What If You Could Actually *Trust* Your Kernel?

Gernot Heiser, Leonid Ryzhyk, Michael von Tessin, Aleksander Budzynowski
NICTA and University of New South Wales

Abstract

The advent of formally verified OS kernels means that for the first time we have a truly trustworthy foundation for systems. In this paper we explore the design space this opens up. The obvious applications are in security, although not all of them are quite as obvious, for example as they relate to TPMs. We further find that the kernel's dependability guarantees can be used to improve performance, for example in database systems. We think that this just scratches the surface, and that trustworthy kernels will stimulate further research.

1 Introduction

Despite depending on computer systems for an increasing part of our lives, we have grown used to them not being particularly reliable. We seem to have accepted hacked servers and virus-infected desktops as an inevitable part of life.

Where dependability is important, we go to (at times extraordinary) lengths to reach some approximation of it. For example, security evaluations under Common Criteria [12], used primarily in national-security systems, rely on expensive processes, and at the most thorough levels also a degree of formal methods. However, even with this very expensive approach, national security depends in the end on testing and code inspection, which cannot give any guarantees.

Increasingly we don't even go that far, and instead trust molochs such as Windows, SELinux or Xen, and pretend this is fine as long as they sit behind a firewall.

In this paper we ask the following question: what could we do differently if we could actually *trust* the system, or at least its kernel? What new approaches, simplifications, efficiency gains could such a kernel enable?

Only a short while ago this question would have been considered hypothetical. This has changed with a few key achievements. One is the formal proof of the functional correctness of the implementation of the seL4 microkernel [13]. This showed that it is feasible to build (small) systems which can be guaranteed never to operate out-of-spec. In particular, seL4 is provably immune to such problems as code injection and privilege escalation. Proofs of general security properties, such as encapsulation, are in progress [14]. It is also possible to extend the guarantees it provides to a complete (albeit small) *trusted computing base* (TCB) [1].

Another piece of the puzzle is a verified compiler [16], which allows us to extend the correctness from the language to the hardware boundary. This means for the first time we can have a truly dependable software base for systems, and it makes sense to ask what we can do with it.

In the following we explore some areas of the design space of dependable systems built on seL4. These are, at present, mostly ideas, meant to stimulate further research.[1]

2 Virtual machine encapsulation could be trusted

These days, virtualization seems to be considered by many the cure for all ills, especially security [5]. This goes even to proposals to build a complete OS based on a hypervisor [20], using virtual machines (VMs) to encapsulate individual activities. A cynical observer might think that history is about to repeat itself, with hypervisors taking the place of OSes, and VMs replacing processes, only to end up at the same point in 10 years.

Somehow there seems to be a belief that a large code base, such as Xen [3] with its 1 MLOC or so, suddenly becomes trustworthy when it is called a hypervisor. Such a belief is totally unfounded. A study of OpenBSD has shown that 33 security vulnerabilities per MLOC were

[1] A public release of the kernel and its formal specification is available from http://ertos.nicta.com.au/software/seL4/.

found over a period of seven years [19]. This, of course, is a lower bound, as no-one knows how many vulnerabilities were left undiscovered.

The NOVA project [22] addresses this by reducing the TCB of a VM to 36 kLOC, a size where, with years of maturation, it is likely that less then a handful of vulnerabilities are left. This is a long wait, and at the end there is still no certainty. Some commercial ARM-based hypervisors are even smaller [7, 11, 17] and might shorten the wait.

However, only a formally verified hypervisor, such as seL4, can provide real security through virtualization, and on that basis, a system like CubeOS [20] starts making sense. Building it on a massive hypervisor such as Xen produces a fortress built on sand.

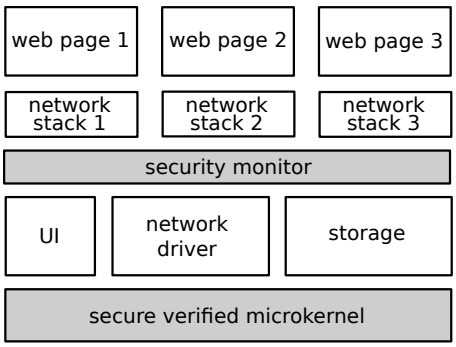

Figure 1: Secure web browser architecture. Components that belong to the TCB of the system are highlighted using grey shading.

3 Web browsing could be secure

Web browser security is a pressing problem in modern end-user systems. An endless stream of exploits has affected all major browsers and stimulated research into secure browser architectures.

Secure rendering of web pages requires the browser to enforce a security policy that restricts communication among different web pages as well as different objects inside a page. An important example is the *same origin policy (SOP)*, which means that pages from different sources cannot observe or alter each other's state and behaviour. In addition, scripts running inside a web page must be denied unauthorised access to OS resources, including file systems and network interfaces.

In traditional browser architectures, implementation of the security policy is scattered around the source code of the browser, leading to numerous vulnerabilities. Chrome [4], OP [8], and other recent browser implementations address this problem by encapsulating the security policy in a separate module, the *browser kernel*. The browser kernel starts an instance of the rendering engine for each page in a separate process inside an OS sandbox. Renderer processes can only access system resources and communicate with each other indirectly, via the browser kernel. This architecture ensures that a malicious web page cannot bypass the security policy implemented by the browser kernel.

The problem here is that this approach is still at the mercy of the OS, which is in the browser's TCB, and typically comprises tens of MLOC. A malicious web page that manages to take over the renderer process can then exploit a vulnerability in the OS to compromise the entire system.

IBOS [23] recently demonstrated that the TCB of a browser can be substantially reduced with a microkernel approach. IBOS is based on a modified version of the L4Ka::Pistachio microkernel [15]. It uses a process per open web page, encrypted storage, individual network stack instances per process, kernel-controlled secure screen rendering and user-level device drivers to minimise the TCB of the browser.

While this architecture presents a big step towards secure web browsing, it can be further improved to provide formal security guarantees. Our proposed design is shown in Figure 1. The trusted part of the system consists of the verified, general-purpose microkernel and a user-level security-monitor process. The monitor is responsible for instantiating browser processes with permissions to directly communicate only with their private network stack processes and the monitor itself. Similarly to the IBOS kernel, the monitor moderates communication between browser processes and implements packet filtering, storage encryption, DMA buffer management, and frame-buffer page mapping. The microkernel ensures that browser processes cannot violate constraints on communication imposed by the security monitor; for instance, that different browser processes cannot communicate directly, bypassing the monitor process.

This architecture can be extended to include a complete OS stack running inside a VM, alongside the browser, with isolation between browser tasks and the VM being enforced by the security monitor.

The monitor is the only part of the TCB, apart from the microkernel, that needs to be verified in order to formally prove that the resulting system correctly implements its security policy. The IBOS authors report that the implementation of the SOP policy in the Pistachio kernel is less than 9 kLOC. This is within reach of modern verification techniques [13, 14].

4 TPMs could be useful

The concept of a *trusted platform module* (TPM) [24] has been introduced by the *Trusted Computing Group*

(TCG) to enable secure boot [2], storage, authentication and *remote attestation*. In particular, the remote attestation facility provides evidence that a system is running a well-defined software configuration. This is achieved by accumulating hashes (called *measurements*) of software loaded into the system. TPMs are now widely available on PCs and other platforms.

However, for end-user devices the TCG approach to trustworthiness has been considered a failure [25]. There are two main reasons:

- Having a well-defined software configuration is not the same as having a *trustworthy* configuration [10]. In general, the software is still buggy, and the TPM only guarantees that the system is executing known (buggy) code rather than unknown (potentially malicious) code. Furthermore, if a bug in the software stack leads to loading code *without* prior measurement, remote attestation will report a software stack which differs from what is actually running on the system.

- The TCG approach cannot reasonably deal with the software variety and evolution common in real-world systems. Every version of every component in the TCB of the application produces a hash value which has to be pre-calculated, stored and compared against the actual value during attestation. The TCB includes any piece of code which can interfere with the operation of the application. If the OS cannot be guaranteed to be free of exploits, this necessarily includes *every* program loaded previously. Finally, the OS itself can be configured in many different ways, thus producing an untractable number of possible hash values.

As an example, consider a bank customer performing financial transactions from their smartphone or home PC. The attacks most likely to be successful are (1) phishing, resulting in the theft of customer's password or other authentication details, and (2) malicious code injection. A bank could completely rule out phished credentials from being misused if it could ensure that bank transactions originate from a specific machine (one of the client's pre-registered machines), and it could render malware harmless by rejecting any transaction which does not originate from a trusted software stack running on the client's machine. In theory, the TPM's remote-attestation functionality enables the bank to do this.

In practice, the bank's clients wants to configure their PC or smartphone to their taste, including installing arbitrary apps. Not only is the total number of apps too large for the bank to manage, but many of them will not be trustworthy and must therefore be kept out of the TCB (and therefore the system).

Existing TCG-based solutions therefore try to manage this problem by restricting the software stack, e.g. by allowing the user to only run a specific Linux image with a specific root file image and set of drivers, and only a narrowly-defined set of applications. This forces the user to boot into a specific banking-only OS environment that only supports a limited range of hardware.

An alternative solution, known as *dynamic root of trust measurement* (DRTM), allows the user to switch between trusted and untrusted environments at runtime. To this end, execution of the general-purpose untrusted OS is suspended until the trusted code has finished running. This is acceptable if the trusted code is small and self-contained and only runs for fractions of a second [18], which is not the case for online banking and other applications that require OS support.

Both of the above approaches effectively force the user to boot up a complete banking OS and suspend their own OS for every banking session. This is certainly not acceptable to an average user who wants to be able to switch back and forth instantly between the banking application and their normal environment.

The solution to this problem consists of: (1) isolating the TCB of the banking application from the rest of the general-purpose OS, and (2) restricting the TCB to small components that do not change, or only change rarely and in a controlled fashion.

This calls for a formally verified kernel such as seL4. Firstly, strong process-isolation guarantees provided by seL4 eliminate most software from the TCB. Secondly, a verified kernel changes rarely because no bug fixes are ever necessary, and the hashes of the remaining changes are manageable. Finally, verification can guarantee that all TCB components are measured before execution, thus ensuring that remote attestation is meaningful.

An implementation of this scenario runs seL4 as the lowest-level software component, acting as a hypervisor which runs a general-purpose OS in a virtual machine (VM). Isolated in another VM runs the banking environment. The banking VM has a verified loader (which therefore changes rarely and predictably). It loads a minimal OS personality, which includes drivers for mouse, keyboard and screen, as well as a crypto library. It also manages SSL endpoints, and links them to the TPM [6]. This allows the untrusted network stack of the general-purpose OS to be securely used for communication. On top of the mini OS runs the banking application. The TCB of the banking transaction changes rarely and is under control of the bank. Seamless UI integration could be achieved by secure sharing of the screen between VMs, as in CubeOS [20].

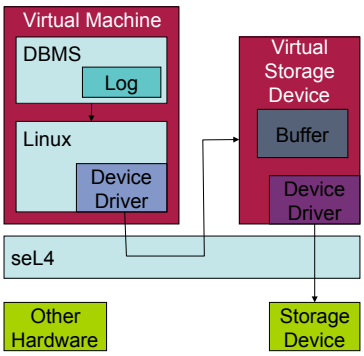

Figure 2: Database setup with virtual disk for buffering writes.

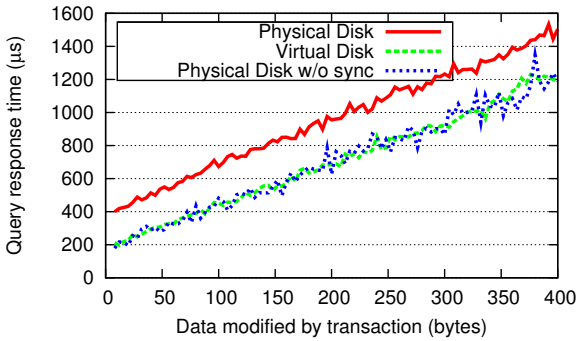

Figure 3: Comparison of different approaches to database transaction logging.

5 Database costs could be reduced

Database systems (DBMSes) are designed to maintain the ACID properties of transactions (atomicity, concurrency, integrity and durability) [9], despite faults in the underlying system. These days, they typically rely on RAID to protect them from disk failures, and a UPS protects from power failures. This leaves OS crashes as the most important failure to protect against.

Protection against OS failures is typically done by write-ahead logging: the DBMS records in an append-only log the intended modifications to its state before performing the actual state change. Persistence of the log is critical: the DBMS relies on log data to survive any system crash. This is done by blocking the commit of a transaction until it is known that the log data is safely recorded on persistent storage, typically by issuing a `sync` system call after writing the log.

This means that log writes are performed synchronously to transaction processing, putting disk write operations on the critical path of transactions.

The `sync` could be avoided, and any I/O performed asynchronously to transaction processing, if the DBMS could depend on the underlying OS not to crash at inopportune moments. A formally verified kernel would not crash, and thus a DBMS implemented to run directly on seL4 could safely buffer log data in memory until it has been written to disk asynchronously (and in larger batches, reducing I/O costs). However, this would require a significant re-write of the lower layers of the DBMS, and probably verification of much more code than just the microkernel.

Virtualization allows us to achieve the performance gains of asynchronous disk writes without a major rewrite. As shown in Figure 2, we can run an unmodified DBMS inside a virtual machine. The guest OS underneath the DBMS is also unmodified, except that its disk driver is replaced by a driver for a virtual storage device, which is implemented in a second VM. The guest OS's driver communicates with the virtual device via seL4 IPC.

The virtual storage device buffers the log data in RAM until it is written to a real disk, but the DBMS can continue processing transactions concurrently to the physical disk writes. The virtual device is simple enough to be considered dependable after undergoing thorough quality assurance; it could be formally verified if needed.

This setup can also protect against power failures if no UPS is available: The buffer in the virtual disk can be limited to a size which can safely be written to disk if a power failure is detected. Our measurements show that capacitances in standard server power supplies support system operation for more than 100 ms after mains power is lost, enough to write 1.5MiB of data.

The physical disk driver requires a bit of special consideration in this case: the DBMS depends on that driver not crashing while there is still log data to be written. The driver can be kept quite simple, as it only needs to support a minimal set of configuration and data transfer operations required for writing the database log (a separate driver can be used for database recovery). An appealing approach is synthesising the driver automatically [21], making it correct by construction.

Some initial data showing that this approach can indeed increase transaction throughput is shown in Figure 3. It compares transaction latency of MySQL running on virtualized Linux (Wombat) on seL4. "Physical" refers to Linux using a native driver to directly access the disk, "physical without sync" is the same setup but with I/O synchronisation suppressed at the end of the transactions. "Virtual disk" refers to the setup of Figure 2 where the Linux writes are buffered and written asynchronously. The graph shows that the latter two cases have about the same performance, yet the "physical without sync" setup sacrifices database durability in order to achieve this performance gain, while the "virtual disk" setup does not.

6 Conclusions

We have made a case that a formally verified OS kernel opens up new possibilities for system security. Some of the opportunities we discussed are being pursued already, although on top of platforms of dubious trustworthiness. As such, they can create an incorrect illusion of security, which can actually be worse than a well-understood lack of security. A verified, and thus really trustworthy, OS kernel can provide *real* security.

Besides these more obvious use cases, the trustworthy kernel opens up new opportunities. One is to make a TPM practically useful, via a system architecture which leverages the TPM for attestation but supports configuration changes and software evolution. Another one is to use the kernel's dependability guarantees to buy performance, by eliminating costly synchronous logging operations in databases.

In summary, the existence of a verified OS kernel has opened up new regions in the design space for systems. We are confident that we have only thought of a fraction of those, and encourage the research community to contribute more.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] ANDRONICK, J., GREENAWAY, D., AND ELPHINSTONE, K. Towards proving security in the presence of large untrusted components. In *5th SSV* (Vancouver, Canada, Oct 2010), G. Klein, R. Huuck, and B. Schlich, Eds., USENIX. 1

[2] ARBAUGH, W. A., FARBER, D. J., AND SMITH, J. M. A secure and reliable bootstrap architecture. In *IEEE Symp. Security & Privacy* (1997), pp. 65–71. 3

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *19th SOSP* (Bolton Landing, NY, USA, Oct 2003), pp. 164–177. 1

[4] BARTH, A., JACKSON, C., REIS, C., AND THE GOOGLE CHROME TEAM. The security architecture of the Chromium browser. Technical report, Stanford Security Laboratory, 2008. 2

[5] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *19th SOSP* (Bolton Landing, NY, USA, Oct 2003), pp. 193–206. 1

[6] GOLDMAN, K., PEREZ, R., AND SAILER, R. Linking remote attestation to secure tunnel endpoints. In *1st WS Scalable Trusted Comput.* (Alexandria, VA, USA, Oct 2006), pp. 21–24. 3

[7] GREEEN HILLS SOFTWARE. INTEGRITY secure virtualization. http://www.ghs.com/products/rtos/integrity_virtualization.html. 2

[8] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *IEEE Symp. Security & Privacy* (Oakland, CA, USA, 2008), pp. 402–416. 2

[9] HÄRDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *Comput. Surveys 15* (1983), 287–317. 4

[10] HEISER, G. Trusted ⇐ trustworthy ⇐ proof—position paper. In *1st Conf. Future Trust Comput.* (Berlin, Germany, Jul 2009), D. Gawrock, H. Raimer, A.-R. Sadeghi, and C. Vishik, Eds., Vieweg+Teubner, pp. 55–59. 3

[11] HEISER, G., AND LESLIE, B. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *1st APSys* (New Delhi, India, Aug 2010), pp. 19–24. 2

[12] *Information Technology — Security Techniques — Evaluation Criteria for IT Security*, 1999. ISO/IEC International Standard 15408. 1

[13] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *22nd SOSP* (Big Sky, MT, USA, Oct 2009), ACM, pp. 207–220. 1, 2

[14] KLEIN, G., MURRAY, T., GAMMIE, P., SEWELL, T., AND WINWOOD, S. Provable security: How feasible is it? In *13th HotOS* (Napa, CA, USA, May 2011). 1, 2

[15] L4KA TEAM. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/. 2

[16] LEROY, X. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *33rd POPL* (Charleston, SC, USA, 2006), J. G. Morrisett and S. L. P. Jones, Eds., ACM, pp. 42–54. 1

[17] LYNUXWORKS. LynxSecure embedded hypervisor and separation kernel. http://www.lynuxworks.com/virtualization/hypervisor.php. 2

[18] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *3rd EuroSys Conf.* (Apr 2008). 3

[19] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: Does software security improve with age? pp. 93–104. 2

[20] RUTKOWSKA, J., AND WOJTCZUK, R. Qubes OS architecture, version 0.3. http://qubes-os.org/Architecture.html, Jan 2010. 1, 2, 3

[21] RYZHYK, L., CHUBB, P., KUZ, I., LE SUEUR, E., AND HEISER, G. Automatic device driver synthesis with Termite. In *22nd SOSP* (Big Sky, MT, USA, Oct 2009). 4

[22] STEINBERG, U., AND KAUER, B. NOVA: A microhypervisor-based secure virtualization architecture. In *5th EuroSys Conf.* (Paris, France, Apr 2010). 2

[23] TANG, S., MAI, H., AND KING, S. T. Trust and protection in the Illinois Browser Operating System. In *9th OSDI* (Vancouver, BC, Canada, Oct 2010), pp. 1–15. 2

[24] TRUSTED COMPUTING GROUP. Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module. 2

[25] WAIDNER, M. An industry perspective on trusted computing. Invited talk at WS Scalable Trusted Comput., Oct 2010. 3