ProvenCore: Towards a Verified Isolation Micro-Kernel

Stéphane Lescuyer
Prove & Run
77, avenue Niel
Paris, FRANCE
stephane.lescuyer@provenrun.com

ABSTRACT

We report on an ongoing project aiming at a fully secure micro-kernel named ProvenCore. This operating system is both developed and specified in a single specification language called Smart. The Smart models are used to generate efficient C code and express low- and high-level properties of the implementation, and first among them guarantees of integrity and confidentiality for the various processes running on the kernel. ProvenCore is designed to be used as a secure world operating system in mobile devices, beneath a professional application platform or a Trusted Execution Environment.

Keywords

Separation Kernel, Isolation, Formal Proof, Certification Toolchain

1. INTRODUCTION

As formal methods for software verification gained in scalability and maturity, they became ripe for use in industrial applications as a complement or a replacement of other validation techniques on critical pieces of software. Their use is acknowledged by official standards, such as in the latest version DO-178C of the norm for airborne systems [7], or in evaluations at the highest levels of the Common Criteria (CC) [1].

Operating systems bear a central role in computing systems, in that all software that runs on the system depends in one way or another on the OS, as much as on the hardware beneath it. Also, operating systems typically run in privileged modes in which there is no protection from some faults and where bugs can have arbitrary effects on the system. Therefore, formally verifying an operating system is a key step in achieving a trustworthy software architecture, in particular in bringing full meaning to formal verifications of other components that would run on said OS. In a similar way, compilers are key components because they bridge the gap between the sources and the actual executable binaries. This issue has been addressed in different ways, by providing a full-fledged certified C compiler such as CompCert [11], or by a case-by-case equivalence proof between source and binary code, as in seL4 [14]. Much effort has been invested to address the issue of operating systems as well, targeted specifically at *micro-kernel* architectures (seL4 [10], PikeOS [3], OLOS [13]) or hypervisors/virtualization plaforms (*baby hypervisor* [6]).

Modern handsets and tablets have become ubiquitous and their usage has evolved to encompass more and more sensitive applications, like *e-banking* and digital rights management (DRM). Both users and service providers require secure environments for that sort of applications, unfortunately typical high-end operating systems for mobile devices are way too large to be amenable for formal verifications. The ARM TrustZone technology [5] adds specific hardware support to ARM processors that separates the execution platform in two worlds: the "rich" or "normal" world on one side, and the "secure" world on the other side. A simple ultra-privileged mode called "monitor" mode allows control to switch from one world to the other. Rich operating systems like Android or iOS can run on the rich side without modifications, along with their vast range of user applications on top, whereas security-critical services can run in the protected secure world. This secure world requires its own operating system, which need not be as versatile as the one on the rich side. This architecture can for instance be used to separate the personal and professional worlds on a single handset, thus allowing a secure *Bring Your Own Device* (BYOD) policy. Another possible application is to equip the secure world with a Trusted Execution Environment (TEE), as specified by Global Platform [4], which acts as a specific kernel for secure services called Trusted Applications, such as DRM management, cryptologic functions, etc.

This article presents ProvenCore, a micro-kernel currently in development at Prove & Run which aims at being formally proven and suitable for use on the secure side of a TrustZone architecture, either as the basis of a BYOD stack (see Fig. 1) or a TEE. In Section 2, we present the most salient features of ProvenCore, as well as the security policies it can implement, and its high-level properties. The development of ProvenCore is performed in a specific language and toolchain also developed by Prove & Run, which we detail in Section 3. Finally, in Section 4, we sketch the architecture of the formal proof of the ProvenCore's functional specifications and isolation properties.

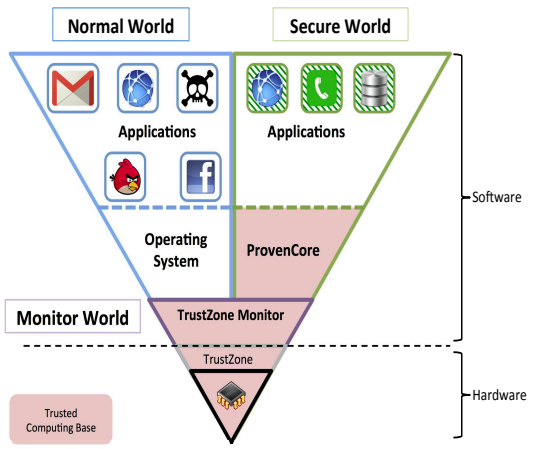


Figure 1: Overview of a possible BYOD architecture based on TrustZone and ProvenCore

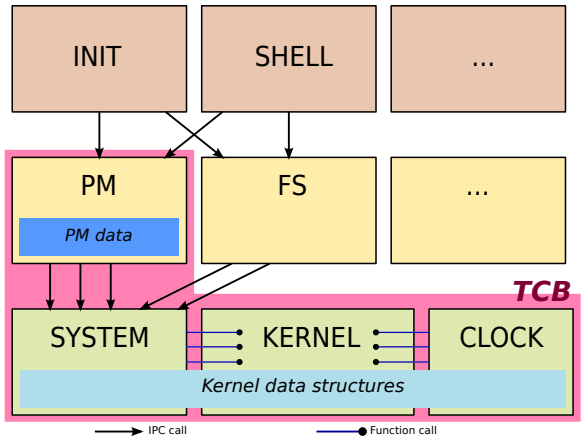


Figure 2: Minix 3 architecture, with TCB outlined

2. ProvenCore

Overview. ProvenCore is a micro-kernel that is largely inspired by Minix 3.1 [15]. Unlike Minix 3.1 which was designed for x86 processors and used segmentation, ProvenCore is designed for ARM architectures and uses Memory Management Unit (MMU) to manage virtual address spaces for the various processes and ensure their isolation. We chose Minix to start with because of its micro-kernel architecture, the way it is relatively simple and well documented, yet versatile and efficient enough for the applications we have in mind.

The micro-kernel architecture is essential to make a formal proof possible since by design it allows to reduce the Trusted Computing Base (TCB) to a bare minimum, and therefore the amount of code that must be formally specified as well. Indeed, micro-kernels typically only have the critical services running in privilege mode, the actual "kernel", while other secondary features, in particular file systems or drivers, run as unprivileged services over said kernel. Note that the TCB is not necessarily reduced to the privileged layer, because the kernel may depend in one way or another on one of the unprivileged services, in which case this service belongs to the TCB as well. It is a difference between pure micro-kernel design for software modularity, and micro-kernel design for proof containment: that some feature may be moved to some unprivileged service does not entail that it is not in the TCB, the kernel should not depend on it or be sufficiently defensive with respect to the values returned by this service. Figure 2 shows the typical architecture of Minix, with the lowest layer being composed of three processes called the *kernel tasks*, the intermediate layer of so-called *system processes*, user-mode services such as the Process Manager (PM) or the File System (FS), and the top layer of regular user processes. System processes and kernel tasks communicate via message-passing IPCs, and the PM in particular is in charge of much work, e.g. telling the kernel where to allocate new processes, where to copy data and code when on `fork()` and `exec()` system calls. The kernel trusts the PM, in that it does whatever is required of him by the PM, which indubitably makes the PM part of the TCB, as we outlined on Fig. 2: the PM code does not need to use privileged instructions, yet it is critical for the separation that the kernel must guarantee. On the contrary, we made sure the FS and the various drivers needed not be trusted by the kernel, by making the latter more defensive. In essence, the proofs we perform make no distinction between system processes and user processes, and a malicious system process cannot bring the system down anymore than a user one.

Another issue with the Minix 3 TCB is that it therefore consists in four different processes, communicating with each other with the generic IPC mechanism, three of which (the kernel tasks) run with interrupts disabled and the PM running with interrupts enabled. This means that the TCB is everything but sequential code, and makes formal reasoning about its behaviour on the source code level rather impractical. For instance, we identified that some invariants of the system implicitly depended on the PM having higher priority than other system processes, which in itself was not enforced by the kernel since the PM could be subject to preemption and priority penalties. For these reasons, we moved the PM to the lowest layer and merged all four processes in a single one[1]. Doing so, we reach an architecture with a sequential TCB suitable for program verification, as in Fig. 3.

A sequential TCB means in particular that all the code in the TCB runs with interrupts disabled, which is not an issue in our setting since there are no real-time constraints for the ProvenCore applications that we have described above.

Features. ProvenCore is a rather general-purpose micro-kernel and provides the following features:

- creation, deletion of processes;
- execution of programs taken from a given set of codes in the kernel image, that constraint arising from the

[1]The PM aside, the kernel tasks have been merged in a similar way in the most recent versions of Minix.

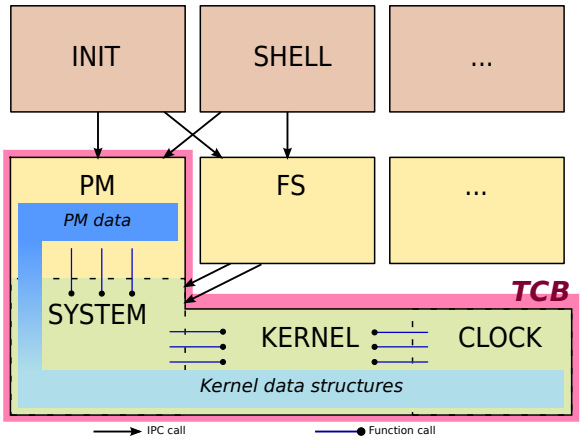


Figure 3: Minix 3 TCB made sequential

need to not depend on a file system, and completely reasonable in a "closed" setting like a TrustZone secure world, where available codes will be checked and authorized beforehand by the OEM for instance;

- synchronous message-passing inter-process communications with timeouts;
- asynchronous notifications, also used to notify user-space drivers of hardware interrupts;
- process-to-process data copies, guarded by a precise system of authorities managed by the processes themselves using dedicated system calls;
- a shared memory system *à la* System V to allow potential zero-copy transfers, which can be essential when dealing with DRM for instance.

The kernel also manages various authorities and privilege structures that enable him to enforce a certain number of *security policies*. Some of these are static and part of the initial kernel configuration, but can be reconfigured for each concrete architecture without invalidating the proofs from Section 4, while others are added and removed at run-time by the processes themselves. The security policies that are enforced by ProvenCore are as follow, using the classification given in the MILS architecture document [2]:

Resource allocation policy There are two sides to resource allocation, one for time and one for space:

- time resource policy is rather limited: ProvenCore's scheduler uses round-robin with priority queues and the static configuration can specify a maximum priority for each process independently; at run-time, a process can try and change its own priority, but it won't go beyond the maximum authorized priority;
- the amount of physical pages that some code can allocate can be limited at a process' creation, to allow for instance a TEE to limit how much physical memory each of its TA can pretend to.

Access control policy Access to the various system calls is configured on a call-by-call, process-by-process basis so that it is possible to reserve certain system calls for some processes in the image, for instance to forbid forking or IRQ registering to Trusted Applications.

Moreover, process-to-process copies are controlled by a mechanism of *authorities* which specify where a copy can read or write, who can ask for a copy, who can manage the authority itself, etc. These authorities are modified by the processes as they see fit, and managed by the kernel to ensure that every process-to-process data transfer has been agreed upon by both endpoints beforehand.

A similar system protects the shared memory machinery as well, albeit with slightly different constraints. For instance, ProvenCore forbids that two processes have write access to the same page at the same time, which is necessary to preserve a strong isolation property.

Information flow policy Information flow is controlled by statically configurable data on a process-by-process basis, and can be used to allow or disallow IPCs in both ways, or just one way, or just allow asynchronous notifications, etc. This mechanism is used by Minix originally to enforce that synchronous IPCs go "downward" in the architecture of Fig. 2, to prevent deadlocks altogether and also to prevent system processes or kernel tasks to be stuck waiting for a less privileged process. ProvenCore keeps this mechanism, in particular the TCB process can only perform asynchronous notifications, and can only receive a kind of *ask-and-wait-for-a-reply* IPC so that it is ensured that processes asking the kernel for something will politely wait for an answer.

Isolation. To conclude this presentation of ProvenCore, we explain what exactly are the high-level properties that it should verify, and why these properties in particular. The typical property of interest for such a system is *non-interference* [12]; unfortunately, it is typically not true as is because on one hand processes do interfere willingly via IPCs and copies, and on the other hand the finiteness of physical resources makes the existence of processes potentially observable by all. The main property of ProvenCore is the *isolation* property, which can be divided into the two following properties, each being a limited form of non-interference:

Integrity Integrity ensures that the resources of a process (code, data, registers) cannot be tampered by other processes, unless said process gave explicit authorization therefor.

Confidentiality Confidentiality ensures that the resources of a process (code, data, registers) cannot be observed by other processes, unless said process gave explicit authorization therefor.

Integrity ensures that a process will run as if it were alone on the system, until it actually decides to interact with another

process. In particular, it can be preempted and rescheduled and will see no difference in its state. Confidentiality on the other hand ensures that provided that it does not send one of its secrets to other processes, the process can change its secrets without other processes being able to depend on this actual change.

Isolation is our main property because it is fundamentally what the processes cannot guard against on their own. Of course we expect the kernel to perform according to reasonably precise functional specifications. Nonetheless, theoretically speaking, the kernel could send IPCs to the wrong endpoints, or copy data from the wrong sources, and still processes could come up with counter-measures such as encryption, signatures, authentication protocols. What they will always rely on, though, is some sort of safe (in the sense of both integrous and confidential) sandbox to perform their computations. This is why we single out the isolation property, even if the proofs we will present in Section 4 also establish functional specifications that ProvenCore verifies.

3. ProvenTools

Language. ProvenCore is implemented in a language called Smart and developed by Prove & Run. Smart lets one write both the implementation and the specifications, including the various properties, axioms, auxiliary lemmas, and so on. Smart is a strongly-typed polymorphic functional language with algebraic datatypes (structures and variants). A detailed presentation of the language is out of the scope of this article, but the main specificity of the language is that it helps separating data-flow and control-flow in programs. For instance, each *predicate* (the name for a function in Smart) can have an arbitrary number of outputs, associated to an arbitrary number of *labels*, which act as exit codes. Labels can be used in many ways, the two most frequent being in stead of boolean return values or as exception values. Control is performed by catching and transforming labels, and predicates may return some outputs only with some labels, which for instance makes it impossible to use meaningless outputs in exceptional cases.

Part of the language is reserved for logical specifications: one can define pre- and postcondition contracts, local assertions and loop invariants in predicates. One can also define logical properties or inductive predicates to help in proofs, the latter being the only part of the language that cannot be transformed in executable code. Finally, one can assert or prove logical properties by writing hypotheses or lemmas/theorems. As an example, here is a simple auxiliary lemma on one of the abstract models of ProvenCore:

```
public lemma not_running_decrease_writable
  (state s, prediction pi, event e,
   state t, nat i, address p)
program {
  istep(s, pi, e, t) =>
  !is_running(e, i) =>
  writable(t, i, p) =>
  writable(s, i, p);
}
```

This lemma expresses that given a transition of the system from a state s to a state t, summarized by a trace e, and provided that i was not the index of a process running on the system during that transition, then the set of addresses in i's address space that are writable by some other process can only decrease. It is naturally a key piece of the integrity property, and quite intuitive as well: access rights in i should not be created whilst i is blocked. It can decrease though, typically if i was synchronously waiting for a message and got it during that very transition, which removed the access rights for the sender on the same occasion.

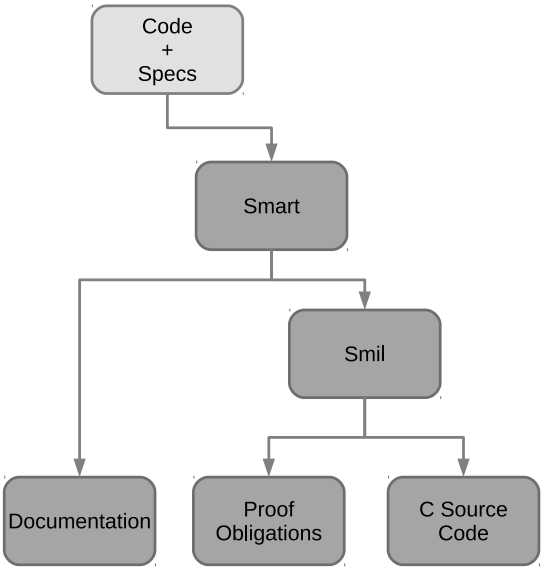


Figure 4: ProvenTools tool-chain overview

Tool-chain. Prove & Run develops a complete tool-chain around Smart, in the form of a set of Eclipse plugins. Altogether, these plugins provide a complete IDE for editing Smart models, browsing the proof obligations, proving them and generating executable code. They include the many features that people have come to expect from IDEs: content assist, refactoring, quick browsing between references and declarations, etc.

Figure 4 shows what happens to Smart models once they have been entered in the tool: Smart sources (containing both code and specs) are compiled into Smil, the *SMart Intermediate Language*, a simpler form of the language which is used as a starting point for the two main components exploiting Smart models: the prover and the C generator.

Prover. ProvenTools generates *proof obligations* for all logical lemmas, theorems, preconditions, assertions, invariants that appear in the Smart models. It also generates obligations for labels that remain unhandled in the code, forcing one to handle all possible exit cases of all predicates, or to prove that they are actually impossible. These proof obligations are presented to the user in a special Eclipse view next to the Smart editors, and can be proved semi-interactively by a combination of the following:

- a dedicated automated prover that uses already proved obligations or assumed hypotheses to try and discharge new obligations automatically;
- one of the many manual *hints* that can be applied interactively in the IDE to the proof obligations;
- an oracle which tries to apply the manual hints automatically in the background, based on user-defined strategies, and will propose a possible proof as soon as it finds one; it differs from the automated prover in that it is not trusted and the proofs it finds are retried as if they had been done manually.

Whether manual or automatic, all proof steps can be browsed in detail in the tool, which makes a possible review of the proofs possible, unlike what happens when using black boxes as back-end solvers. When code maintenance or evolution breaks some proofs, a helpful system assists the user into *adapting* former proofs to the new changes to make it as easy as possible to iteratively maintain a piece of code and its specifications, which may otherwise (and still is in some cases) a very time-consuming task.

C Generator. The other key component that interprets Smil programs is the generator of actual C code. It is indeed convenient to write the code and the specs in the unified Smart language, but the executable part of the Smart models must then be transformed to executable code.

As explained above, Smart is a purely functional language, which helps the formal reasoning in many ways, but the functional computation model is not adequate for the actual code, especially for kernel code as in the case of ProvenCore. To that end, the C generation plug-in in ProvenTools try and replace functional modifications of structures in the models by in-place updates, thus effectively transforming the functional implementation to an imperative one. Of course this is only correct if the Smart code handles its different values *linearly*, i.e. without never trying to read a "former" value after applying a functional update on it. These constraints are actually quite natural once a certain coding discipline is acquired, and they do not apply to Smart predicates that are only used logically, in particular in the abstract models used in the refinement proofs (cf. Section 4). A typical example of potentially dangerous code is that of a loop which modifies various cells in an array (e.g. by following a linked list from one cell to another), keeping a reference to some cell (e.g. the head of the list), and reusing this cell after the loop without first re-reading it. Stronger but similar constraints are enforced by the typing mechanism in the memory model used in VCC and presented in [8]. Instead, the C generation plug-in performs various global static analyses to make sure that it is safe to transform the code to imperative style, and reports errors (or introduces copies, depending on whether the user said copies were acceptable) when it cannot make sure it would be safe. Extra analyses also allow to globalize objects if there are at most one live instance at any time, and remove *ghost* parts from the generated code. In earlier experiments, we were able, with this method, to generate C code for a complete model of Minix 3.1 that did not require any dynamic allocation, and ran at a speed comparable to the original C code.

The program transformations detailed above are not trivial by any means and it is legitimate to wonder about their correctness. First of all, we plan on targeting CompCert's input subset of C in our generator so as to be able to use the certified compiler to reduce the gap between the generated C code and the corresponding binaries. Also, CompCert's input semantics is formally specified and we are currently engaged in a project with the CompCert team that involves formally studying the gap between Smil and their input language, aiming at proving our C generator with respect to CompCert's semantics. The good thing about this approach is that we keep models effect- and pointer-free to ease the reasoning, and push the burden of bridging the gap between the functional and imperative worlds on a set of transformations and static analyses, which could be proved correct once and for all.

Documentation. One of the goals of the ProvenCore project is to achieve EAL7 certification of the kernel, and therefore one of the objectives for the ProvenTools is to help achieve high-level certifications. As explained above, all proof steps are kept and presented explicitely in the tool, to help reviewing the various arguments used in proofs if necessary. Documentation of the code and its specifications is also fundamental in a certification effort, and Smart has a way to attach structured documentation to all declarations in the languages, which the tool can use to display information about declarations. Documentation is especially important on all hypotheses, to the point that there is a special view to browse all the various assumptions made in a workspace and report those that are undocumented. Assumptions take two forms in Smart:

- hypotheses, which are actual logical results that are assumed;
- some Smart predicates and types are called *implicit* because they have no actual Smart implementation, but are just declarations of external implementations, typically axiomatized in Smart and realized in C or assembly.

For instance, the following shows an excerpt of a polymorphic axiomatization of a type `set<A>` of finite sets of elements of type `A`:

```
public mem(set<A> s, A e) -> [true, false]
implicit program

public add(set<A> s, A e, set<A> t+)
implicit program

public hypothesis add_mem_1(set<A> s, A e)
program {{ set<A> t }} {
  add(s, e, t+) => mem(t, e);
}
```

It shows two implicits `mem` and `add` for testing membership in a finite set and adding an element to a set, and one axiom `add_mem_1` specifying that after being inserted, an element should belong to the new set. Implicit predicates are assumptions because they create constraints on the explicit

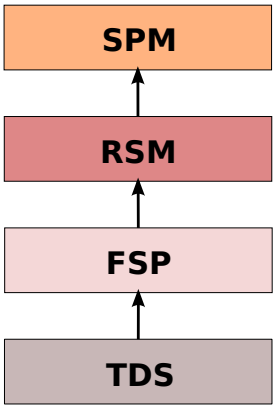


Figure 5: The refinement chain

implementation provided in C, and the external implementations must be reviewed as such. In the finite set exemple for instance, an actual implementation would require membership to be decidable, and insertion to be a total operation. To that end, the tools can help by forwarding the documentation associated to the implicits in Smart to their external implementations. More generally, we are working on enhancing the tools to provide more help to automatically create some of the documents required for a high-level certification.

4. PROOFS

In the former sections, we have presented ProvenCore and its required properties, as well as the context in which we are developing it. This section gives a short description of the way in which the properties are proved and how the proofs are organized.

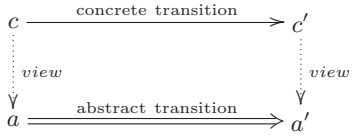
Refinements. The proof proceeds by establishing *refinements* between successive models, from the most abstract to the most concrete, the latter corresponding to the actual model used for code generation, and the former the one where the high-level priority, in our case the isolation property described in Section 2, is defined and proved. The different models in our refinement chain are shown in Fig. 5. This use of various abstract models, each more abstract than the previous one, has several advantages:

- the different layers provide some separation of concern in the overall proof, with the lower-level proofs cluttered with low-level properties and invariants, and devoid of functional properties, and the higher-level centered on functional specifications;
- each layer of abstraction makes it possible to remove details that do not make sense anymore and would only artificially complicate the models and the proofs if they were kept;
- each layer of abstraction makes it possible to change the representation of the states of the transition system in order to internalize in the structure of the state some invariants of the lower layer;
- having a simple top-level abstract model leads to high-level properties that are easier to express, and more importantly easier to *read*.

Incidentally, a high-level CC evaluation will require different levels of description of the system being evaluated, and we have given our own abstract models the names based on the different CC levels of description which reflect their respective roles:

- the *Security Policy Model* (SPM) is the most abstract level and the one at which we express and prove the isolation property, it models the kernel as an abstract controller and the various processes as abstract machines each enjoying their own independant physical resources (see Fig. 7);
- the *Refined Security Model* (RSM) is not present in the CC, and acts as an intermediate step in our proofs to split the rather impractical gap between FSP and SPM; in contrast to the SPM, the RSM machines all share the same physical ressources, managed by the controller (see Fig. 6);
- the *Functional SPecifications* (FSP) is a model roughly equivalent to the TDS in functionality but using data structures and algorithms that are much easier to reason with; its main functional difference with the TDS is that it uses a linear view of the RAM similar to the RSM, i.e. MMU address translation has disappeared;
- the *Target of evaluation DeSign* (TDS) is the model that is used to generate the actual C code, and therefore it contains the sequential Smart code of the kernel, as well as models for hardware components (such as interrupts or MMU) which are not turned into C code but are necessary to complete the TDS specifications.

For each refinement, we proceed by defining an abstraction *view*, i.e. a function from the concrete model state to the abstract model state (represented by the upward arrows in Fig. 5), and then show a correspondence or *commutation* lemma that establishes that transitions from c to c' in the concrete model entail transitions from the view of c to the view of c' in the abstract model. It can be summarized by the following commutation diagram:



This involves showing that the views actually exist, since the view is typically not a total function. That way, as we climb towards the higher levels, we reach models that are simpler and more flexible than the TDS but that still simulate all its possible behaviours. For instance, here is the signature of the partial view from RSM to SPM, along with the corresponding commutation lemma:

```
public view(rsm rsm, spm spm+) -> [true, failed]
// Exit with failed if the view couldn't be built
program { ... }
```

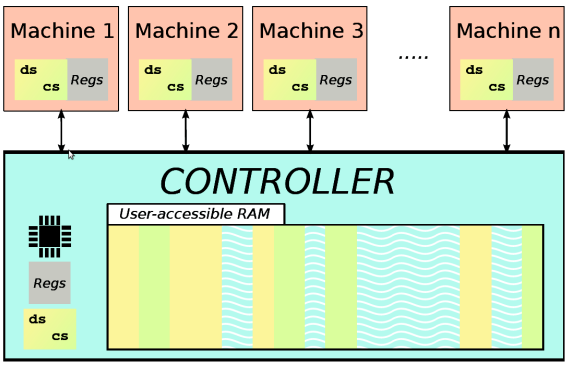


Figure 6: Refined Security Model

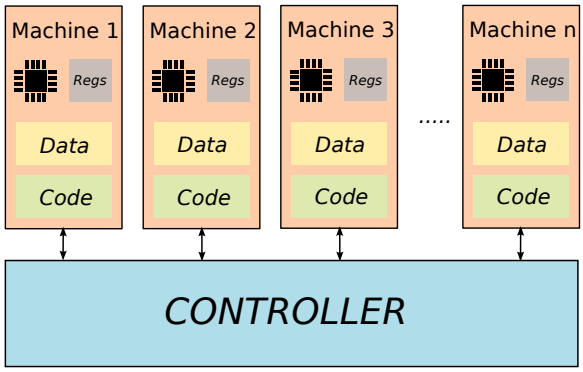


Figure 7: Security Policy Model

```
public lemma view_step(rsm rsm, rsm_pred pi)
// The commutation lemma between RSM and
// SPM transitions
program {{ event e, spm_pred vpi,
  rsm rsm2, spm spm, spm spm2 }}
{
  wf(rsm) =>

  ?rsm.sts.step(rsm, pi, e+, rsm2+) =>
  [failed : error] view(rsm2, spm2+) =>

  view_prediction(rsm, e, pi, vpi+) =>
  [failed : error] view(rsm, spm+) =>
  [failed : error]
    spm.sts.step(spm, vpi, _, spm+) =>

  spm = spm2;
}
```

In this lemma, `rsm.sts.step` represents a concrete transition, and `rsm` and `rsm2` play the respective roles of c and c' in the above diagram, whereas `spm.sts.step` is an abstract transition and `spm` and `spm2` correspond to a and a'. This commutation lemma only holds for well-formed RSM states, as indicated by the premise `wf(rsm)`, and also expresses that the views will not fail, by transforming their `failed` labels into `error` labels, which is one way of asserting that something cannot happen and must be proved. Note that the SPM transition function is not total either, and that this lemma proves that the transition between the views exists as well.

Model Reuse. The refinement chain presented above can be very handy to reuse part of one proof effort in another. There are mainly two ways that they can be used:

1. When designing an abstract model like the SPM, it is normal to consider the high-level property for which the SPM is being designed and simplify the model accordingly. This may mean for instance that functionalities that are irrelevant for the properties envisioned may be discarded. Nonetheless, these functionalities may be of interest in other setting, for another high-level property, in which case it is possible to just "branch" another SPM from the last level where the functionality is still present. For instance, in ProvenCore, the scheduling disappears between the FSP and the RSM, and if we wanted to certify properties about the scheduler, we could write another dedicated SPM and prove its refinement to the FSP.

2. It is also possible that a security policy model or functional specifications are flexible enough that they may account for different implementations. For instance, ProvenCore's SPM is very general and could be a suitable SPM for many separation kernels. Of course, the lower one can "graft" in the refinement chain, the more proofs can be reused.

We actually have experienced the second kind of proof factorization possibilities above. In an earlier proof of concept, we performed an isolation proof on Minix 3.1 on x86 with segmentation from the SPM down to the FSP. When starting work on ProvenCore, we noticed that by making the FSP slightly less architecture-dependent, it could be flexible enough to refine both to the original Minix for Intel with segmentation and to ProvenCore's TDS. This means that by taking sufficient care in the initial design of the FSP, one can share most of the proofs for the certification of an OS and to a port to another architecture. The lower-level refinements are the harder to prove of course, but still this may spare a lot of extra work. For instance, we spent about 3 person-years for the initial SPM-to-FSP proof of Minix 3.1, and we estimate the TDS-to-FSP refinement in ProvenCore to roughly the same amount. This is a reasonable time overall: seL4 is reported to have required 11 person-years, to which 1.5 person-year were recently added [9] in order to achieve a proof of isolation of user processes.

5. CONCLUSION

We have presented ProvenCore, a micro-kernel being developed in a dedicated specification language Smart along with its functional specifications. The fundamental properties enjoyed by ProvenCore are the integrity and confidentiality of processes. Its features would make it an ideal candidate for a secure operating system on a TrustZone aware mobile device.

6. REFERENCES

[1] Common criteria certification. `http://www.ssi.gouv.fr/en/certification/common-criteria-certification/`.

[2] Euro-mils: Secure european virtualization for trustworthy applications in critical domains. http://euromils.eu/downloads/2014-EURO-MILS-MILS-Architecture-white-paper.pdf.

[3] PikeOS. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[4] Trusted execution environment. http://www.globalplatform.org/specificationsdevice.asp.

[5] TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[6] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2010.

[7] C. C. Ben Brosgol. Do-178c: A new standard for software safety certification. http://www.ieee-stc.org/proceedings/2010/pdfs/bmb2623.pdf, 2010.

[8] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for c. *Electron. Notes Theor. Comput. Sci.*, 254:85–103, Oct. 2009.

[9] M. Daum, N. Billing, and G. Klein. Concerned with the unprivileged: User programs in kernel refinement. *Formal Aspects of Computing*, 26(6):1205–1229, oct 2014.

[10] 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[11] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[12] J. Rushby. Noninterference, transitivity, and channel-control security policies, dec 1992.

[13] M. Schmidt. *Formal Verification of a Small Real-Time Operating System*. PhD thesis, Saarland University, Saarbrücken, 2011.

[14] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013.

[15] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*. Pearson, third edition, 2006.