* of some subsystem is a set of all its successors in a given *reliance relationship*, see Figure 1.

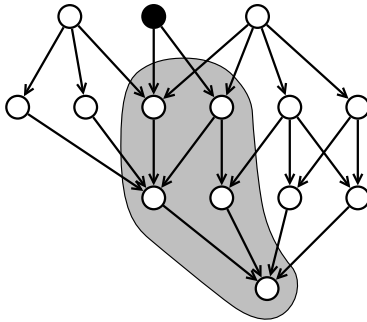


Figure 1. A *reliance set* (denoted with gray color) of the chosen subsystem (denoted with black color) in a given *reliance relationship*.

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 an 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 the correct service for all its well-behaving clients. *Defensive correctness* implies *defensive consistency*.

III. THE CHOICE OF A BATTLEGROUND

We believe that the problem of building a dependable operating system represents a very reasonable common ground, on which different approaches for building dependable software systems can be compared. By the term—dependable operating system—we mean that:

- it must have as small *trusted computing base* as possible;

- its subsystems must unconditionally follow the *principle of least authority*;
- and the whole system must be *defensively correct*.

We expect that the above set of goals will be extended over time. At the moment we focus on those properties that cannot be added later as an afterthought. The current version of our kernel, which we use to demonstrate proposed techniques, provides at the moment only two services:

- There is a terminal driver that interprets sequences of SCAN-codes: it puts new characters on the screen, it deletes characters from the screen if you press Backspace key, it reboots the computer if you press Ctrl+Alt+Del.
- There is a small subsystem which displays uptime in HH:MM:SS format.

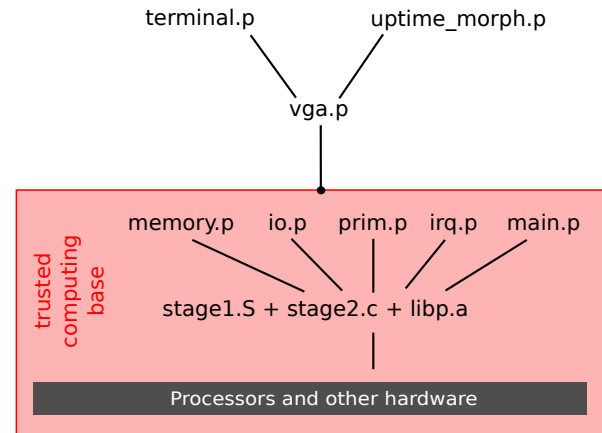


Figure 2. *Reliance relationship* of the DC2 operating system [2]. Individual subsystems are shown as nodes. Edges indicate which subsystem uses services of which other subsystem. Clients are above; servers are below.

Reliance relationship of individual subsystems is shown in Figure 2. Part of the kernel is written in assembly (`stage1.S`). Part of the kernel is written in the C programming language (`stage2.c`, `libp.a`). The remaining parts of the kernel (`memory.p`, `prim.p`, `io.p`, `irq.p`, `main.p`, `vga.p`, `uptime_morph.p`, `terminal.p`) are written in a domain specific language P designed to enable us to achieve the declared goals. Due to limited space, we resigned to introduce P programming language’s formal syntax, typing rules and semantics. We describe informally what makes this language different from other programming languages. The complete language definition can be found in the technical documentation [1].

IV. SAFETY

We want to limit a possible damage that can be caused by faulty or even malicious pieces of software we decide to include in the operating system. Subsystem’s safety is in an inverse proportion to its authority. Every untrusted

subsystem is therefore started in a sandbox. At this moment, subsystem is absolutely safe. Subsequently we trade, at our discretion, a small amount of the safety for total usability of a given untrusted subsystem. In concord with *principle of least authority* (POLA), the authority of the untrusted subsystem is raised to the point, in which it already can provide all the expected services. If they are incorrect, faulty, or malicious, the possible damage is very limited and in advance predictable.

Untrusted subsystems informally, but explicitly, declare required capabilities. In this respect we consciously deviate from the mainstream approach where untrusted operating system components are installed with *excess authority* that allows them not only to perform intended actions, if they are correct and flawless, but it allows them also to cause a lot of damage if they are incorrect or flawed.

With lexical scoping we are able to hide definitions of powerful functions from the sight of untrusted subsystems. TCB reflects our goals concerning safety. TCB therefore passes capabilities to properly attenuated proxies to constructors of untrusted subsystems. For example, TCB does not pass to the constructor of the `vga` subsystem a capability that would allow `vga` subsystem to write any byte anywhere in the physical memory. Instead, TCB passes it a capability to a trivial proxy that `vga` subsystem can use to manipulate at will only a small region of the physical memory where frame-buffer is mapped.

Creation of custom proxies to powerful services represents a general-purpose attenuation mechanism. All untrusted subsystems get exactly as much authority as they objectively need to provide expected services. Proxies are typically implemented trivially by currying or one-line anonymous functions. Figure 3 shows usage of custom proxies in DC2 operating system. The triviality with which they can be created encourages us to use them wherever we need them. For example, the proxy between `vga` client and `memory` server enables `vga` client to manipulate a 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 a specific rectangular region of the screen. Etc.

V. CONFIDENTIALITY

The safety of the programming language and the properties of the type system enable us to prove that untrusted subsystems cannot leak secrets of certain types we pass to them, i.e., to solve the so called *confinement problem*. These techniques are described e.g. in [4] and elsewhere. This is not our original contribution. The static type system of our safe programming language enables us to easily reason about this problem. Known capabilities can be leaked from the examined subsystem to its context only through free variables of that subsystem. If we consider the context, in

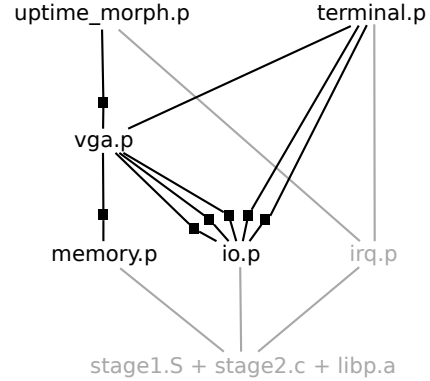


Figure 3. Custom proxies (■) mediate appropriately attenuated services of servers below them to their clients above them.

which given subsystem appears, we can determine types of its free variables. By examining these type terms we can determine values of which types can flow out of the subsystem and which cannot. To some degree, this technique is useful. It enables us to prove useful statements about the possible flow of capabilities in the system. This technique may be useless for proving interesting statements concerning the flow of raw data in the system. In case of programming languages with dynamically checked types, a similar goal can be achieved by run-time filtering of information flow.

The same procedure can also be used for solving the complementary problem. We can prove that the analyzed untrusted subsystem cannot learn secrets of certain type, i.e., it cannot learn new capabilities to powerful functions that would escalate its authority.

If, for example, we want to prove that the authority of `uptime_morph` in Figure 3 cannot suddenly escalate, we identify free variables of a given subsystem and types of these variables:

```
put_char : i32 i32 i32 -> []
tick     : ?.[]
```

The first free variable (`put_char`) is a function-capability. The designated function takes three 32-bit integer parameters and it returns the unit value. As long as the whole system is well-typed, there is no way how `uptime_morph` could use this capability to learn new capabilities. The second free variable (`tick`) is a channel-capability. It can be used for receiving the unit value from the designated channel. As long as the whole system is well-typed, there is no way how `uptime_morph` could use this capability to learn new capabilities. QED.

Soundness of this kind of proofs relies on the semantics of the safe subset of P programming language in which all untrusted subsystems are described. That means that:

- Untrusted subsystems cannot forge new capabilities.
- Untrusted subsystems can learn new capabilities only

by parenthood, introduction, endowment or by initial conditions [5, §9.2].

- Untrusted subsystems cannot violate type constrains.

The technical documentation [2] of DC2 operating system omits these proofs because they are trivial. In this section we formulate one such proof. Other proofs can be formulated analogously.

VI. ROBUSTNESS

“Our computers are fragile—that is essentially the problem. Attacking is simple because so much can fail. Defending fragile system is hard because you have to essentially find every bug in the system to prevent it from being exploitable [8].” These fragile software systems are as reliable as their least reliable component; as secure as their most severe security flaw and as safe as their most malicious component. This is annoying in case of non-critical systems and it is unacceptable property of critical systems. E.g. a short circuit in a single household cannot cause blackout in all households, offices, factories, hospitals etc. in the city. Critical software systems must have a similar property—robustness. The effects of a failure of a single domain of trust must be compartmentalized.

The previous sections sketched some useful properties of the P programming language:

- Untrusted subsystems can be put in a sandbox.
- Untrusted subsystems can be installed consistently with POLA (principle of least authority).
- We can easily prove that untrusted subsystems cannot learn new capabilities over time. Section V contains one such proof. Other proofs can be formulated analogously.
- We can easily prove that untrusted subsystems cannot leak capabilities. These proofs can be formulated analogously to the proof given in Section V.
- Tiny TCB (trusted computing base) consisting of 1988 lines of code minimizes the cost of formal verification.
- Sandboxing of individual untrusted subsystems and defensive correctness of the entire system enable us to verify individual untrusted subsystems separately.

A. Defensive consistency

Defensively consistent software systems differ from *fragile* software systems in respect that here clients cannot, neither by unintentional errors nor by intentional malice, force servers to provide an incorrect service for any of their clients. Without disrupting other relevant attributes of dependability described in the prior text, we have achieved [9] this level of robustness by:

- retrofitting the standard library of the chosen programming language
- and forbidding dangerous language constructs within untrusted subsystems.

Figure 4 shows how easily can we prove defensive consistency of DC2 operating system. The proof itself relies on the semantics of the adapted programming language.

B. Defensive correctness

If we are concerned with *critical systems* [10] then *defensive consistency* is still too weak achievement. It is not satisfactory merely to ensure that clients cannot force servers to provide an incorrect service for any of their clients. It is equally essential to ensure that clients will not be able to force servers to stop providing correct services for all their well behaving clients. Platforms, where we can assure that, can then be used for building *defensively correct* software systems. To the best of our knowledge, no programming languages were designed to support defensive correctness. In the next sections we give some more details about the Pict programming language, we analyze its weaknesses and describe possible ways how it can be retrofitted to enable us to build defensively correct software systems.

The original Pict programming language: It was created with the goal to explore usability of a programming language whose semantics is mapped to pi-calculus. Figure 5 shows

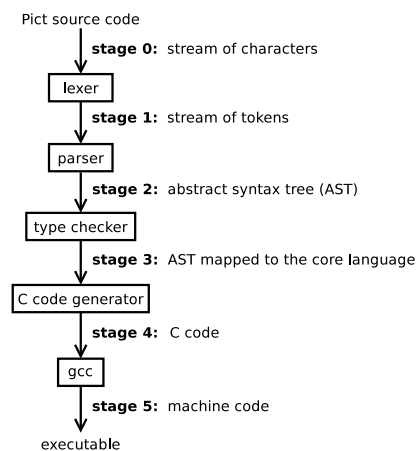


Figure 5. Simplified diagram of individual compilation phases of the original Pict compiler.

individual compiler phases. The *C code generator* breaks the whole Pict program to a set of *threads*. Each thread is a short finite sequence of actions. There is no interleaving among individual threads. Each activated thread is executed from the beginning until it terminates or blocks. The scheduler then picks the next activated thread and so on. Since we know everything about the nature of individual actions, and at compile time we know precisely from which primitive actions are individual threads composed, the compiler is therefore able to compute how much memory will at most individual threads try to allocate from the common pool of free memory. It is therefore possible to compute how much memory could at most individual threads try to allocate from

DC2 is defensively consistent because all its subsystems (TCB, vga, terminal, uptime_morph) are defensively consistent:

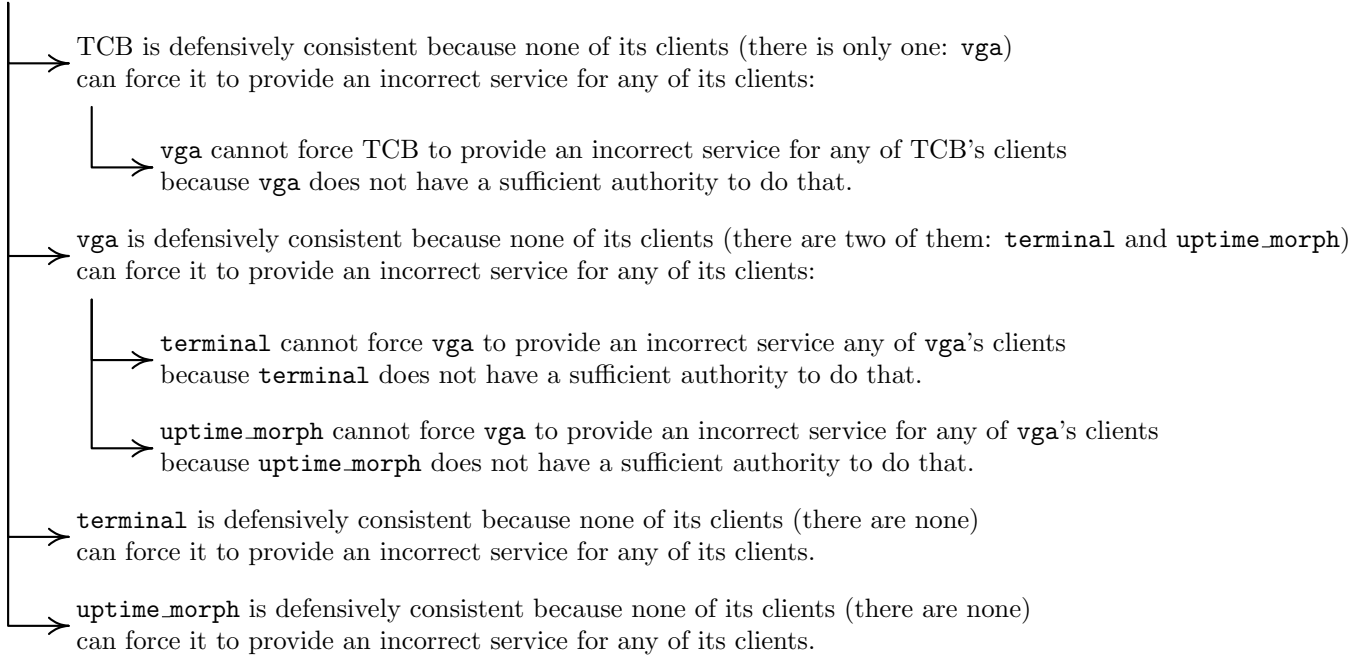


Figure 4. A proof of DC2's defensive consistency.

the common pool of free memory shared by all threads. The scheduler then checks if it is safe to execute a chosen thread and if not, it invokes the garbage collector. If neither garbage-collector is able to reclaim sufficient amount of free memory to run the activated thread, the whole system stops.

The proposed P programming language: The P programming language was defined with respect to all possible different scenarios of denial of service (DoS) attacks, that we could to imagine. This new language version not only enables potentially fruitful cooperation among mutually suspicious subsystems, but it also enables us to defend servers from their clients that could try to perform DoS attacks. In the following paragraphs we describe the influence of three different DoS scenarios on the language definition. Below, we describe only an obvious subset of all problematic situations we have considered. These and other less obvious problems are described in compiler's technical documentation [1, §11.4.8].

A defense against "cancer" processes: By "cancer" we mean a situation when some of the subsystems allocates and retains all the available free memory and consequently prevents progress of other subsystems. To be able to efficiently defend the rest of the system from the cancer-like scenarios, we should enable the programmer to put every untrusted subsystem to a separate *memory domain* with its own *memory quota*. It is up to the programmer to judiciously create one domain for each untrusted subsystem.

In the P programming language, the desired construct has the following form: $(\text{lim } i \text{ } p)$ where *lim* is simply a keyword, *i* is an integer that defines memory quota of the new domain and *p* is a process term defining the behavior of the new memory domain. Our reduction rules [1, §11.4] define for every situation to which domain will belong newly created objects. If the pool of the free memory is depleted, it is now trivial to determine the domain that is responsible for this situation.

Obviously, the set of services we lose, when some of the subsystems must be killed, depends on its position in the overall reliance relationship of a given system. Let us illustrate this on Figure 2. If *uptime_morph* is killed, all the other subsystems can continue to provide expected services because *uptime_morph* has no predecessors. Similarly, if *terminal* is killed, all the other subsystems can continue to provide expected services because *terminal* has no predecessors. On the other hand, if we kill *vga*, then we lose services of *vga*, *uptime_morph* and *terminal* because both, *uptime_morph* as well as *terminal* rely on *vga*. The larger the system and the sparser the reliance relationship, the more advantages the defensive correctness provides.

A defense against "spammer" processes: Reliance relationship (Figure 2) enables us to recognize two directions for message passing:

- Messages that flow in the top-down direction represent

requests for services sent by clients (above) to servers (below).

- Messages that flow in the bottom-up direction either represent responses of servers to prior clients' requests, or they indicate asynchronous occurrence of some event to which those clients subscribed.

Let us first focus on messages sent in the top-down direction. By "spammer" scenario we mean a situation when some client floods some server by excess number of messages. Such a behavior could also lead to depletion of the common pool of available free memory as in the previous case. We want to defend the rest of the system from spammers. We achieve this by forcing spammers "to be responsible" for all the messages they generate until they are consumed. Our reduction rules put each sent message to the same memory domain to which belongs the process that has sent them. Spammers will be killed because their memory consumption exceeds their memory quota.

If, for example, `uptime_morph` in Figure 2 begins to act as a spammer, i.e., if it sends some extreme number of messages to some of the servers to which it has access, then these messages will be accounted to the same memory domain where `uptime_morph` belongs. Sooner or later that domain exceeds its memory quota and, while we will be sorry to lose it, we will have to kill it. The rest of the system, that is independent from `uptime_morph`'s services, will continue to operate.

A defense against "disregardful" processes: Let us now focus on messages sent in the bottom-up direction. For example, the `irq` subsystem (see Figure 2) sends a unit value up to the `uptime_morph` every time when IRQ 0 occurs. IRQ 0 is generated periodically by the programmable interval timer (PIT) chip with some fixed frequency (20 Hz in our case). These notifications enable `uptime_morph` to keep track of time to be able to identify moments when the contents of the screen must be updated. Let us suppose that `uptime_morph` contains an error or intentionally disregards these notifications. Sooner or later the whole pool of free memory would be depleted. If that happens, it is fair to blame (and kill) `uptime_morph` for this situation. After we kill `uptime_morph`, what would remain from the original system would provide fewer services than the original but it would continue to operate and the level of services might be sufficient for achieving the intended goal. Failures or malice of non-crucial subsystems cannot undermine operation of crucial subsystems. It would not be fair to blame `irq` subsystem. To motivate `uptime_morph` to consume these notifications, we would like them to be owned by `uptime_morph`.

In different situations we therefore need two different policies concerning message ownership. We can enforce them by supporting two different variants of the *send* operation:

- *regular send* operation where the sent message is

accounted to the sender,

- and so called *donating send* operation where the sent message is accounted to the receiver.

Clients will be given the permission to invoke services of servers via *regular sends*. If servers are expected to autonomously contact clients, then clients will be obliged to grant servers the permission to perform *donating send* to the some designated channel. In the context of an N-tire system with a chain of servers, one and the same subsystem can act both as a client as well as a server. When it acts as a client of some server below, it should be obliged to use *regular send*. When it acts as a server responding back to clients above, it should be permitted to use *donating send*. The rule is applicable and plausible also in situations when the client-server relationship forms a partial order. Attempts to violate granted permissions:

- to perform an input operation on a channel,
- to perform a regular output operation a channel,
- or to perform a donating output operation on a channel

are detected at compile time by the type checker. We achieve this by building upon the idea of I/O-types [4].

There seems to be a flaw in this scheme. Servers, as soon as they get permission to perform the *donating send* operation, can flood their clients and sink them. However, clients in any case already rely on servers. Clients do not have any choice but to trust servers that they are delivering the promised services. Therefore coming up with mechanisms that could protect clients from misbehaving or malicious servers makes no sense. At least, not in the client-server system. Peer-to-peer systems need to address this problem though.

With respect to reliance relationship shown in Figure 2 and semantics of the P programming language we can prove that DC2 operating system is defensively correct, see Figure 6.

VII. CORRECTNESS

Even though there is an evidence that formal verification can be applied to complicated systems, see e.g., formal verification of a C compiler [6] done in Coq [7], these methods are still relatively expensive. In general, overall cost of formal verification directly depends on the total size of the software system we are trying to verify. By ensuring defensive correctness, we disrupt this rule. The proof of correctness of subsystem *S* remains valid regardless of any modifications of any subsystem which does not belong to its *S*'s *reliance set*. For example, the proof of correctness of the *terminal* (shown in Figure 3) remains valid regardless of the actual behavior of `uptime_morph`

The size of the trusted computing base (TCB) also contributes to the costs of formal verification. Current size of TCB of DC2 operating system is 1988 lines of code, see Figure 7. We do not count the size of the compilers,

DC2 is defensively correct because all its subsystems (TCB, `vga`, `terminal`, `uptime_morph`) are defensively correct:

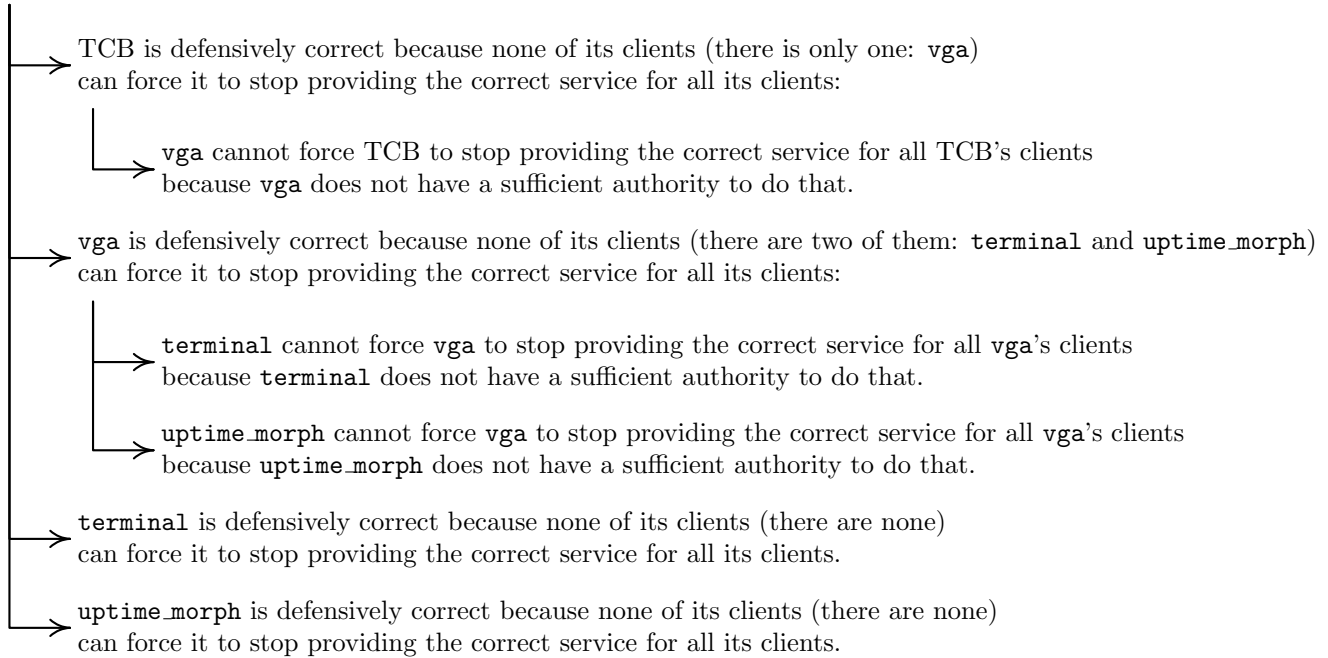


Figure 6. A proof of DC2's defensive correctness.

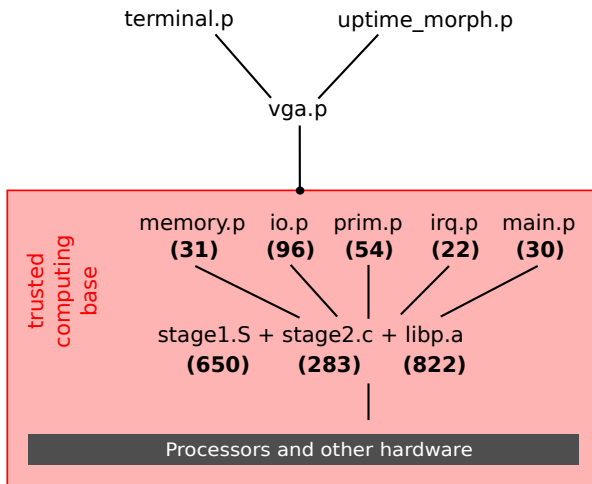


Figure 7. The size, in lines of code, of individual parts of the kernel that belong to the trusted computing base (TCB).

the linker, and the underlying hardware. The P compiler is written in 6585 lines of Ocaml code. It is based on the work of Pierce and Turner [3]. It generates C code that can be further processed by an ordinary C compiler or the certified C compiler [6]. The TCB takes control from the boot loader and provides hosting for sandboxed subsystems written in the safe subset of the P programming language. There is

a simple garbage-collector that deletes unreachable objects and a simple scheduler for processes written in the safe programming language P. They define behavior of individual sandboxed subsystems.

As we will add new untrusted subsystems to the operating system, the growth of TCB will be unavoidable. However, TCB growth will not be proportional to the size of the added untrusted subsystem. TCB growth will be proportional to the size of the security policy that defines authority of a given untrusted subsystem. Security policies are expressed simply by passing properly chosen capabilities to constructors of untrusted subsystems.

VIII. FUTURE WORK

We would like to make the audit of the proposed scheme more cost effective. In other words, if there is a flaw in the scheme, we would like to reduce the cost required for finding it. The current version of the compiler generates frustratingly bloated C code. This lengthens the time required for generating executable code to a point where we risk that auditors will lose patience and that in turn makes the auditing more expensive. Our plan therefore is:

- 1) to add a proper support for separate compilation;
- 2) to reduce the amount of the generated C code;
- 3) to write a document with the definition and the tutorial of the programming language by updating existing documents [11, 12].

The first two subgoals will simplify practical testing and penetration. The last subgoal will simplify theoretical evaluation and examination of the (relevant) source code.

The next important step is a validation of the proposed TCB—whether it is sufficiently complete. This can be done by implementing a reasonably complete dependable operating system around it. This can be achieved by a small but dedicated team.

When we integrate optimizations that are performed by Pict compiler to P compiler, we can run common benchmarks. Pierce and Turner constructively proved that pi-calculus can be compiled effectively [3].

Our hope is that we will come up with a trusted computing base that will be sufficiently attractive for experts in formal verification. P programming language’s semantics is formally defined in the technical documentation [1]. This opens the door for creation of a certified compiler.

In its current form, the prototype of the P compiler and the P runtime cannot be used for building real-time systems. If we aimed at that, we would have to reconsider the scheduling mechanisms, the scheduling policy, and the garbage-collection scheme as well as the related `kill` procedure.

IX. RELATED WORK

Dependable operating systems are composed from:

- a small trusted computing base (TCB)
- and a large set of useful untrusted extensions (e.g., device drivers) that follow principle of least authority (POLA)

How big is the set of untrusted device drivers, that typically depends on the quantity of the time devoted to creation of new device drivers. How small is the TCB depends on the quality of basic ideas that are used to build the TCB.

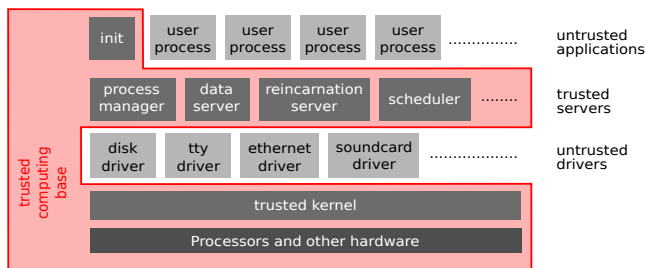


Figure 8. Internal structure of the Minix3 operating system [13].

The traditional approach for building dependable operating systems is by means of microkernel architecture. A mature example of this approach is Minix3 [14]. The whole operating system is implemented as a set of processes with separate address spaces, see Figure 8. The advertised [15] size of the kernel is 6000 lines of code (LoC). The reality is more complicated. First, the authors do not count all numerous

header files that this kernel includes. These are part of the TCB too because they define crucial macros and constants that are used to implement the kernel. Second, what matters is the size of the whole TCB and that, in addition to the actual kernel, contains also:

- the process manager,
- the data server,
- the reincarnation server, and
- the scheduler.

Flaws in these subsystems may invalidate assumptions concerning the authority of untrusted OS extensions. After we have laboriously collected the names of the files that define TCB behavior and dropped comments embedded in those files, we have got our conservative estimate of the Minix’s TCB size which is 30 404 lines of C and assembly code. There are multiple disjoint communities, e.g., Genode [16], working on projects that have comparable properties.

Microkernel architecture is not the only way how to achieve *compartmentalization* of untrusted OS extensions. Some safe programming languages have similar capabilities. Some research groups noticed this and tried to explore this new possibility. One of the latest and sufficiently mature project is Singularity operating system [17].

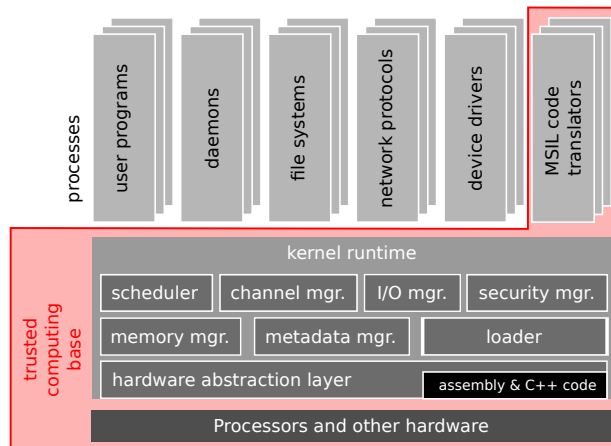


Figure 9. Internal structure of the Singularity operating system [18].

According to Figure 9 and the available source code, we were able to estimate the size of Singularity’s TCB at 150 000 LoC. We expect that others will be able to provide better results—a dependable operating system with a smaller and simpler TCB. This will be an important step, otherwise Singularity could continue to be used as a straw man argument against usage of safe languages as a base for dependable OS, see e.g., [19].

Minix3 and Singularity share the viewpoint on how POLA should be enforced. In both cases there is a global *access control list (ACL)* that defines security policy for every untrusted device driver. This security policy is enforced by a global *reference monitor* which is in both cases part of the

TCB. Some of the shortcomings of Minix’s ACL system are patched by *memory capabilities*. Each memory capability gives its holder the permission to manipulate specific region of the virtual memory of some specific process. This security mechanism cannot be simulated within the proposed ACL.

We have failed to find a reason why we should support the ACL concept. We enforce all our security policies by starting untrusted subsystems in a sandbox. Proper set of capabilities are then passed to individual untrusted subsystems by our trusted initialization code.

There is an ongoing positive trend to retrofit existing safe programming language to a point when they can be used for building defensively consistent systems. Some of the retrofitted programming languages are Ocaml [20], Java [21], JavaScript [22], Mozart/Oz [23].

We model and reason about authority of individual subjects informally. Alfred Spiessens, in his PhD thesis [24], proposes a formal treatment.

X. CONCLUSION

There is still a significant amount of work to be done, but we already see the first signs that our goal—creation of a dependable operating system with the smallest *trusted computing base* (TCB)—can be created. The TCB already has a concrete shape; we were able to create various different untrusted extensions on top of it in concord with *principle of least authority*; but of course we cannot yet claim that what we have is a complete operating system. We expect that the current size of the TCB (1988 lines of code, see Figure 7) will increase as we will gradually add new functionality (new untrusted extensions on top of TCB) and as we will focus on more aspects of quality (e.g., performance), but there is a reasonable hope that we will produce a smaller TCB than any other team.

The three DoS attacks described above as well as some other ones are covered by Section 11.4.8 (Discussion) in the compiler’s technical documentation [1]. We described how can be new language constructs used to achieve defensive correctness. These scenarios also justify language’s semantics.

The P programming language is still too immature for release outside laboratories. Nevertheless, we were curious about comprehensibility of some of the employed concepts. We have therefore decided to spend some time communicating them to others and to receive some preliminary feedback. We were working with four students. Their innocent viewpoint helped us to identify problematic parts of the language. As a result, we have moved from tagged 31-bit integers to 32-bit integers and we plan to abandon isorecursive types in favor of equirecursive types. They also corroborated our theory that concepts such as *permission*, *authority*, *designation*, *capability*, *object-capability security model*, *pi-calculus*, *functional programming*, *principle of least authority*, *type system*, *sandbox* and *powerbox* can be

easily explained to novices. These students have learned how to provide expected functionality without the need to trade it for security [25–28].

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] M. Košík, “Implementation of the P compiler and its runtime,” 2011, <http://bit.ly/fn6YQ2>.
- [2] —, “DC2: A defensively correct operating system,” 2011, <http://bit.ly/kgKhga>.
- [3] B. C. Pierce and D. N. Turner, “Pict: A programming language based on the pi-calculus,” in *Proof, language and interaction: Essays in honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, <http://bit.ly/lfq70F>.
- [4] D. Sangiorgi and D. Walker, *The pi-calculus: A theory of mobile processes*. Cambridge University Press, Jul. 2001.
- [5] M. S. Miller, “Robust composition: Towards a unified approach to access control and concurrency control,” Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
- [6] X. Leroy, “Formal verification of a realistic compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.
- [7] Y. Bertot and P. Casteran, *Interactive theorem proving and program development*, ser. Texts in theoretical computer science. An EATCS series. Springer-Verlag, 2004.
- [8] A. Bogk, “Defense is not dead,” *27th Chaos Communication Congress*, 2010, <http://bit.ly/g99y7W>.
- [9] M. Košík, “Taming of Pict,” in *SOFSEM*, ser. Lecture Notes in Computer Science, V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, Eds., vol. 4910. Springer, 2008, pp. 610–621.
- [10] I. Sommerville, *Software engineering*, 7th ed. Addison Wesley, 2004.
- [11] B. C. Pierce and D. N. Turner, “Pict language definition,” 1997, <http://bit.ly/fHoRDJ>.

- [12] B. C. Pierce, “Programming in the pi-calculus: A tutorial introduction to Pict,” 1997, <http://bit.ly/fD8Ft5>.
- [13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Countering IPC threats in multiserver operating systems,” in *Accepted for publication at 14th pacific rim international symposium on dependable computing (PRDC '08)*, Taipei, Taiwan, Dec. 2008.
- [14] A. S. Tanenbaum and A. S. Woodhull, *Operating systems: Design and implementation*. Pearson Prentice Hall, 2006.
- [15] “Minix 3, Google summer of code ideas 2011,” <http://www.minix3.org/soc-2011>.
- [16] “Genode operating system framework,” <http://bit.ly/gSh3cu>.
- [17] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, “Deconstructing process isolation,” in *MSPC '06: Proceedings of the 2006 workshop on memory system performance and correctness*. New York, NY, USA: ACM, 2006, pp. 1–10.
- [18] 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*. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–15.
- [19] G. Klein, “Correct OS kernel? Proof? Done!” *USENIX login*, vol. 34, no. 6, pp. 28–34, 2009.
- [20] M. Stiegler, “Emily: A high performance language for enabling secure cooperation,” *International conference on creating, connecting and collaborating through computing*, vol. 0, pp. 163–169, 2007.
- [21] 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*. New York, NY, USA: ACM, 2010, pp. 1–7.
- [22] “Google Caja,” <http://code.google.com/p/google-caja>.
- [23] F. Spiessens and P. Van Roy, “The Oz-E project: Design guidelines for a secure multiparadigm programming language,” *Multiparadigm programming in Mozart/Oz*, pp. 21–40, 2005.
- [24] A. Spiessens, “Patterns of safe collaboration,” Ph.D. dissertation, Université catholique de Louvain, Belgium, Feb. 2007.
- [25] M. Kallo, “Powerboxed video card driver,” Master’s thesis, Slovak University of Technology in Bratislava, 2008.
- [26] J. Valo, “Ovládače jednoduchých zariadení v jazyku Pict,” Bachelor’s Thesis, Slovak University of Technology in Bratislava, 2008.
- [27] M. Hečko, “Ovládače jednoduchých zariadení v jazyku Pict,” Bachelor’s Thesis, Slovak University of Technology in Bratislava, 2008.
- [28] O. Kallo, “Ovládače jednoduchých zariadení v jazyku Pict,” Bachelor’s Thesis, Slovak University of Technology in Bratislava, 2009.