capDL: A Language for Describing Capability-Based Systems

Ihor Kuz, Gerwin Klein, Corey Lewis, Adam Walker
NICTA and University of New South Wales
Sydney, Australia
firstname.lastname@nicta.com.au

ABSTRACT

Capabilities provide an access control model that can be used to construct systems where safety of protection can be precisely determined. However, in order to be certain of the security provided by such systems it is necessary to verify that their capability distributions do in fact fulfil requirements relating to isolation and information flow, and that there is a direct connection to the actual capability distribution in the system. We claim that, in order to do this effectively, systems need to have explicit descriptions of their capability distributions. In this paper we present the capDL capability distribution language for the capability-based seL4 microkernel. We present the capDL model, its main features and their motivations, and provide a small example to illustrate the language syntax and semantics. CapDL plays a key role in our approach to development, analysis, and verification of trustworthy systems.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.4.6 [**Operating Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Design

Keywords

Capabilities, capability distribution, security analysis, seL4, Isabelle

1. INTRODUCTION

Capabilities [1] are a powerful approach to building secure systems. They provide an access control model that allows system designers to minimise authority of processes and that can be used to precisely analyse the protection state of such systems. This is particularly useful in the presence of security requirements that limit information flow and impose isolation between system components. In this paper we motivate the need for, and introduce our specific approach to, explicit descriptions of the capability distribution in such systems.

APSys 2010, August 30, 2010, New Delhi, India.

In a fully capability-based system, all objects, including resources such as devices and memory, and system objects, such as processes and communication channels, are referenced by *capabilities* – unique tokens that act both as references and provide access rights to objects. In order to access an object or perform an operation on one, a subject must hold a capability to this object, and the capability must provide sufficient rights for the operation. Capabilities may be transferred between subjects, meaning that the set of objects accessible by subjects can change over time. If a subject does not possess a capability to an object, and cannot ever acquire such a capability, it will not be able to access the object.

Since objects are only accessed through capabilities, they can be used to restrict a subject's access to only those objects that the subject requires to perform its tasks correctly, but to no others. This allows systems to be designed according to the *principle of least privilege* [9]. Furthermore, the capabilities in a system can be distributed such that they create distinct isolated subsystems, where subjects in different subsystems cannot influence or communicate with each other in any way.[1] Besides strict isolation, the capability model can also be used to create systems that allow limited inter-subsystem communication over authorised channels. This enables the construction of systems in which *information flow* is strictly controlled.

A capability-based access control model that also provides a suitable authority transfer scheme, such as *take-grant* [6], can be shown to be *safe*. This means that all future access rights that a subject may obtain can be decided by analysing the current system state. Thus, in order to determine the security of a capability-based system (in particular with regard to its access and information flow policies) it is sufficient to analyse its *capability distribution*, i.e., the distribution of capabilities over all the subjects in the system. Such an analysis will take into account all possible transformations of the capability distribution to identify subsystems and the possible information flow between them. Given specific access and information flow requirements, the analysis can be used to determine whether a system successfully fulfils these requirements [2].

In existing capability-based systems the capability distributions are implicitly defined by the code that creates objects and transfers capabilities between subjects at runtime. A security analysis thus requires that the code first be analysed to determine which capabilities exist, how they are initially set up, and how they are subsequently propagated throughout the system. Depending on the code, this could be a complex, and potentially infeasible, process.

We propose that capability-based systems should have an explicit representation of the system's capability distribution. While having such a representation is essential for performing a security analy-

[1] Providing the system does not contain any side channels that bypass the capability-based access control.

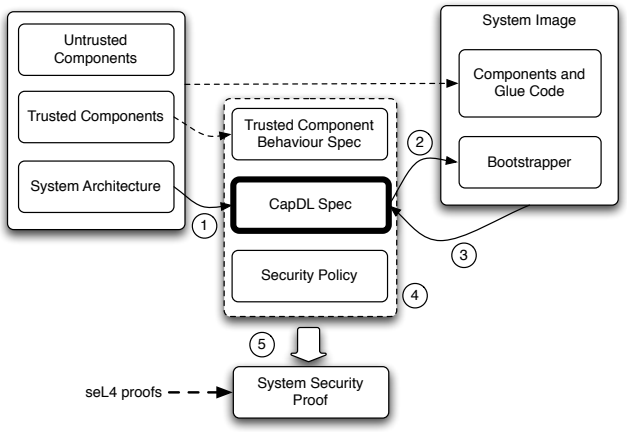

Figure 1: The role of capDL in development, analysis, and verification.

sis or formal security verification of the system, it provides other useful benefits. A clear description of a system's desired capability distribution helps the design and implementation process as well as debugging and documentation. Debugging of the system is facilitated by being able to refer to the expected capability distribution, as well as by having access to the actual capability distribution of the system being debugged. A description of a system's expected and actual capability distribution can be used to produce clearer and more complete system documentation.

In this paper we present *capDL*, a capability distribution language for seL4 [5], a formally verified capability-based microkernel. The purpose of capDL is to describe state snapshots of systems running on seL4.

CapDL plays a central role in our overall vision for development, analysis and verification of trustworthy embedded systems (Figure 1). In this vision, the capDL specification of a system is manually written or generated from a system architecture description (1). The specification can be combined with component code, glue code and a bootstrapping process to produce a runnable system image (2). Alternatively a capDL specification can be dumped from a running system (3). The capDL specification (whether hand-written, generated, or dumped) together with behaviour specifications of the system components serve as input into a security analysis tool that verifies whether the system architecture fulfils the required security policy (4). This verification can be further extended with refinement proofs of the underlying seL4 kernel to prove security properties of the actual system implementation (5).

In the rest of this paper we first present a brief overview of seL4 in Section 2 followed by a description of the capDL language in Section 3 and a comparison of capDL with other approaches to describing capability-based systems in Section 4. CapDL is a work in progress. In Section 5 we present the current status of this work and discuss our plans for its future. Finally we conclude in Section 6.

2. OVERVIEW OF SEL4

The seL4 microkernel is a small operating system kernel designed to be a secure, safe, and reliable foundation for a wide variety of application domains. As a microkernel, it provides a minimal number of services to applications. The kernel services are general enough for composing more complex operating system services that run as applications on the microkernel. In this way, the functionality of the system can be extended without increasing the code and complexity in privileged mode, while still supporting a wide number of services for varied application domains

Kernel services are provided through a small set of kernel implemented objects whose methods can be invoked by applications. These objects can only be accessed and manipulated using tamper-proof capabilities. The operations an application can perform are, therefore, determined by the set of capabilities the application possesses. The capabilities are stored in kernel managed memory and can only be manipulated indirectly through the kernel. Capabilities can be copied, moved, and sent using seL4's inter-process communication (IPC) mechanism. The propagation of capabilities through the system is controlled by a take-grant-based model.

The set of objects implemented by the kernel can be grouped into six categories:

Capability Management Capabilities in seL4 are stored in kernel-protected objects called *CNodes*. A CNode has a fixed number of slots which is determined when the CNode is created. Individual CNodes can be composed into a *CSpace*, a set of linked CNodes. In order to invoke an operation on a capability, that capability must be stored in an application's CSpace.

Object and Memory Management The *Untyped Memory* capability is the foundation of memory allocation and object creation in the seL4 kernel. A kernel object is created by invoking the *retype* method on an Untyped Memory capability. After a successful retype invocation a capability to the new object is placed in the application's CSpace. Untyped capabilities can also be used to reclaim retyped memory with the *revoke* method.

Virtual Address Space Management A virtual address space in seL4 is called a *VSpace*. In a similar way to CSpaces, a VSpace is composed of objects provided by the microkernel. The objects for managing virtual memory are architecture specific. On the Intel IA32 architecture the root of a VSpace consists of a *Page Directory* object, which contains references to *Page Table* objects, which themselves contain references to *Frame* objects representing regions of physical memory.

Thread Management Threads are the unit of application execution in seL4 and are scheduled, blocked, unblocked etc, depending on the application's interaction with other threads. A *TCB* (thread control block) object exists for each thread and provides the access point for controlling the thread. A TCB contains capabilities that define the thread's CSpace and VSpace. Note that multiple threads can share the same CSpace and VSpace or parts thereof.

Inter-process Communication (IPC) *Endpoints* (EP) are used to facilitate inter-process communication between threads. *Synchronous Endpoints* provide rendezvous-style communication, allowing the passing of data or capabilities between applications. When only notification of an event is required (with no need to send message data), then *Asynchronous Endpoints* (AEP) can be used.

Device I/O management Device drivers run as applications outside of the microkernel. To support this, seL4 implements I/O specific objects that provide access to I/O ports, interrupts, and I/O address spaces for DMA-based memory access.

3. THE CAPDL LANGUAGE

The main purpose of capDL is to describe the capability distribution of a system running on top of seL4. The language is intended

to be used in several scenarios and has been designed with these in mind. Initially two separate goals led to the development of the language.

The first goal was to have a representation of the system that was suitable for security analysis, which would involve mapping a capability distribution to a security model and determining whether it fulfils security requirements. The second goal was to enable developers to easily specify the desired capability distribution of their system and provide it as input to a bootstrapping process, which would automatically create required objects and configure and populate the appropriate spaces to reflect the specified structure.

For the first goal what is needed is a format for describing a snapshot of the capability distribution in a system. To be suitable for security analysis the snapshot must describe which objects exist in the system and which capabilities they have access to. For the second goal the specification must be sufficiently detailed to allow automated code generation. Therefore it needs to include all information about capability arguments that such implementations will need. Some of this information will not be relevant for security analysis. For instance, for a security analysis it is necessary to know which frames of physical memory a process in the system can access via its virtual memory, but it is not necessary to know under which virtual address each of these physical addresses is visible to the process. For a concrete implementation of a bootstrapping component on the other hand, this latter information is crucial.

CapDL allows specifications to be underspecified, that is, to omit details of objects or capabilities, or whole objects themselves. A key use for underspecification is in early design and for communication of system designs. For example, a specification may omit objects required for bookkeeping during system initialisation, since these are not necessary to understand the overall system. CapDL also allows abstraction of specification, which involves creating a new specification that contains less details, but has equivalent semantics. For example, to facilitate security analysis we can often abstract a complex CSpace graph into a single CNode containing all relevant capabilities — reducing the complexity of analysing a hierarchy of CNodes, but maintaining the semantics of the original CSpace.

Besides these, the language has several other requirements. For developers writing manual specifications it is important that system specifications are easy to write. Thus the language provides shorthand for parts that are tedious to enter manually. For example, in capDL, large contiguous blocks of untyped memory objects and capabilities can be specified as ranges, rather than separate entries for each individual object and capability. Likewise commonly used objects and capabilities can be given meaningful names so that specifications can act as documentation and reflect intentions as well as structure. On the other hand, for specifications that are generated automatically (for example when dumping the state of the system), shorthand is not appropriate, so the language also has the option of representing a capability distribution in the most straightforward way, without requiring use of more complex shorthand. Of course, specifications differing only in the use of shorthand should be equivalent, and in capDL it is possible to show their equivalence.

A capDL specification has two main sections, the *objects* section, specifying all the objects in the system, and the *capabilities* section, specifying all the capabilities in the system.

The language model reflects the seL4 kernel object model as described in Section 2. All seL4 object types are supported by capDL. Since some types are architecture specific, each capDL specification must include an architecture declaration, which subsequently limits the object types that may be used in it.

Capabilities are typed based on the object type that they reference. A capDL capability includes a reference to the object that it refers to, and, if applicable, capabilities also include a field for the access rights that the capability bestows. Besides access rights, some capability types store extra information relating to how the object is accessed. For example, Endpoint capabilities also store a *badge* that is sent during IPCs performed through that capability. These badges help to identify participants in the communication occurring over the endpoint.

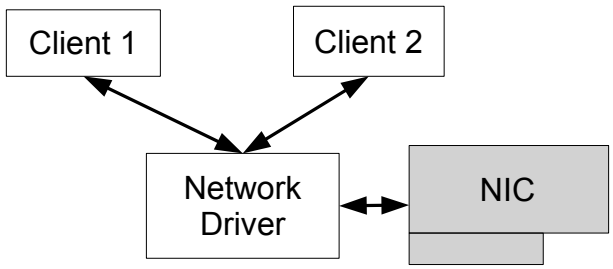

Figure 2: An example seL4-based system.

There are two main classes of objects in capDL: those that are containers, which store capabilities, and those that are not. Container objects include TCBs, CNodes, Page Directories (PD), and Page Tables (PT). These objects provide a mapping from slots to capabilities. By default slots are numbered, but can also be explicitly named to improve clarity. Some containers, such as TCBs, have a fixed size, while others can be created with arbitrary sizes.

Untyped Memory objects are also containers, but are different to the others in that they do not store capabilities, but are conceptual containers for other kernel objects. When a new object is created by retyping an existing Untyped Memory object, the new object is contained in (or *covered* by) that Untyped Memory object. Since Untyped Memory objects can be retyped into smaller Untyped Memory objects, hierarchies of these objects can exist.

Non-container objects are always of a fixed size and include Endpoints and Frames. Note that, while Frames can be of different sizes, these sizes are limited by the architecture, and different sized Frames are conceptually separate object types.

We illustrate the capDL language using a simple example system shown in Figure 2. This system consists of three components: a driver component that has access to a network interface (NIC) device and two client components that communicate with the driver. Each component runs as a separate process and has its own protected address space and individual CSpace.

Figure 3 shows a more detailed view of the objects and capabilities used in the system For clarity we show only the network driver and one client. We see that each component runs a single thread and therefore contains a single TCB. Each component has a single-CNode CSpace and a small virtual address space in which to run. The components have access to endpoints over which they communicate, and they both share a small region of virtual memory which they use to transfer packet data. The network driver component also has access to NIC interrupts through an asynchronous endpoint.

A fragment of the corresponding capDL specification is shown in Figure 4. This specification starts with the *objects* section (line 1), which lists all the objects used in the system. Note that all the objects belonging to a single component are derived from the same Untyped object (lines 3, 12, and 13). This is not required, but makes it easy to destroy and clean up after a process by revoking the capability to the parent Untyped object and making the children inaccessible.

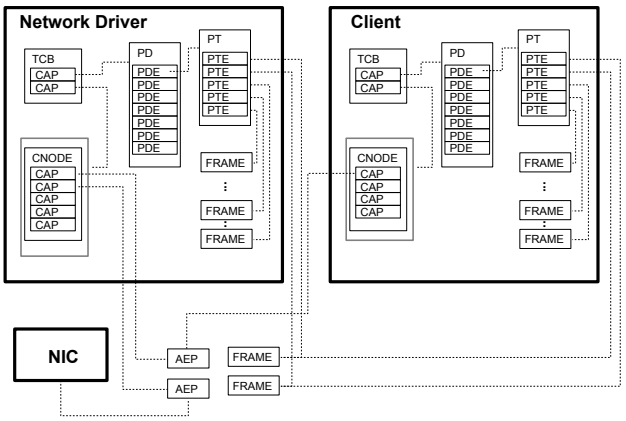

Figure 3: Capabilities involved in the example system.

This is a common pattern in seL4-based systems, and being able to specify it is an important feature of the language.

The capability distribution is described in the *capabilities* section (line 19) where we place capabilities to the appropriate objects in the various containers. We show the network driver's TCB (line 21) and CNode (line 26), and the clients' CNodes (lines 31 and 32). Note that the CNodes contain capabilities to the shared endpoints, but that the capabilities have different badges (lines 27, 31, and 32). This distinguishes the two clients from each other when they communicate with the driver. We also show the structure of the network driver's VSpace (lines 33 and 35) consisting of a Page Directory, a Page Table and various Frames. While not shown here, shared memory is created by mapping the same Frame objects into different VSpaces.

While this example has been kept small in order to keep it simple, it nevertheless highlights some of the key features of the language that fulfil our requirements. As we've mentioned, it allows shorthand for ease of writing and reading (for example, in lines 6 and 7 we specify multiple objects in a single statement, then refer to these objects in lines 37 and 38). It also allows underspecification and abstraction of capability distributions. In lines 3 to 10, for example, we leave out details of Untyped object hierarchies: an implementation of this distribution may actually use a hierarchy of Untyped objects instead of a single Untyped object to create these objects. This is useful both for system description and for analysis.

We extend this example to show how capDL supports abstraction for system analysis. The initial specification shows a detailed process description consisting of a TCB, a CSpace, and a VSpace (lines 21 to 23). Figure 5 represents an abstract version of this process consisting of a single TCB that contains all externally accessible capabilities (the AEPs for communication and interrupts as well as the Frames shared with the client processes). Such a TCB is not a valid seL4 object, however, it may be a valid abstraction of a seL4 process (if refinement can be proved) and can simplify reasoning about such processes. Given that capDL has formally defined semantics, we intend to explore the automatic generation of such abstractions. Note that capDL-based system abstraction and analysis is still a work in progress, and a full discussion of this is, therefore, outside the scope of this paper.

4. RELATED WORK

Existing operating systems with capability-based access control such as KeyKOS [4], EROS [12], and Amoeba [8] do not provide means to explicitly define capability distributions. Capabilities are

```
objects -- The object section starts here                          1
                                                                   2
DRIVER_ut = ut {                                                   3
 DRIVER_tcb = tcb                                                  4
 DRIVER_cspace = cnode (10 bits)                                   5
 DRIVER_code[5] = frame (4k)                                       6
 DRIVER_data[11] = frame (4k)                                      7
 DRIVER_vspace = pd                                                8
 DRIVER_pt = pt                                                    9
}                                                                  10
                                                                   11
CLIENT1_ut = ut {...}                                              12
CLIENT2_ut = ut {...}                                              13
                                                                   14
DRIVER_aep = aep                                                   15
IRQ_aep = aep                                                      16
SHARED_frames[2] = frame (4k)                                      17
                                                                   18
caps -- The capabilities section starts here                       19
                                                                   20
DRIVER_tcb = {                                                     21
  cspace: DRIVER_cspace                                            22
  vspace: DRIVER_vspace                                            23
}                                                                  24
                                                                   25
DRIVER_cspace = {                                                  26
  1: DRIVER_aep (RW, badge: 0)                                     27
  2: IRQ_aep (R)                                                   28
}                                                                  29
                                                                   30
CLIENT1_cspace = { 1: DRIVER_aep (RW, badge: 1) }                  31
CLIENT2_cspace = { 1: DRIVER_aep (RW, badge: 2) }                  32
DRIVER_vspace = { 1: DRIVER_pt }                                   33
                                                                   34
DRIVER_pt = {                                                      35
  0x0: SHARED_frames[] (RW)                                        36
  0x2000: DRIVER_code[] (R)                                        37
  0x7000: DRIVER_data[] (RW)                                       38
}                                                                  39
```

Figure 4: A capDL specification.

```
DRIVER_tcb = {                                                     1
  : DRIVER_aep (RW, badge: 0)                                      2
  : IRQ_aep (R)                                                    3
  : SHARED_frames[] (RW)                                           4
}                                                                  5
```

Figure 5: An abstraction of a capDL specification.

distributed by the system code at runtime and thus distribution remains implicit in the code. Coyotos [10] introduced CapIDL [11] which is a CORBA IDL based language for describing IPC interfaces of the processes in the system. While the CapIDL interfaces are related to, and represent, capabilities, the language does not provide a means for explicitly describing which processes provide and use which interfaces, so the actual capability distribution is never made explicit. Language-based capability systems such as E [7] provide a way to describe capabilities in a programming language (typically as references to objects), however, as with capability-based operating systems the capability distribution is implicit in the code, and is never explicitly presented as with capDL. Higher level architecture description languages such as AADL [3] do provide a means of describing an overall system architecture including the components and their interconnections, however, the abstractions that they operate on are at a much higher level than a capability distribution. Such architecture descriptions could potentially be mapped down to a capDL system description given appropriate mappings between the high-level concepts and the capabilities required to implement them. Various formal models of capability systems exist such as the original take-grant model [6]. Their purpose is a formal security

analysis or specification. They are lacking the necessary detail for system implementation and debugging tools. The advantage of our approach is the direct connection between analysis and implementation within one language. The textual capDL language presented here is one way of representing the underlying model. We have also developed a binary capDL representation that is used as input to our system bootstrapper. While it would be possible to use other languages such as XML to represent capDL specifications, we have not yet investigated these options.

5. STATUS AND FUTURE WORK

The capDL model has formal semantics in the theorem prover Isabelle/HOL and we have implemented a compiler for the capDL language. Besides checking syntactic correctness and consistency of a specification, the compiler also produces a canonical representation of the specification, allowing us to check the equivalence of specifications that use shorthand. The compiler is flexible and includes different backends, including one that produces a binary representation of the specification that can be used as input into a system bootstrapper, and one that produces a graphical representation of the capability distribution in the *dot* format. We also have a debugging tool that produces a capDL dump of a running system's capability distribution. We have implemented an automated bootstrapper that takes as input the binary variant of capDL and produces an appropriate capability distribution in a running system.

A new feature that we plan to introduce extends the language to allow us to specify CSpace manipulation operations (such as creating, moving and destroying capabilities). These operations will have a formal semantics and will be used for analysis of system behaviour. We are also working on integrating capDL into our overall development and security analysis process as described in Section 1. This will include a mapping of capDL to the current security model, as well as automated security analysis tools based on capDL descriptions.

6. CONCLUSIONS

CapDL is a language for explicitly describing the capability distribution of a seL4-based system. It plays a key role in our effort to design, build, analyse, and formally verify trustworthy embedded systems, tying together work being done by system developers and formal methods practitioners. We have designed capDL to closely reflect the seL4 model, and to be flexible enough to allow full and partial specification of a seL4 capability distribution. CapDL has also been designed to be easy to write by system designers as well as to be automatically generated and processed by debugging tools. The underlying capDL model has a formal semantics, which makes it suitable as a key element in our system analysis and verification tool chain. We have developed tools to process capDL specifications, generate running systems from it, dump system state to it, and have started work on developing security analyses based on capDL system descriptions.

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.

7. REFERENCES

[1] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

[2] D. Elkaduwe, G. Klein, and K. Elphinstone. Verified protection model of the seL4 microkernel. In J. Woodcock and N. Shankar, editors, *VSTTE 2008*, volume 5295 of *LNCS*, pages 99–114. Springer, 2008.

[3] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Feb 2006.

[4] N. Hardy. KeyKOS architecture. *ACM Operat. Syst. Rev.*, 19(4):8–25, Oct 1985.

[5] 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 *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[6] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[7] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[8] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Comp. J.*, 29:289–299, 1986.

[9] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63:1278–1308, 1975.

[10] J. Shapiro, M. S. Doerrie, E. Northup, and M. Miller. Towards a verified, general-purpose operating system kernel. In *NICTA workshop on OS verification*, Sydney, Australia, Oct 2004.

[11] J. Shapiro and M. Miller. *CapIDL Language Specification*. Johns Hopkins University, Feb 2006.

[12] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.