Annex: A Middleware for Constructing High-Assurance Software Systems

Tristan Newby[1] Duncan A. Grove[1] Alex P. Murray[1] Chris A. Owen[1]
Jim McCarthy[1] Chris J. North[1]

[1] Defence Science and Technology Organisation
Edinburgh, South Australia 5111,
Email: `tristan.newby@dsto.defence.gov.au`

Abstract

Cross Domain Solutions and Multi-Level Secure systems are becoming more popular as the benefits of merging data from different security levels becomes more widely understood. Software forming the Trusted Computing Base of such systems must maintain isolation between data from differing security levels while providing some means of bridging that isolation under strictly supervised conditions. We cannot expect to be able to build such trustworthy software using contemporary software development tools and techniques.

We describe the Annex Object Capability System, a tiny, security-focused software development framework and middleware for implementing high assurance application software on top of existing highly certified COTS μkernels. By leveraging existing operating system provided process space isolation, we are able to provide the programmer with a simple, familiar environment for building complex, yet truly secure software.

1 Introduction

Historically, the mainstream computer industry has dealt with security problems via the path of least resistance, wallpapering over cracks as they appear. While this has sufficed until now, there is increasing evidence that computer security is in need of structural reform.

One prevailing change is a shift from open world to closed world security policies. For example, most corporate firewalls now block all traffic by default, using overrides to allow white-listed communication. Unfortunately implementing these types of policies on top of standard computer systems is fragile since these systems typically permit any actions as long as they are not explicitly disallowed. Trying to achieve closed world semantics using exclusion based mechanisms is difficult to get right, especially in a highly dynamic environment. Striking a good balance between under-constraint and over-constraint remains a difficult task, but there are mechanisms that can be used to manage this problem.

One approach gaining favour is the use of sandboxing to isolate programs, or even parts of programs, from each other so that a malfunction or compromise in one does not affect any others.

A more powerful extension is to combine a component based methodology with strong isolation and then use *capabilities* for controlled communication between otherwise isolated components. With capabilities, programmers are freed from having to protect against anything that could possibly happen and can instead focus on what a program must do to fulfil its purpose. Rather than needing to prevent operations after the ability to perform them has already been conceded, capability checks guarantee that explicit authorisation has been granted before any action is performed. With capabilities, these checks happen automatically because the authority to perform an action is inextricably bound with the mechanism for performing it.

One area where such trustworthy software is required is in Cross Domain Solutions (CDSs). CDSs can replace replicated computers and networking at multiple security domains with a single set of equipment that is able to process data at multiple sensitivities. At the core of such systems is a Trusted Computing Base (TCB) which must maintain the boundaries between the different domains. It may also provide some approved, assured, auditable path for selected data to cross those security boundaries. The software that is used to build such a TCB needs to be trustworthy and amenable to formal proofs of its security correctness.

In this paper, we describe the design of the Annex Object Capability System, a middleware layer designed to be deployed on COTS secure μkernel based operating systems, utilising their strong guarantees of isolation between components and message passing primitives. We envisage the primary deployment scenario for Annex systems as forming a TCB for larger systems such as CDS, however it is powerful and flexible enough to construct a workstation replacement.

Section 2 of this paper describes the high level Annex Object Model. Section 3 dissects the system's components including the Object Capability Reference Monitor and elements that make up a standard object. A prototype implementation is discussed in Section 4 and Section 5 shows how all this interacts to produce real-world systems and devices. Section 6 examines the system's security properties and performance characteristics while Section 7 places Annex in the context of related work.

2 Annex Object-Capability Model

The Annex Object-Capability Model is an *object-capability system* (Miller & Shapiro 2003), which combines capabilities and object oriented programming. At the most abstract level this equates to conceptual "circle and stick" diagrams like those shown in Figure 1. Nodes represent *objects* and the directed graph of connected edges represent the *capabilities* that objects hold to other objects.

An object-capability system manages the complexity of capabilities by naturally decomposing complex systems into their component objects. Using capabilities to address each of the objects allows the construction of fine-grained security policies expressed through the programming model rather than imposed as an afterthought to system design.

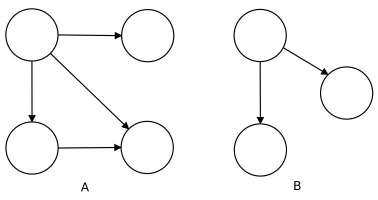


Figure 1: Objects connected by the capabilities that they hold

Objects are self-contained units of state and processing that reside exclusively in a partitioned process with their own address space, unable to directly access any code or data outside of their boundary. They are the fundamental unit of isolation, although they may also hold capabilities allowing them to communicate beyond their border.

Capabilities are unforgeable, authority carrying references that combine the *name* of the object to which they refer as well as the *permissions* required to access it (Dennis & Van Horn 1966), into one atomic entity. They provide a strong mechanism for selectively bridging the isolation between objects.

An object may wield the authority associated with any capability that it holds to *invoke* a method on the object that is designated by the capability. The method parameters may contain call-by-copy data and/or copies of further capabilities. The former provides a mechanism for permitting *information flow* between objects while the latter provides a mechanism for allowing *authority flow*, where capabilities can be copied, or *delegated*, from one object to another. These mechanisms allow programmers to compose object collections that perform useful functions.

The only rule governing capability propagation is that “only connectivity begets connectivity” (Miller 2006). With this simple premise, programmers can manage complexity *and* security at the same time by using good object-oriented decomposition, which naturally enforces the Principle of Least Authority (POLA) (Saltzer 1974). This provides defence-in-depth, as any exploitable bugs in one part of the system are unable to be leveraged to access other parts of the system.

The object-capability model provides a sound framework in which to formally reason about the security properties of the system. As long as the security of the underlying isolation mechanism holds, an isolated object can not *interfere* with the behaviour of any object that it is not ultimately connected to via capabilities. Proving such *non-interference* is a critical step in proving the absence of covert channels in high grade systems.

In particular, object-capability systems provide a strong basis for reasoning about the upper limits on connectedness and hence authority of different parts of the system. For example, the group of objects labelled A in Figure 1 may become fully connected by the objects passing capabilities to each other, however, disjoint graphs A and B can never become connected. This allows us to prove that one part of the system cannot interfere with another part.

While standard capabilities like those described above naturally enforce object-level access controls, the *permissions* list associated with each capability used in Annex provides a convenient and efficient mechanism to control which methods the capability may invoke on the associated object.

Objects may also be able to derive a new capability from an existing one that they have. The newly created capability can only possess an equal or lesser authority to that from which it was derived. This is known as *rights attenuation* and provides a mechanism to scale access control down from the object level to method level checks.

Finer grained assertions can be made by considering the internal behaviour of objects to analyse how information and authority flow through them (Murray 2010). Although this is a much more complicated problem, the hard boundaries between objects enforced by the object-capability model reduces the evaluation state space making such evaluations more tractable than for traditional monolithic programs.

One of the core goals when designing Annex was that calls between objects should be transparently networkable. All method calls look the same to the programmer whether the call is to a local object or one hosted on a remote device. Capabilities must therefore have a global namespace. The use of Mobile IPv6 means that the IPv6 address used in each capability will continue to be valid regardless of where each device is actually located on the network.

Due to the fact that a method call may need to communicate across the globe, all method calls between objects are asynchronous, allowing objects to make a method call and continue to do useful work before pausing to collect the result of that method call. Annex method calls work in a very similar manner to MPI (MPI Forum n.d.).

In order to simplify reasoning about the resultant distributed system, a form of cooperative multitasking has been implemented. Objects explicitly choose when to give up processor time. However, they are unable to claim the results of any method calls without yielding the processor, encouraging them to yield from time to time.

Object programmers need only write the code that implements their object’s functionality. A small, standard wrapper layer of code envelops each object in the system, making the implementation details transparent to the programmer. Consequently much existing code can be turned relatively easily into an Annex object or object collection without having to be rewritten from scratch.

Annex was designed such that it can be layered on top of an existing secure μkernel rather than attempting to build the isolation mechanisms ourselves from scratch. The focus was on developing a useful, intuitive interface for programmers that retained the strong security underpinnings of the μkernel. For example, the Annex programming model explicitly disallows dynamic memory allocation. This removes several classes of bugs from occurring, and ensures that Annex systems extend conformance to the Separation Kernel Protection Profile (SKPP) (National Information Assurance Partnership 2007) into the programmer space.

3 Annex Components

An Annex system is comprised of several components working together to implement the model described in Section 2.

3.1 The Object Capability Reference Monitor

Annex's Object Capability Reference Monitor, OCRM or simply monitor, provides an abstraction of the underlying operating system and acts as a reference monitor (Anderson & Co. 1972) for the Annex system. It is designed to be verifiable, hence it is small, and performs minimal functions. Primarily, it mediates delivery of messages between objects.

The monitor provides restricted versions of the scheduling and interprocess communications mechanisms that are provided by the underlying operating system and is built around the following major data structures and components: the object table, capability, catalogue, clist, message, message queue, promise, requests table and scheduler.

The *object table* lists all objects that exist on a given device. It allows the monitor to keep track of information about the objects including their status and how many method calls are outstanding for each object. When an object is created, its process space is created and initialised, an entry is made in the object table and a master capability is created with permissions for all methods exported by the object. This capability is returned to the object's creator.

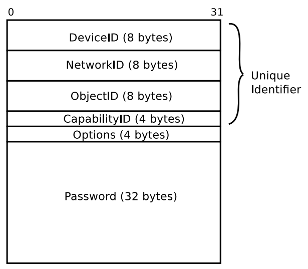


Figure 2: Capability structure

Capabilities are represented within the Annex monitor as shown in Figure 2. The DeviceID and NetworkID taken together form the IPv6 address of the host system. The ObjectID uniquely identifies the specific object on that device. There may be more than one capability to a given object on a particular host, so the CapabilityID is used to uniquely identify each capability to a particular object. The Password field is a large random number that makes forging valid capabilities computationally infeasible.

All capabilities that reference objects on the local device are stored in the monitor's *catalogue*. The catalogue provides services for capability creation, derivation, validation and destruction, and storage of the authority associated with each capability in the form of a permission bit-vector.

Each object accesses capabilities indirectly via an index (or *handle*) into its *clist*, and does not have access to raw capabilities themselves. This prevents the object from directly reading or writing a capability's fields, preventing the object from bypassing the controlled mechanisms for capability presentation or transfer by forging or altering a capability. This is also necessary to prevent violation of MLS policies as detailed by Boebert (Boebert 1984). The indirection involved for capability access using clists is illustrated in Figure 3.

The Annex monitor creates a pseudo-object, which allows objects access to their *clist*. If an object has a capability to its clist, it may be able to manipulate aspects of the list of capabilities that it has. For example, it may be able to change the "scope" (see Section 3.8) of a capability that it holds. Clists are a

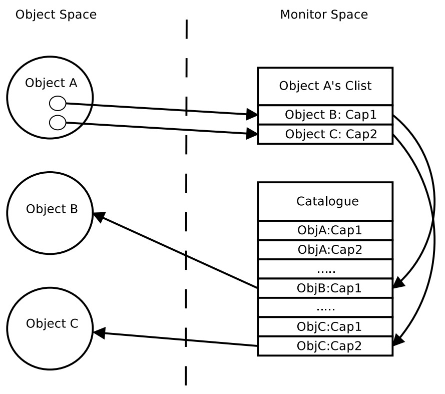


Figure 3: Clist indirection

facet of the catalogue in the context of each individual object.

All communication between objects within the Annex system occurs by message-passing. *Messages* are atomic, meaning that they contain all information required to initiate the method call on the destination object including all parameters required by the call.

All messages between objects are passed via the monitor. The monitor uses a *message queue* to store messages that it has not yet passed on to the destination objects.

As Annex is a distributed system, it provides a means of making calls on objects, then continuing to do useful work as that method call is processed. When the method call is made, the monitor returns a token, known as a *promise* to the calling object. When the caller wishes to claim the result of the method call, it presents the promise back to the monitor, which notifies that caller when the results are able to be claimed.

In order to match method replies with their initiating calls, the monitor keeps track of them in its *requests table.*

Delivery of messages is handled by the *scheduler.*

The Annex monitor is event-driven. Events are either an "interrupt" from a hardware component or a message arriving from an object. The monitor manages message events using a single message queue and deals with each message in turn. The monitor is single threaded, making it easier to ensure the correctness of both design and implementation.

The Annex monitor delivers as many of the messages that it has outstanding in its message queue as it can before accepting any new messages from objects, or processing hardware events ("interrupts"). Although the word "interrupt" is used, these events do not cause the Annex monitor to be interrupted or preempted. They are treated as data sources that have data ready to be processed by some registered object, and as such are scheduled in exactly the same manner as other events.

The object table provides *subjects*, the clist and catalogue together denote the sum total of the *authority* in the system, while the message queue and requests table enable the monitor to track the (flow of) *content* or *data* in the system.

3.2 Annex Objects

Objects are isolated units comprising both code and state. Each object resides in its own *address space* and has no direct access to any other memory, including shared memory or shared libraries. Objects may only communicate with each other by passing messages and must have a capability allowing such communication. Part of every object's address space is reserved for use as a *message buffer* which is used to process both incoming and outgoing messages.

Although an object may make several different methods available to other objects, each object has a single entry point. When a new message is delivered to an object, the entry point function is invoked and it determines how to process the received message based on its content. Having a single entry point simplifies calling conventions.

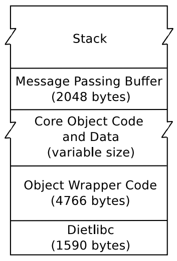

Stack
Message Passing Buffer (2048 bytes)
Core Object Code and Data (variable size)
Object Wrapper Code (4766 bytes)
Dietlibc (1590 bytes)

Figure 4: Memory layout of a standard Annex object

A common object wrapper is provided for every object, abstracting the implementation details from object programmers and providing message-passing, task management and other object administration. Wrapping object code in platform specific management code allows object programmers to specify an object once, and have it run on any platform for which wrapper code exists.

Objects operate on an event loop. They are idle until they receive a message from the monitor. Once a message has arrived for an object, no others can be received until the object has yielded the CPU. This allows objects to use a single message-passing buffer for all calls that they make and receive.

In order to facilitate the required object isolation, objects are statically compiled executables.

3.3 Messages and Message Passing

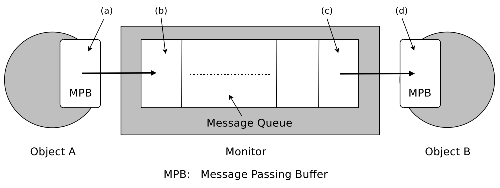


Figure 5: Message Passing in Annex

Figure 5 shows the process of Object A sending a message to Object B. Object A must first construct a correctly formatted message in its message passing buffer (a). It then signals the monitor, which collects the message, placing it in its message queue (b). Before enqueueing the message, the monitor checks that Object A has a valid capability to Object B. Eventually the message progresses to the head of the queue (c). When Object B is able to receive it, the monitor places the message in Object B's message passing buffer (d). Finally, Object B's entry point function is invoked, and it can process the new message.

Object programmers are shielded from the details of message passing. A pre-processor is used on the object definitions at the compilation stage to generate remote procedure call (RPC) code for each method. The RPC code arranges for the marshalling and demarshalling of parameters and aids in providing a simple method calling syntax.

All messages are self-contained. No references other than capabilities can be passed between objects. As there is no shared memory, method calls are performed with call-by-copy semantics. While this does impose an increased burden on system efficiency, the increased security and stability from having no shared pointers outweighs this disadvantage.

Each message that invokes a method call contains: a capability providing the destination address, which the monitor uses to route the message to the appropriate device and object; a method identifier; and any parameters for the method call. The monitor on the device containing the destination object is responsible for checking the validity of the capability, and that the permissions allow invocation of the identified method. This ensures that the caller's authority is limited to that of the specific capability presented.

There are four main message types. Method invocation (CALL) and method response (RETURN) mark the beginning and end respectively of a method call. Processor yield (ENDTURN) is used to stop a method call from processing, and puts the object into an idle state, enabling it to receive the results of an earlier method call that it made. RESUME messages are used to synchronise multiple outstanding method calls on a single object.

A CALL may be *one-way*, in which case the caller does not receive a corresponding RETURN message, or *asynchronous*, in which case the RETURN message is received some time later.

3.4 Tasks and Turn-Based Multitasking

In the Annex system, objects must always be in one of two mutually exclusive states. They are *active* while doing some computation, and *idle* once the computation returns or they are waiting for the result of a method call.

When an object receives a CALL message, a new thread of control, or *task*, is created. This task exists until it sends a RETURN message, at which point the task is destroyed and its stack and other resources are reclaimed.

While a task encompasses one complete method call on an object, a *turn* refers to the time period during which an object is continuously executing. A turn starts with either the invocation of a method or a return from waiting on a promise, and runs until either the method call completes or the object waits on another promise.

At any point in time, more than one task may exist for any particular object, but only one of these may be scheduled in any single turn. Each task explicitly releases control of the CPU by sending the monitor an ENDTURN message (e.g. by waiting on a promise), which also saves the task's context. The context is restored when it is re-invoked after having received a RETURN message corresponding to a CALL message it has issued previously. The only task interleaving points are those involving ENDTURN messages.

Annex makes use of the preemptive multitasking that is a feature of modern operating systems to allow objects to run "in parallel", in addition to the coarse-grained cooperative multitasking described earlier, to avoid many of the consistency problems associated with preemptive multitasking. The Annex scheduler enables this by enforcing the rule that objects may not execute more than one turn at any point in time. This rule means that we can treat an object's turn as a "critical section". Objects can then be assured that their state (e.g. the values of global variables) will not be altered by another task during their turn.

This is achieved without explicit locking or other synchronisation primitives.

As object execution can be modelled as a series of critical sections, program correctness can be much more simply formalised. To avoid multiple outstanding tasks interfering with global variables, these variables can often be moved on to the stack, becoming the equivalent of thread local storage.

Annex's tasks and turn-based multi-tasking removes whole classes of bugs related to standard multitasking and multi-threading. It also allows Annex systems to scale effortlessly along with the number of available processing cores.

3.5 Requests and Promises

Since Annex is a distributed system, it provides a means of making calls on objects on remote devices while continuing to do useful work as that method call is processed. Annex uses the idea of *promises* (Friedman & Wise 1976) to implement this feature. An object receives a promise from the monitor when a method call is made. The result of the method call is claimed by a call to `promiseWait`, providing the earlier received promise as a parameter. As well as `promiseWait`, there are functions that let objects wait on some, or all, of a set of promises.

After the method call is made and a promise received, the object can continue to perform useful work, however, once a call to promiseWait (or equivalent) has been made, no further work is done by that task until the promise has been fulfilled. When the call to promiseWait is issued, the object sends the monitor an ENDTURN message. The object is then able to be scheduled to process another message. This may be either a reply to a previous call that the object made, or a new call, resulting in a new task being created.

The monitor maintains a *requests table* to associate RETURN messages with their earlier CALL messages. When the monitor receives a CALL message from an object, it creates a new requests table entry with a unique request ID and stores information identifying the source object and task. The request ID is then inserted in the CALL message before it is passed on to the destination object.

When that object sends its RETURN message back to the caller object, it includes the request ID. The monitor reads the request ID from the message, and uses that to look up the object and task that this RETURN message is destined for. The message can then be scheduled for delivery.

If the object making the call, and the object being called are on separate devices, then entries are made in the requests tables on each device. The request table entry in the caller's monitor records the calling object and task, and is used to route the response back to the correct object and task, as in the local case. It inserts the request ID into the message, which is then sent to the remote device for processing.

When the message is received by the remote device, it records the existing request ID (Request ID 1), and the IPv6 address that the message came from in its requests table. A new request ID (Request ID 2) is created and inserted into the message before being delivered to the destination object.

When the RETURN message, containing Request ID 2, is delivered to its monitor, Request ID 2 is replaced with the original request ID (Request ID 1) and the message is sent back to the original monitor using the recorded IPv6 address. When the message is received by the original monitor, it is delivered as per the local version described earlier.

In this way it is possible to generate request IDs that traverse a network while remaining unique on both the caller's device and the called object's device.

3.6 Permissions

When objects pass capabilities to other objects, they may not wish to convey the same access rights (permissions) that they have. For example one object may wish to pass a capability that refers to itself, to another object but restrict the second object from calling anything but one particular method on it. This is done by generating a permissions mask, then passing it to the `deriveCap` system method on the capability of which we wish to derive a new version.

The supplied permissions mask is logically ANDed with the permissions of the capability used to make the call so that a derived capability can never gain more authority than the one from which it was derived.

3.7 System Methods

System methods are methods that are executed on behalf of a particular capability but are actually run by the monitor rather than the target object. An example is `destroyCap`, which removes that capability from the monitor's catalogue of capabilities. System methods execute on the device that hosts the object that the capability refers to. All capabilities have a set of standard system methods attached to them, however access to these methods may be removed by permission restrictions.

3.8 Capabilities - Local vs Global Scoping

Capabilities held by objects exist in one of two mutually exclusive states, *locally scoped* or *globally scoped.* They are locally scoped by default and go "out of scope" at the end of the task that they are currently being used in and are removed from the object's clist at the completion of that task. Once a capability has been removed from an object's clist it may not be used in any further method calls.

For an object to retain a capability beyond the scope of the task to which it was passed, it must explicitly set the capability to be *globally scoped.* Globally scoped capabilities remain in the object's clist beyond the end of the task that they were passed to and can be used by other tasks on that object, including those that they were not explicitly passed to. Objects may release globally scoped capabilities by setting them to be locally scoped. When that task completes they are removed from the object's clist by the monitor.

As `makeGlobalCap` and `makeLocalCap` are methods that run on an object's clist, an object must have a capability to its clist (with appropriate permissions) in order to call these methods. This means that we can construct objects that are unable to retain any of the capabilities that are passed to them. The ability to ensure that an object only ever has locally scoped capabilities (by not giving it a capability to its clist) is a confinement mechanism that assists in constructing systems based on the principle of least authority.

3.9 Error Handling

When a method call fails for whatever reason, provision is made for communicating details of the failure back to the calling object. When an error condition occurs inside an object method, the object's author should return from the call using the

`RETURN(`*ErrorCode*`)` statement. This returns the *ErrorCode* information and some details about the failure to the caller, including the name of the object, the method ID, and the source code line that the error was generated at. The caller can access this information by calling `annex_perror(`*pPromise*`)`. The *pPromise* parameter is the promise that was supplied to the original call that failed.

4 Prototype Implementation

In order to rapidly design and develop prototype Annex systems, we have initially used a hand-rolled implementation of *ttylinux* as the underlying operating system. Although some elements of the system have had to be built differently to how they would if running on a secure μkernel, almost all of the programmer-specified object code (exceptions being for hardware I/O and some code to handle object creation and message passing) will be able to run unaltered on top of other operating systems. Using Linux has also enabled us to use the same code-base to compile a system for use on an embedded ARM chip, and on standard x86(_64)-based PCs.

We modelled the monitor and each object as separate processes and used *dietlibc* (dietlibc 2014) to statically compile each executable. This ensures that there are no shared libraries and all code for an object exists within its own process space.

In addition to having no shared code, we have been mindful of the fact that particular objects will need to be small in terms of code size in order to make formal verification a tractable proposition.

Permissions bit-vectors are 128 bits in size, with the first 16 bits reserved for system methods.

To implement the message passing, sockets were used. This does impose some overhead on the system, however, as we envisage Annex being used in niche situations (such as implementing a TCB in a CDS), the isolation, and hence security benefits, far outweigh the modest performance impact. In addition, there are mechanisms for mapping regions of memory between processes on secure μkernels that may prove to be more efficient.

One of the benefits of having a robust simulator for development effort is that standard debugging tools such as gdb are able to be used. We also utilised *replay* (Murray & Grove 2013), a tool for visualising graph modification over time. An Annex plugin allowed us to view the timing and content of messages.

5 Example Object systems

Annex has been designed so that software systems built using it are highly amenable to security proofs, and the example systems that have been developed demonstrate that benefit. The first example discussed is a software-only data diode which shows the simple power of the system. Then a much more complex software system, a graphical user interface, is described. Finally, an example hardware and software design is discussed as well as several real world devices that have been built using this design, demonstrating some new opportunities that become available in the high-assurance solution space due to our system design.

5.1 Data Diode

The concept of a unidirectional network link or data diode has been around for some time, but it was not until the 1990's that a hardware implementation of a data diode became a reality. The Starlight Interactive Link Data Diode (Anderson et al. 1996) was certified to EAL7 under the Common Criteria in 2005.

Although the concepts are well understood, no software-only data diodes have been able to achieve such a high level of certification. This is due in large part to the complexity of software-based systems.

We contend that Annex, running on a secure μkernel, provides a framework that is highly amenable to both formally evaluating the implementation of the diode itself, and the means of passing data through it.

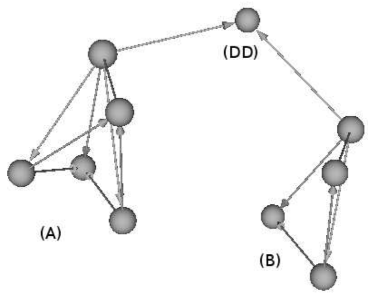


Figure 6: A collection of objects using the diode (DD) to securely communicate in one direction

Listing 1 shows the source code for a simplified version of a data diode that is restricted to passing 32-bit values. The diode sits between two disjoint graphs of objects and only allows data to pass in one direction between them.

```
uint32_t local_data;

EXPORT write_up (IN uint32_t data)
{
  local_data = data;
  RETURN(OK);
}

EXPORT read_down (OUT uint32_t data)
{
  data = local_data;
  RETURN(OK);
}
```

Listing 1: diode.def

The diode is specified in a few simple lines of code. As long as the assumed object isolation properties are satisfied, and the object wrapper and monitor code shown to be secure, then it is a fairly simple matter to prove that this object cannot move data from the reader side to the writer side.

An object that wanted to write data to the diode would need a capability to the diode that contained the permission to call the *write_up* method.

`diode.write_up(SYNC, diode.cap, data);`

A diode object (a structure that contains pointers to the objects MPI stub) is associated with a capability (`diode.cap`). The `SYNC` keyword specifies that the calling object will immediately wait for the results rather than use a promise to claim them later.

Equivalently, an object that wanted to read from the diode would need a capability to the diode that contained the permission to the *read_down* method.

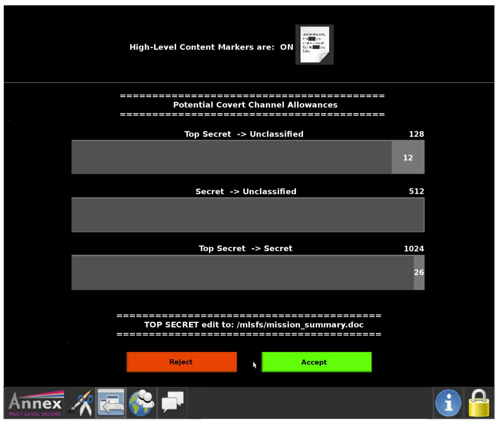
5.2 GUI

The development of a Graphical User Interface (GUI) subsystem demonstrates the power and flexibility of our object capability system. Each element to be drawn to the screen is represented by an object called a *guiUnit*. This object type has attributes such as a background colour, border width and height, an image and/or text component, and placement. GuiUnits are connected together to form a tree, and each may only draw within the confines of its parent guiUnit.

Rendering to the screen is performed by a GUI server. This server queries the guiUnits to determine how to draw them. The server also reads mouse, keyboard and touch input and directs it to the appropriate guiUnit for processing.

While the performance of our Annex GUI subsystem is not comparable to modern high-performance interfaces, it has proven to be more than capable for the applications that we have used it for, including multi-level messaging, network management, cross domain cut and paste, covert channel monitoring tool and blue force tracking.

The covert channel monitoring tool is shown in Figure 7. Icons that launch other applications can be seen on the taskbar at the bottom of the screen.


High-Level Content Markers are: ON

```
==========================================
Potential Covert Channel Allowances
==========================================
```

Top Secret -> Unclassified 128 12

Secret -> Unclassified 512

Top Secret -> Secret 1024 26

```
==========================================
TOP SECRET edit to: /mlsfs/mission_summary.doc
==========================================
```

Figure 7: Covert Channel Monitoring Tool screen rendered by the Annex GUI.

5.3 Cross Domain Solutions

Annex has been used as the software framework for several prototype real world devices. These devices share a common basic architecture (shown in Figure 8) but address different aspects of cross domain security.

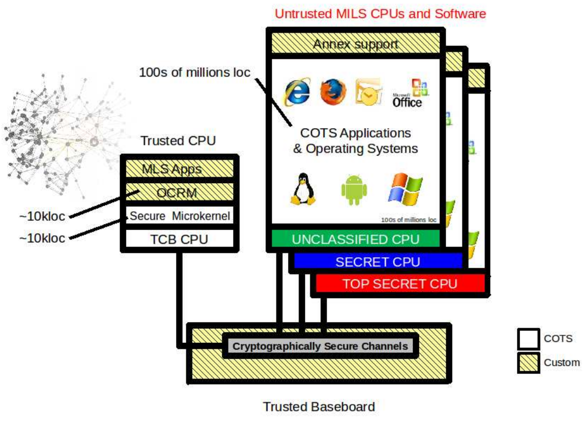


Figure 8: Annex Cross Domain Architecture.

The standard architecture that we use consists of a simple custom baseboard. On top of the baseboard sit 4 COTS single level computing platforms, each comprising a systemboard, CPU, memory and storage. These are entire computers that operate in an isolated domain atop the baseboard.

The baseboard includes hardware encryption modules adjacent to each at-level connector so that all communication across it is encrypted. Each level sees the baseboard connection as a network adapter and can read from and write to it as per any network adapter. For our purposes, each of these operate at a given security level, nominally Unclassified, Secret and Top Secret. The fourth board acts as a TCB, running our Annex Object Capability System and it mediates access to peripherals and controls information flow between the other boards.

The baseboard includes a dynamically developed list of allowed communications channels, both internal and external to the device, meaning that Annex can ensure that each at-level domain only ever communicates with other devices at their equivalent level, i.e., Secret can only ever interact with another Secret domain. While all communication between devices takes place (encrypted using DH-STS AES256) over the Unclassified link, this is simply a design choice, it could just as easily take place over a Top Secret network.

5.3.1 Annex Router

The Annex Router is conceptually similar to a VPN concentrator. In addition to the basic architecture described above it includes a second network interface at each security level, allowing access to an entire network at each classification. In this case Annex software is used only to create, remove and audit network connections. It should be noted that Annex software has no access to the network data itself, only the metadata used to route the encrypted packets.

5.3.2 Minisec

Two versions of the Minisec device were produced according to this architecture (a first prototype used a different architecture to achieve a different aim).

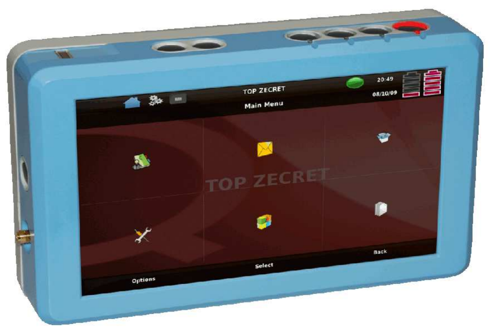


Figure 9: Minisec2.

The Minisec2 is a touch screen device, pitched somewhere between a tablet device and a smart phone. It uses Qtopia as an operating environment on each of the at-level domains, providing services such as VoIP and email at each classification over encrypted channels. The device incorporates trusted buttons that allow the user to select which domain has access to the screen, touch input and audio output at any given time. The Minisec2 was designed for tactical military use, with two large hot-swappable batteries, making the device available without downtime for charging.

The Minisec3 moved the target market to a desktop/server space with a desktop PC form factor. Each domain was upgraded from its original Atom-based board to a Core i7 mini-ATX form factor. This allowed us to run unmodified Windows 7 or Ubuntu operating systems on each domain. The Minisec3 has buttons to allow the user to specify which is the active domain at any given time, and includes a greatly enhanced Annex domain which includes the GUI subsystem.

This form factor, and the strong isolation between the objects running in the TCB, lends itself to use with COTS user applications across different security levels. For example, Microsoft Word has been shown to run in separate domains while securely sharing a multi-level document using such a TCB to filter data to and from each domain (Owen et al. 2011).

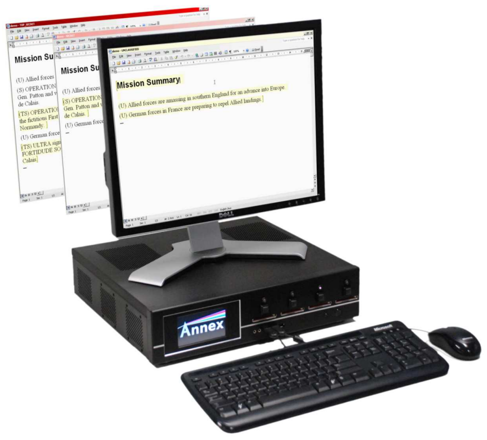


Figure 10: Minisec3 demonstrating the concept of a single document being edited at multiple levels.

Another example of newly developed capability that this architecture allowed was the concept of cut and paste data downgrade. As the Annex-based TCB has complete control of how data is moved between domains, it is able to selectively, and with appropriate user review, downgrade textual information from high to low.

6 Evaluation

Rather than reimplement the strong isolation properties provided by secure μkernels, we have explicitly chosen to leverage them to provide a programming environment that lends itself to more complex high assurance systems. Creating a generic framework in the monitor and object wrapper allows system programmers to focus on how their objects interact without needing to be overly concerned with low level details.

One measure that lends itself to system security analysis is the size of the code base. SeL4 consists of 8700 lines of C code and 600 lines of assembler (Klein et al. 2009). Using sloccount, the Annex monitor consists of 8616 lines of C code, while the object wrapper has 1560 lines of C code and 2 lines of assembler, and if we were to optimise the Annex code for use in a particular scenario, we would be able to cut down these numbers. While the count of lines of code does not by itself infer any security properties, it is indicative of a system designed with security in mind.

6.1 Formal Analysis

An initial formal analysis of Annex, based in the HOL theories of the Isabelle proof assistant(Nipkow et al. 2002), was commissioned. The motivation was to capture a formal model from an advanced prototype system, and to use it as a touchstone for discussions between modellers and developers in a future high assurance development effort. As such it helped initiate an artifact trail that could be used to produce the deliverables required in a high evaluation process.

The ensuing report (McCarthy 2013) quickly introduces a simple model for the distributed communication context in which Annex sits, and the capability, promise and message structures that control and utilise it. The report then concentrates on the two major elements of Annex, the monitor and the object wrapper code, whose properties must be fully analysed before any resultant systems could be formally verified.

For each element, the relevant data structures and methods thereon are modelled in detail, whilst the behaviour (in particular, message passing) is encoded in a state machine (SM) model using these structures in its state description.

Thus, for the monitor, all of the major components - the object table, capability, catalogue, clist, message queue and requests table - are SM state in this sense, and the scheduler is simply represented in the SM transition logic.

Similarly, for Annex objects, the task table, promise table, results table are SM state, and the turn-based multitasking is a consequence of the SM transition logic. To allow for a generic treatment (as opposed to the specific analysis of a particular application) the object methods are fed in as a collection of generic functions on the object data and an abstract computing stack.

Strong isolation allows graph theory to be used to provide inter-object authority bounds, while careful composition of permissions allows intra-object information flows to be verified. In this manner we hope to divide and conquer the evaluation problem.

6.2 Performance

We state quite clearly that Annex is not designed to develop a fully-fledged operating system that can compete with Windows, Ubuntu or OSX for interactive user experience. However, there are an increasing number of niche uses for which the performance tradeoff is a small price to pay for a much more robust security model.

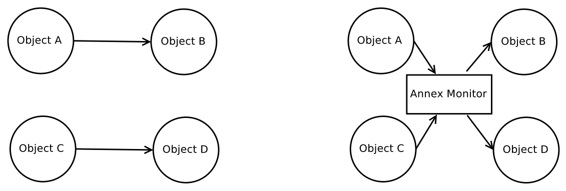


Figure 11: Conceptual, (left), vs actual, (right), message passing in the Annex system

The hub-and-spoke structure of the Annex architecture (see Figure 11) results in some limitations. The fact that all messages pass through the monitor, together with the call-by-copy semantics of method calls, means that there are bottlenecks in the system. We focus strongly on the side of security in the security-performance tradeoff.

Using processes to host objects and sockets to move messages between objects and the monitor means that each method call, from invocation to reply requires at least 6 context switches, however this impost will be lessened if we are able to deploy on a multi-core chip. Part of Annex's design is that the number of objects that are able to run concurrently scales linearly with the number of available processing cores. Previous work (Newby 2008) looked at how to port Annex to multi-core chips such as the Cell processor or Intel's Single-Chip Cloud Computer.

The data copying overhead required to move messages between monitor and objects may be able to be removed if we can use page-transfer mechanisms provided by the underlying μkernel, while thoughtful architecting of systems can bypass much of this performance impost.

7 Related Work

The concept of isolation, bridged only through well defined interfaces, lies at the core of computer security (Aiken et al. 2006). As early as 1981, Rushby (Rushby 1981) identified the need for isolated processes in the development of multilevel secure systems on commodity computing platforms. Recently, there has been renewed interest in applying this principle to computing systems at various levels of abstraction.

In the software sandboxing space, Capsicum (Watson et al. 2010) is a capability system used by the Chromium browser on BSD-based systems to isolate each tab or web session. This stops problems with rendering one tab from affecting others. Similarly, uPro (Niu & Tan 2012) enables programmers to break their program into sections and define interfaces between them which are then checked by a run-time harness.

Annex owes its heritage to the E language (Miller 2006). In E, as in Annex, objects are referenced by capabilities, however, in E some objects may run together in a shared space called a vat. While method calls between vats use a distributed calling convention, calls internal to a vat do not. In contrast, in Annex all calls between objects use distributed calling conventions. In addition, E does not use permissions.

While these efforts are improvements over standard programs using shared memory, none are appropriate for high assurance systems.

The benefits of being able to access information of differing security classifications and the financial benefits of being able to remove multiple workstations from desktops is leading to renewed interest in Cross Domain Solutions. Policy makers are reducing the security requirements of such systems (at least for lower classifications of data) and commercial vendors are beginning to bring systems to market that are able to meet these reduced standards.

Galois' Trusted Services Engine (Galois Inc. n.d.) connects to up to 4 networks at differing security levels and allows users to read from lower levels but prevents write-up. Raytheon's High Speed Guard (Raytheon 2014) and Small Format Guard (Raytheon Trusted Computer Solutions 2014) are bidirectional filtering routers with rules that can be set prior to deployment. All of these systems use SELinux (National Security Agency n.d.) to control data flow, with Raytheon hosting theirs on Red Hat Enterprise Linux (Red Hat n.d.) (RHEL).

Thales' Trusted Security Filter (Thales n.d.) is also a bi-directional filtering router with a predefined, non-configurable filter. The internal architecture of the device is unspecified, however an updated assurance evaluation (SERTIT n.d.) shows that the filter is specified in software.

Thales and Raytheon have also formed a partnership to provide the Australian Department of Defence's Next Generation Desktop (NGD), a version of Raytheon's Trusted Thin Client (Raytheon Trusted Computer Solutions n.d.). This product consists of a Distribution Console (DC) which acts as a VPN concentrator, sending out a single (low) level data stream and pre-configured virtualisation software that allows a client to connect to the servers. Keys appear to be pre-shared. The DC runs SELinux on RHEL to maintain data separation.

SecureView (AFRL 2014), a collaboration between the US Air Force Research Labs, Intel and Citrix, is a CDS that uses XenClientXT as a bare metal hypervisor to isolate multiple virtual computers of differing classification.

The CDS's described above are architecturally similar to the Annex hardware platform described in Section 5.3. However, where they use a single shared CPU and either complex virtualisation software or entire COTS operating systems to manage isolation, we use simple replicated hardware.

In 2007 the Separation Kernel Protection Profile (SKPP) was formalised by the Information Assurance Directorate within the NSA. In October 2008, Green Hills Integrity-178B (Green Hills Software Inc 2010) (a fixed variant of their real time operating system) was certified compliant against the SKPP. This compliance gives Green Hills the only operating system to be certified to Common Criteria EAL6 augmented.

NICTA's seL4 (NICTA n.d.) is a formally verified μkernel. Its security properties, such as functional correctness, have been captured in an abstract specification and, through refinement proofs, a correspondence was shown with its underlying C implementation. It uses capabilities to partition and reference memory.

As both Integrity-178B and seL4 are μkernels they provide a bare minimum of primitives for a user-space program. By keeping this set small, they are able to guarantee that they behave correctly, but their programming interfaces are more difficult to work with than standard computer systems.

We see Annex as extending the trust that secure μkernels such as seL4 and Integrity provide, into a usable space for programmers. The guaranteed isolation for executing processes makes them perfect for hosting Annex object systems.

8 Conclusion

Designed to run on top of existing highly certified COTS μkernels, the Annex Object Capability System provides a strong platform for driving high level evaluation and certification beyond a separation kernel and into user applications.

While multi-level systems are gaining favour again, solutions to constructing the cross-domain component are often focused on supporting fine-grained security policy at the expense of a TCB that is too complex to effectively verify. Annex aims to solve this problem, combining true verifiable security with a familiar programming environment.

An approach to engineering high-assurance trustworthy systems has been shown for exemplar Cross Domain Solutions and Multi-Level Secure systems.

References

AFRL (2014), 'SecureView'. http://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/secureview-government-industry-collaboration-deliversimprobved-levels-of-security-performance-and-cost-saving-for-mission-critical-applications.pdf.

Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G. & Larus, J. (2006), Deconstructing process isolation, *in* 'Proceedings of the 2006 Workshop on Memory System Performance and Correctness', MSPC '06, ACM, New York, NY, USA, pp. 1–10.

Anderson, J. & Co. (1972), 'Computer security technology planning study'. http://www.csrc.nist.gov/publications/history/ande72.pdf.

Anderson, M. S., North, C. J., Griffin, J. E., Milner, R. B., Yesberg, J. D. & Yiu, K. K.-H. (1996), Starlight: Interactive link., *in* 'ACSAC', IEEE Computer Society, pp. 55–63.

Boebert, W. (1984), On the Inability of an Unmodified Capability Machine to Enforce the *-Property, *in* '7th DOD/NBS Computer Security Conference'.

Dennis, J. B. & Van Horn, E. C. (1966), 'Programming semantics for multiprogrammed computations', *Communications of the ACM* **9**, 143–154.

dietlibc (2014), 'diet libc - a libc optimized for size'. http://www.fefe.de/dietlibc.

Friedman, D. & Wise, D. (1976), The impact of applicative programming on multiprocessing, *in* 'International Conference on Parallel Processing', pp. 263–272.

Galois Inc. (n.d.), 'Trusted Services Engine (TSE)'. http://corp.galois.com/trusted-services-engine.

Green Hills Software Inc (2010). http://www.ghs.com.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. & Winwood, S. (2009), sel4: Formal verification of an os kernel, *in* 'Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles', ACM.

McCarthy, J. (2013), Modelling the annex object capability reference monitor, Technical Report DSTO-GD-0751, Defence Science and Technology Organisation, Information Networks Division.

Miller, M. S. (2006), Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA. http://www.cypherpunks.to/erights/talks/thesis/submitted/markm-thesis.pdf.

Miller, M. S. & Shapiro, J. S. (2003), Paradigm regained: Abstraction mechanisms for access control, *in* '8th Asian Computing Science Conference (ASIAN03)', pp. 224 – 242.

MPI Forum (n.d.). http://www.mpi-forum.org.

Murray, A. & Grove, D. (2013), Replay: Visualising the structure and behaviour of interconnected systems, *in* 'Proceedings of the Thirty-Sixth Australasian Computer Science Conference - Volume 135', ACSC '13.

Murray, T. (2010), Analysing the Security Properties of Object-Capability Patterns, D.Phil. thesis, University of Oxford.

National Information Assurance Partnership (2007), 'U.S. Government Protection Profile for Separation Kernels In Environments Requiring High Robustness'. https://www.niap-ccevs.org/pp/pp_skpp_hr_v1.03.pdf.

National Security Agency (n.d.), 'Security Enhanced Linux'. http://www.nsa.gov/research/Selinux/.

Newby, T. (2008), An Evaluation of the Cell Processor for an Implementation of the Annex System, Masters thesis, University of Adelaide.

NICTA (n.d.). http://ertos.org/research/sel4.

Nipkow, T., Wenzel, M. & Paulson, L. C. (2002), *Isabelle/HOL: A Proof Assistant for Higher-order Logic*, Springer-Verlag, Berlin, Heidelberg.

Niu, B. & Tan, G. (2012), Enforcing user-space privilege separation with declarative architectures, *in* 'Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing', STC '12, ACM, New York, NY, USA, pp. 9–20.

Owen, C. A., Grove, D. A., Newby, T., Murray, A., North, C. J. & Pope, M. (2011), PRISM: Program Replication and Integration for Seamless MILS, *in* 'IEEE Symposium on Security and Privacy'11', pp. 281–296.

Raytheon (2014), 'High-Speed Guard'. http://www.raytheon.com/capabilities/products/cybersecurity/highspeedguard/.

Raytheon Trusted Computer Solutions (2014), 'Small Format Guard'. http://www.trustedcs.com/products/SmallFormatGuard.html.

Raytheon Trusted Computer Solutions (n.d.), 'Trusted Thin Client'. http://www.trustedcs.com/products/TrustedThinClient.html.

Red Hat (n.d.), 'Red Hat Enterprise Linux'. http://www.redhat.com/products/enterprise-linux/.

Rushby, J. (1981), The design and verification of secure systems, *in* 'Eighth ACM Symposium on Operating System Principles', pp. 12–21. (ACM *Operating Systems Review*, Vol. 15, No. 5).

Saltzer, J. H. (1974), 'Protection and the control of information sharing in multics', *Commun. ACM* **17**(7), 388–402.
URL: *http://doi.acm.org/10.1145/361011.361067*

SERTIT (n.d.), 'Certified Products, TSF101'. http//sertit.no/product/15.

Thales (n.d.), 'Trusted Security Filter TSF101'. https://www.thalesgroup.com/en/content/trusted-security-filter-tsf101.

Watson, R. N. M., Anderson, J., Laurie, B. & Kennaway, K. (2010), 'Introducing Capsicum: Practical capabilities for UNIX', *;login: the USENIX Association newsletter* **35**(6).
URL: *https://www.usenix.org/publications/login/december-2010-volume-35-number-6/introducing-capsicum-practical-capabilities-unix*