2 Innovative Claims and Application Context

2.1 Innovative Claims

We propose to design and evaluate a computer system architecture, called TIARA, combining novel hardware features, a new form of operating system, and unique application middleware.

The TIARA Architecture is motivated by the needs of the National Intelligence Community; the TIARA hardware will provide for fine-grained tracking of the *security context* (e.g. classification level and compartment) of data, non-bypassable enforcement of a broad range of security policies, while simultaneously building a *dependency network* tracking the provenance of all application data, showing how each application object is influenced by the values of raw sensor data, human inputs, and computational processes.

The TIARA hardware, operating system and middleware will cooperatively build a network of dependency relationships that explains:

- What is the security context of each datum.
- How every application datum was derived.
- What raw data was involved in the computation (whether derived from sensor systems or "human intelligence") .
- What computations were performed.
- Which human users contributed to the interpretation of the data.
- What data has been exposed to which human users

This will allow TIARA applications to:

- Explain any conclusion they have drawn.
- Estimate the reliability of any conclusion based on estimates of the trustworthiness of the sensor data, computational processes and humans that contributed it.
- Update these estimates as estimates of the trustworthiness of the inputs change.
- Present alternative views of an application document that are suitable to audiences with varying clearance levels.
- Identify insiders who may have compromised system resources.

In the rest of this section we summarize the key novel ideas in the TIARA architecture:

Tagged Memory: TIARA's power originates in its novel hardware structure which we call a *Security Tagged Architecture* (STA). In TIARA, all words in memory or registers consist of a data field and a *tag field*; the tag encodes both the type of the data and its security context. The TIARA hardware will include a *Tag Processing Unit* (TPU) that operates in parallel with its main data-paths as shown in figure 1. While the main data-paths combine the data fields of two registers in order to compute a result, the tag processing unit checks the tag fields of the same registers, checking that the operation is legitimate (trapping if the operation isn't permitted) and computes a new tag reflecting the type and security context of the result. The TIARA hardware will be very flexible in its encoding of data types and security context, supporting a broad range of access control and information flow models.

Process Credentials: Each TIARA process has an associated *Principal* representing the current privileges of the user; this is held in the *Principal Register* of the processor. The tags processing unit checks not only the tags of the operands, but also the contents of the principal register. If the process does not have the appropriate privilege to perform the selected operation on the input operands, then the process is trapped into a security violation handler. This allows TIARA to enforce standard security properties (e.g. the ***-property**[2]) at the granularity of the individual word in memory.

Security Context Tracking with Word Level Granularity: The security context part of the tag may be used to encode any of a variety of "information flow" formalisms [26], (e.g. one involving multiple levels and compartments) that can be viewed as forming a lattice [33, 11]. Whenever the processor combines the contents of two registers, the result is assigned a tag that is the least upper bound of the tags of the operands. The resultant data and tag are combined to form a new word. Tags of existing data may only be explicitly modified by a special set of instructions and these are accessible only to specific components of the operating system (e.g. the *Domain Manager*, the component that assigns credentials to a process).

Object-Structured Memory: The TIARA hardware establishes a structured view of memory in which all data consists of objects with definite location, bounds and type. The data-type component of the tag encodes all of the primitive data-types supported by the hardware. Thus, immediate data (e.g. small integers), instructions and references to compound objects in memory are all distinguishable from one another.

Object references include both base and bounds and mediate all memory accesses. The TIARA hardware bounds-checks all accesses to memory, trapping any attempt to access outside the range of the referenced object. Thus, buffer overflows and other violations of memory structuring conventions are impossible (and even if they could occur, the payload would carry the wrong tags to be interpreted as instructions) and as a result it is extremely difficult to hijack a Tiara system using standard types of attacks.

Zero-sized Kernel: The object-oriented abstraction established by the TIARA hardware allows the TIARA operating system to be structured in a totally novel fashion. Rather than having a kernel possessing all privileges, as is done in all COTS systems, or "rings" of protection with increasing levels of privilege as was done in Multics [41, 34], TIARA instead wholeheartedly embraces modularity and the principle of least privilege [39, 12].

Each conceptual component (e.g. scheduler, device driver) of the TIARA operating system is implemented as an individual object; privileges are extended to principals on an object-by-object and operation by operation basis. The scheduler might, for example, have the privilege of loading the "principal register" in the hardware (obviously, a very privileged operation) but have no other unusual privilege such as the ability to change the principal of a process or the ability to examine or change data in an application object. Similarly, a debugger might have enough privileges to examine the stack of a faulting process, but be unable to see or modify application objects referenced from that stack. Since there is no collection of code in TIARA that possesses unlimited privilege, we refer to it as having a "zero-sized kernel".

In this regard, TIARA is similar to capability architectures [32, 27, 19, 42]. However, TIARA's object references do not encode privileges; these are represented separately in the Principal Register associated with each process.

Unified Computational Model: The object oriented abstraction facilitates a novel and elegant computational model. All data, whether persistent or volatile, are regarded as objects; There is only a single conceptual operation: the application of a function to a set of objects (i.e. a function call). All functions are fully polymorphic and follow the "generic function" model of CLOS [20]: A function call is implemented by dispatching on the data-types of all the arguments to select an appropriate method. TIARA provides hardware to accelerate method dispatch.

Pervasive Access Control: The "principal" of the running process is regarded as an implicit argument in all function calls, allowing access control to be directly implemented by the function calling mechanism: The call is processed if the "principal" possesses the privilege to execute the function on the operands; otherwise the call is aborted. In this regard, TIARA can be seen as a natural vehicle for implementing Role-Based Access Controls (RBAC) [16, 40] and particularly for object-oriented views of RBAC [1].

Certain functions map directly to hardware capabilities (e.g. +); in these cases TIARA transforms the function call into a hardware instruction and enforces the access control checks using its tags processing unit.

Dependency Tracking: The TIARA hardware tracks the security context of the result of all function calls, allowing the upper levels of the software systems to build dependency records tracking how all persistent data was computed, what users contributed and what raw data (sensor or human) were involved.

2.2 Application Context

We will demonstrate how these facilities can be used to build a collaborative intelligence processing substrate and we will use this demonstration to evaluate the effectiveness of the TIARA architecture for providing non-bypassable access control and accountable information flow.

The TIARA platform enables the construction of a new form of intelligence document that we term an *Active Briefing Book*; this is a document that provides for viewing at multiple levels of abstraction, for interactive drill-down, and (most importantly) for the tracing of every conclusion of the document to the computations, people and raw data from which it was derived. The elements of an Active Briefing Book are not just text or images (although they can certainly be presented as such) but rather active objects embodying the chain of reasoning that leads to the document's conclusions as well as to opposing interpretations. Elements of the document are constructed collaboratively: raw data from sensors and human sources are routed to NIC analysts with relevant areas of interest and responsibility and with levels of clearance allowing them to view the data. The analysts invoke a variety of tools to process the data and make personal judgments about the significance and impact of the processed data. These are captured as new document elements by the application substrate and annotated with new dependency information linking the interpretations to the source data, the computational processes and the analysts that contributed to the interpreted data. This interpreted data is then routed to other analysts for further processing and interpretation.

Because every conclusion of such a document is linked to its underlying raw data, computations, and human interpretations, any element of the document may be revoked if the source, reasoning process or computational processes involved are found to be incorrect, untrustworthy or corrupted.

2.3 Summary

The TIARA architecture maintains security tags and dependency chains not just for application data, but also for all of its own resources: Changes to system or application software are tracked and linked to their sources as well as to the individuals and computational processes involved. Furthermore, such changes are strictly constrained by pervasive access and flow control policies.

This has several beneficial consequences:

- A TIARA system is highly secure; the normal means of subverting computer systems (e.g. writing beyond an object's bounds, overwriting stack contents by violating stack abstraction...) are not available to an attacker.
- Even if the computer system is successfully compromised, the consequences of that compromise are traceable and revocable.
- Insider attacks are discouraged; even if an insider possesses the means and motives to subvert the system, he will be unlikely to do so if he know that his actions are likely to be traced and that he is likely to be held accountable.
- Declassification and downgrading of documents is transformed from a human-intensive process to a semi-automatic one.
- Because every element of the document is tagged with its provenance, it becomes possible to automatically prevent highly secure elements from being seen by those lacking appropriate clearances or privileges.

3 Technical Approach

3.1 The Problem

The last 20 years have led to unprecedented improvements in chip density and system performance, fueled mainly by Moore's Law. During the same time, system and application software have bloated, leading to unmanageable complexity, vulnerability to attack, rigidity and lack of robustness and accountability. These problems arise from the fact that all key elements of the computational environment, from hardware through system software and middleware to application code regard the world as consisting of unconstrained "raw seething bits". No element of the entire stack is responsible for enforcing over-arching conventions of memory structuring or access control; nothing enforces the procedure call (and general stack use) abstraction/encapsulation. Outsiders may easily penetrate the system by exploiting vulnerabilities (e.g. buffer overflows) arising from this lack of basic constraints. Attacks are not easily contained, whether they originate from the clever outsider who penetrates the defenses or from the insider who exploits existing privileges. Because there are no facilities for tracing the provenance of data, even when an attack is detected, it is difficult if not impossible to tell which data are traceable to the attack and what data may still be trusted. Because there is no tracing of accountability, insider attackers have limited reason to fear discovery. Similarly, the lack of data provenance makes it difficult to tell what information must be removed in order to downgrade the classification of a document.

Because there is so little inherent structure, the core of the operating system must be protected from other parts of the OS and especially from application layer code. This is typically done by creating a barrier between the kernel and the rest of the software, in which the kernel operates in a separate address space as does each user process. However, the cost of the context switch between kernel space and user space is normally quite expensive, as is the cost of interactions between separate user processes. As a result the system winds up as a large monolithic kernel possessing many disparate facilities all of them sharing unlimited privileges. In addition, this results in a complex computational model with many mechanisms for interactions between user processes and the OS.

All of this is traceable to the unchallenged assumption that computational resources are scarce, particularly processor chip area. In the context of scarce resources, attempts to optimize the performance at all costs eclipsed the quest for a simple elegant architecture that delivers safety and trustworthiness. With today's abundant resources, we should now return to the task of optimizing for these qualities.

3.2 Overview of The TIARA Architecture

TIARA starts with the premise that we can use some of today's abundant computational resources to fix these critical problems using a combination of hardware, system software, middleware and programming language technology. TIARA will be less vulnerable, more tolerant of intrusions, capable of recovery from attacks, and accountable for its actions. TIARA's design imposes minimal impact on overall system performance.

The TIARA architecture achieves these goals through the judicious use of a modest amount of extra, but general purpose, hardware (the *Tag Processing Unit or TPU* shown in figure 1) that is dedicated to tracking the security context of data at a very fine grained level, to enforcing access control policies, and to constructing a coherent object-oriented model of memory. TIARA's TPU runs in parallel with the main data-paths of the system and operates on a set of extra bits tagging each word with data-type, bounds, and security context information. Operations that violate the intended invariants of the system are trapped, while normal results are tagged with information derived from the tags of the input operands. Because of the critical role of tags in enforcing security properties, we call TIARA a *Security Tagged Architecture (STA).*

Each word in TIARA's memory as well as the contents of each processor register is tagged with a set of extra bits that encode its data-type and its security context. Even the Program Counter (PC) is tagged, allowing the PC to encode the security context of any data that was used in conditional branch instructions. Each process has associated with it a "Principal" representing the current privileges of that process; a processor register holds this value while the process is active.

While TIARA's main data path executes an instruction, the TPU examines the principal register, the tags of both operands, the tag of the PC and the instruction. If executing the instruction would violate any access control policy, then the tag unit causes the process to branch to a "security violation" trap handler; otherwise, the tag unit computes the tag of the result.

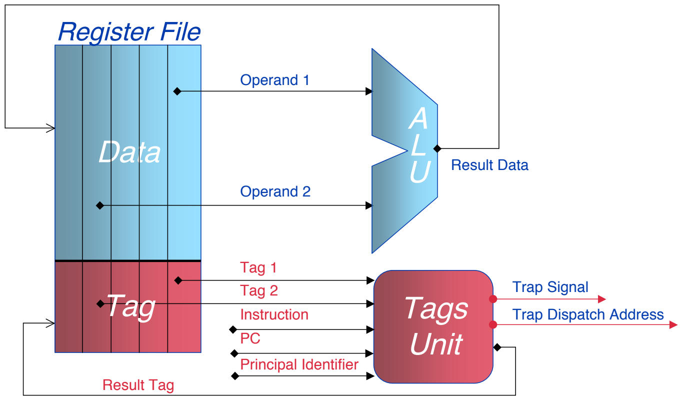

Figure 1: The TIARA Tagged Data Path

Because all words are tagged with their data type, differences between instructions, immediate data (e.g. numbers), and object references are manifest at run-time. TIARA regards all data as objects; in particular, all non-immediate data is accessed through object references that encode both the location and size of the object. All accesses to memory are mediated by object references and all accesses perform bounds-checks in parallel with the load or store, trapping out of bounds references before they take effect.

3.3 TIARA System Layers

The TIARA hardware supports a series of software layers that provide for the enforcement of structuring constraints, access controls and data accountability. The structure of the overall system is shown in figure 2.

The role of the TIARA hardware is to guarantee integrity of the basic memory structuring conventions: All accesses to memory are mediated by object references and are bounds-checked (so there is no ability to overwrite arbitrary areas of memory); no object to which there are existing references can be deallocated (i.e. there are no dangling pointers). In addition, the hardware is responsible for implementing "secure information flow" policies and these policies are enforced at the granularity of the individual word. Each word has both a data-type tag and a security context tag. These later tags form a lattice [11] and the hardware enforces flow policies with respect to this lattice, tagging every result with its appropriate security context.

This establishes a firm and non-bypassable base upon which several layers of software are constructed; each layer provides increasing guarantees and greater accountability. The first of the software layers establishes a consistent object-oriented level of computing while higher layers establish non-bypassable wrappers, access controls, and data provenance tracking. TIARA includes a novel "plan level" of computing in which code is executed in parallel with an abstract model (or executable specification) of the system that checks whether the code behaves as intended.

The first of the software layers is the *Objection Abstraction Layer* in which memory is structured into distinct objects, each characterized by identity, type and extent. This layer erects a computational model that consists solely of the application of a function to a set of objects. TIARA functions are *generic functions*[20], fully polymorphic functions whose implementation is provided by one or more methods. Function invocation involves dispatching to a specific method based on the types of the operands. The fields of an object may be accessed only by invoking a method on that object and these are subject to hardware enforced access controls. This layer also erects a class system in which every object is a member of some class; classes themselves, being objects, are members of a *meta-class*. The main operations of the object system are described by methods on meta-classes and these are the only means for manipulating the internals of classes.

The *Operating System Layer* controls the hardware and manages the core resources of the TIARA architecture. In spite of the name, TIARA does not have an operating system in the classical sense of a distinguished component, executing in a separate context and possessing unlimited privileges. Rather the various functions of a traditional operating system (e.g. the scheduler, memory manager, etc.) are implemented by distinct objects with limited scope of capability and privilege. All objects live within a single flat address space; there is no need to separate kernel space from user space since all objects are inherently protected from one another. Each critical component of the operating system layer (e.g. the scheduler, the virtual memory manager, device drivers) has a limited set of responsibilities and an equally limited set of privileges. These components interact according to a set of system-wide global invariants. Thus, TIARA protects its critical resources by adhering to strict enforcement of the object abstraction and to the principle of least privilege. Objects interact using a single mechanism: the function call and all function calls are checked for compliance with access control policies.

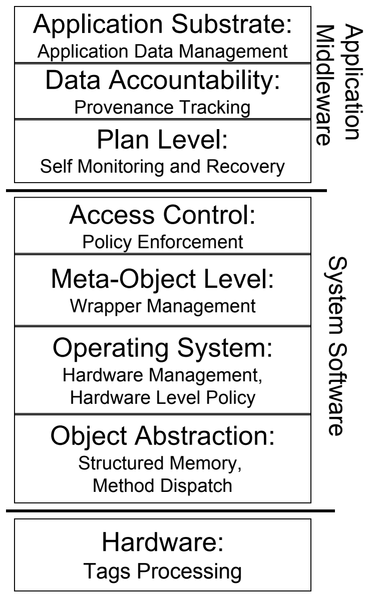

Figure 2: TIARA Layers

The *Meta-Object Layer* is concerned with the imposition of non-bypassable wrappers that are used to provision redundant copies of data, to monitor execution, to track dependencies and to impose access controls. Wrappers are an inherent part of our object model, which is derived from the *Meta-Object Protocol* (MOP) of CLOS [20]. A wrapper is simply a distinguished type of method that is combined with other methods in such a way that the wrapper gets control before other methods, allowing it to control whether the other methods are invoked and with what arguments. In addition, the wrapper method gains control after the other methods execute, allowing it to capture and/or modify the returned results.

The *Access Control Layer* is capable of supporting a variety of role-based (and other) access control models using capabilities provided by the lower layers. As mentioned earlier, TIARA's hardware guarantees that every datum is tagged with both its data-type and its security context; the processor checks every instruction to make sure that the executing process has the privileges necessary to execute that instruction on data from the security contexts of the operands.

The access control layer supplements these hardware checks using *access control wrappers* that check whether it is legitimate for the executing process to invoke the indicated function on the operands. It does this by checking the security context and data-type parts of the operands tags. Access control wrappers are imposed using functions provided within the MOP. Since these MOP entry points are themselves just normal functions, access control wrappers may also be imposed on them, allowing us to use the normal access control mechanisms to control who is allowed to impose a wrapper.

Access control wrappers typically are generated from a more abstract description of an access control policy written in an Access Control Language. It is not a major part of our agenda to develop such a language, however we will create a simple one to drive our early work and then hope to test our approach on formalisms developed by other projects within NICECAP. We anticipate that TIARA can support a broad range of access control policies.

The *Plan Layer* uses system-wide models of the intended behavior to enforce constraints on control and data flows, and to check intended invariant conditions. If the behavior of the system does not correspond to the intended behavior predicted by the system model, then execution is aborted and the dependency records are used to diagnose the cause of misbehavior, to identify data that should not be trusted and to identify what individuals might have been responsible for the failure.

The *Data Accountability layer* is responsible for tracking the provenance of data as directed by the plan layer. Part of this is accomplished by the TIARA hardware which tags all data with its security context. *Data Accountability wrappers* are used to capture the inputs and outputs of all functions of interest and to build dependency records linking the outputs to the inputs and to the invoked function. In addition, data accountability wrappers are interposed around all HCI input and output operations. Output functions track what data has been exposed to which users while wrappers around input routines track which users have contributed data to the computational process.

The *Application Substrate* uses the features of all the lower layers to provide a platform for applications that rely on accountable information flow. It provides a "truth-maintenance" capability that allows conclusions in application documents to be removed if the raw data or computations that led to them are no longer trusted; it also provides a Bayesian reasoning substrate that assesses the trust that should be placed in conclusions given the trustworthiness of the inputs (e.g. sensor data, human judgment, computational tools) to the intelligence analysis process. This substrate also provides tools that hide sections of a document whose security context requires more privileges than are possessed by the audience.

The end result is that application systems are continually checked at runtime to see if they behave as intended, violations of intended invariants are detected and prevented and the provenance of all data is made manifest. Thus, even when attacks succeed, the system knows what data is trustable and what data is suspect. In addition, all data can be traced back to the computational processes, raw data and individuals from which the data was derived. Bayesian inference techniques are used to rate the reliability of conclusions given the trust accorded to the contributing inputs, computational processes and human analysts. Finally since every datum is tagged with its security context, it is easy to identify sensitive data that must be shielded from users without adequate privileges. Thus, an Active Briefing Book can dynamically adjust itself for viewing by users of different privilege levels and can rapidly reassess the reliability of its conclusions and recommendations if the reliability of any of its inputs is called into question.

These techniques derive from research over the last decade in the MIT ARIES and AWDRAT projects [4, 3, 46]. In the rest of this proposal we present more of the details of each of these proposed software layers as well as a description of the proposed core TIARA hardware.

3.4 TIARA Hardware

At a high level, our goal is to track information flows in order to understand the provenance of significant data, to provide confinement [25] and to guarantee that all information flow and access control policies are adhered to. The TIARA hardware contributes to this goal by systematically tracking the security context of all data in the machine, guaranteeing that no process can gain access to data for which it lacks adequate privileges.

The key TIARA hardware structure is shown conceptually in figure 1. The upper half of the picture might be virtually any conventional processor design; it fetches operands from a register file, combines them through an ALU in accordance with the current instruction and then writes the results back into the register file. The lower half is the Tag Processing Unit (TPU); this fetches the tags of the operands from the register file, the identifier of the processes's current principal from the principal register, the tag of the Program Counter and the current instruction and produces a new tag that that is written back into the tag section of the register file as well as a new tag for the Program Counter. The role of this TPU is to enforce basic structuring conventions, data type consistency and compliance with access control policies while imposing minimal delays.

In addition to the TPU, TIARA provides a few other novel pieces of hardware: A Bounds Checking Unit (essentially an adder) to guarantee that accesses to objects are within range; the Garbage Collection Support Unit which consists of a few specialized registers and some trap logic; and a specialized hardware stack that help in managing the security context of the program counter. We will next describe these, starting with the implementation of the TPU.

3.4.1 Ensuring Secure Information Flow

In TIARA, the machine word is the unit of information: each slot of memory and each register contains one word. A word consists of a tag and a value where the tag encodes the data-type and security context. Words are read and written atomically. The program counter associated with a process is tagged with a security class just like any other word. The "Principal Register" of the processor holds a word representing the privileges of the currently running process.

The security context part the tag is used to encode some aggregation of data that is to be treated uniformly from the point of view of security and information flow policy. Security contexts form a lattice; we denote by lub(A,B) the least upper bound in this lattice of the tags A and B.

An information flow policy is simply a set of rules such as A $\rightarrow$ B stating that information is allowed to flow from A to B; these rules are transitive. Flows not included in the transitive closure of the stated rules are disallowed. One goal of the TIARA hardware is to guarantee that these rules are followed. A more detailed treatment of this is provided in [4]; we provide an overview of the approach here.

As each instruction is executed, the TIARA hardware first checks that the operation is consistent with its policies; if so, it then computes a new security tag for both the program counter and the new result. The rules for the new tags are as follows:

- On a non-branching instruction I, acting on operands A and B:
  - The tag of the result is lub(tag(A), tag(B), tag(PC), tag(I)).
  - The tag of the PC is lub(tag(PC), tag(I))
- On a non-conditional branch instruction I: The new tag of the PC is lub(tag(PC), tag(I)).
- On a conditional branch instruction I, branching on data A: the tag of the PC is lub(tag(A), tag(PC), tag(I))

Whenever the processor takes a conditional branch, the tag of the current PC is pushed on an internal stack. When execution rejoins after the branch, this stack is popped and the tag of the PC is restore to its earlier value. This guarantees that the security context of the PC strictly represents the contexts of the precise set of data that has influenced the flow of control.

The basic information flow constraint is that no process can read a data value unless the flow policy allows information to flow from the security context of the datum being read to the security context of the the Principal.

Information flow policy also deal with I/O channels. Both input and output channels are assigned security context labels. Data coming in through an input channel is tagged with the channel's security context and may only be loaded into the processor if the information flow policy allows flows from the security context of the channel to the security context of the Principal of the current process. No data may be written to an output channel unless the information flow policy allows data to flow from the security context indicated by the datum's tag to that indicated by the I/O channel's tag.

A principal may be authorized to temporarily replace itself with another principal as the principal upon whose behalf the current process is running. We implement this with a hardware-supported "role" stack. Using this mechanism, we can implement Role-Based Access Control [16, 40], – a principal may take on any of a variety of "roles" (i.e. other principals), without ever having the access-rights of more than one of those roles at a time. The role stack is also quite useful for logging exactly which principals performed what actions in what roles.

3.4.2 The HEX Unit

The key to implementing all of this is to quickly compute the new tags of the PC and of the result while checking that the current operation is consistent with information flow and access control policies. We do this using a lookup table scheme that is supported by the TIARA hardware.

There have been several previous tagged computer systems that used tag bits to make the types of data manifest at runtime. Going far back, the Burroughs 5500 and its successors, used 3 tag bits (and a 48-bit address field) to encode a range of primitive data types; the Lisp machine [31, 15] used a full 8-bit tag. (The Intel-432 and the IBM series 38 machines also employ tagging systems).

However, TIARA uses a completely novel approach that is considerably more flexible and that deals with both data typing and security issues simultaneously. This is the the *Hash Execution (HEX)* unit shown in figure 3. The HEX is a similar to a data cache or a TLB that operates on tags (i.e. on the type and security context of the data). Under program control, certain fields of the operands, the PC, and the instruction are identified as "tag bits" that encode data types, and security context. The HEX extracts these bit-fields and hashes them into an address used to retrieve data from a set of RAMs that act like an n-way set-associative cache. The data retrieved from the cache contains a "trap flag" indicating whether the operation being attempted is prohibited; if so the operation is aborted and the processor is dispatched to an appropriate error handling routine. Otherwise, the information retrieved by the HEX includes the data type and security context of the result; these are combined with the result produced by the ALU and stored back into the register file. Similarly the updated tag of the PC is calculated and written back.

Like a cache, the HEX unit may miss; i.e. it might encounter a set of principal, tags and instructions that it hasn't yet seen. In this case it consults the *Policy Table* in main memory (just as a TLB would consult the page table) loads the entry into its memory and then resumes. As with other caches, the overhead of misses can be relatively large, but if the cache is of appropriate size, then misses will happen infrequently. Part of the design effort will be to study the temporal locality of the information managed by the HEX and determine appropriate sizes for its memories.

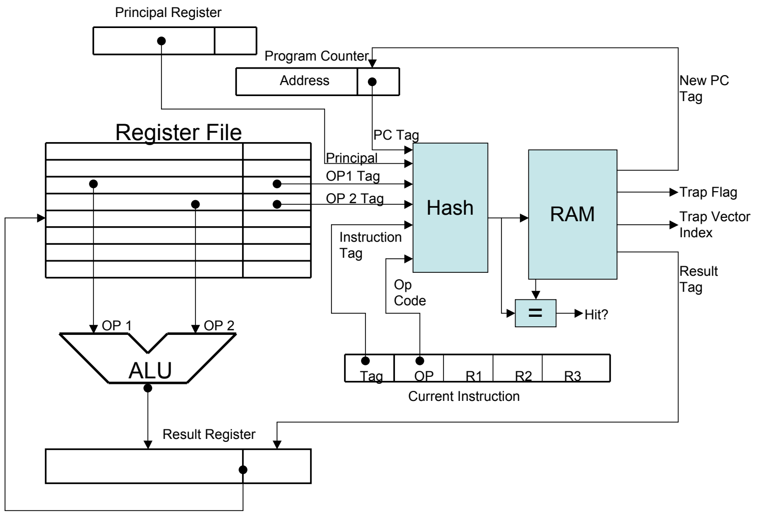

Figure 3: Implementation of Tagged Data Path Using HASH EXECUTION Units

As an example, it is easy to see how multi-level security would be implemented within this framework: The tag of each operand includes a field that encodes the security level of the data, while the HEX lookup finds the maximum of the two input tags and inserts that into the result. Thus, each data word automatically takes on the highest security level of all the input data. On a load operation the HEX would hash the security tags of the object reference as well as the principal register; if the principal register contains a value lower than that of the object reference, then a trap signal is emitted, otherwise the operation proceeds.

The contents of the memories addressed by the HEX may be reloaded dynamically to support varying needs of the computation. In particular, instituting a change in security policy on the fly involves little more than flushing the HEX memories and swapping the pointer to the Policy Table. These mechanisms are describe in more detail in the technical reports presented in [4, 3].

In figure 3 we show the conceptually simplest approach, which uses a single HEX unit. However, in practice it may be more appropriate to use several different HEX units to lookup the individual outputs, since this may lead to a more compact implementation and a higher hit rate. Part of our hardware design task will involve analyzing these tradeoffs in order to fix on an appropriate design. We will do this using both software simulation and prototype implementations in high performance Field Programmable Gate Array (FPGA) technology.

3.4.3 Hardware Support for Garbage Collection

TIARA's hardware must provide an object-oriented model of memory whose structuring conventions may not be violated. Rather than leaving storage management up the programmer, with the attendant risks of memory leaks and dangling pointers, TIARA instead provides for automatic, real-time garbage collection of its memory. The idea that garbage collection is a fundamental service is becoming widely adopted; for example, the Java Virtual Machine (JVM) and the Microsoft Common Language Runtime (CLR) are both garbage collected environments.

During our experience with the Lisp Machine project we developed several simple hardware features that allow the garbage collector to operate in real-time and that avoid any significant run-time overhead. The TIARA hardware will provide similar facilities; these involve little more than the ability to check each load or store operation for whether a word is a pointer data type (which is provided by the TPU) and if so whether the address points into a distinguished area of virtual memory. These hardware checks run in parallel with the normal load/store datapath and suffice to erect the *read-barrier* and *write-barrier* of the GC algorithms that are documented fully in [30].

3.5 The Object Abstraction

The hardware features outlined above are the primitive building blocks for creating a software abstraction layer in which 1) All of memory is regarded as consisting of objects with definite type, extent and identity and 2) All operations performed on these objects are semantically meaningful and consistent with the object types. Thus, raw pointers to arbitrary memory locations are replaced by "object references". The use of arbitrary operations on raw data is replaced by bounds and type-checked semantic operations on structured objects. In particular, raw pointer arithmetic, buffer overflows and the storing of data in arbitrary locations are impossible. This layer is referred to as the Object Abstraction and it represents the base upon which a variety of software object models (e.g. those of C++, Java, C#, Python, Aspect-J, or Common Lisp) may be constructed.

We will employ an object model that is a generalization of these specific object models and that draws heavily on the ideas in CLOS [20] and Aspect-J [21]. The model provides for a class lattice (i.e. multiple inheritance) and for multi-argument method dispatch (as is done in CLOS). This supports a view that combines functional and object oriented programming: A function dispatches to a method that is consistent with the types of its operands; if there are no applicable methods then the application of the function to its arguments is illegal and traps. The HEX unit, described above, efficiently supports method dispatch.

3.6 The System Software Layer

The System Software Layer is responsible for implementing the functionality provided by a conventional operating system such as interrupt handlers, device drivers, trap handlers, management of the physical hardware, resource allocation and scheduling, virtual memory management, persistent storage, authorization and authentication.

In TIARA, each of these is implemented as an individual object with internal storage that is isolated from other components of the operating system layer and that has extremely limited privileges. For example, the scheduler maintains internal information about processor usage and about the priority levels of both system and user processes; this information is inaccessible to other system software components. Conversely, the scheduler has no privileges to access the internals of processes; it cannot for example, read or modify the internal data structures of a process. Device drivers, in particular have extremely limited access rights; they are given a standard object reference with base and bounds that describes the block of memory that is be read or written. They have no other ability to access memory and cannot overwrite storage at random. A major goal of our design efforts will be to determine precisely what module boundaries make sense and what security contexts are needed to enforce the contexts.

3.7 The Wrapper and Meta Control Layer

In addition to these basic features, our object model provides for a "Meta Object Protocol" (MOP) in which classes are regarded as instances of other classes whose methods implement the basic operations of the base classes (e.g. method dispatch, object creation). This layer also provides for method combination (as in CLOS [20] and Aspect-J [21]); in particular this allows for "wrapper methods" to be applied to base methods to implement the concerns of a distinct aspect such as access control or data provenance tracing. Wrapper methods execute before and after the base method and control whether the base method is executed at all. This allows us to construct a series of more abstract software layers that provide for access control, flow monitoring and logging and for self-checking and diagnosis.

Since classes are themselves objects, their behavior is described by another set of classes, commonly termed "Meta-Classes". Methods on these classes control how normal classes behave; in particular they describe how method combination and method dispatch is implemented. This allows us to distinguish different classes of wrappers, such as those used by TIARA internally to implement access and flow controls, from normal user level wrappers. In addition, the meta-level allows us to control who is allowed to impose such wrappers, making the imposition of security-oriented wrappers non-bypassable.

Wrappers are available as part of the toolkit of the application programmer; however, there are a distinguished set of meta-level wrappers that are imposed by system software to implement access controls and to track data provenance. Such wrappers are non-bypassable and access controls imposed at the meta-object level restrict the use of such wrappers to processes executing with special privileges. Since these access controls are ultimately enforced by the hardware, this allows us to create a very strong base for security and provenance management.

3.8 The Access Control Layer

Using the object abstraction as a base, we can provide for the efficient enforcement of a variety of access control policies. The outline of this is as follows: We associate with each process a stack of "Principals", where a principal represents the privileges extended to the current process; the process may dynamically (and temporarily) assume a new principal, but it must authenticate itself to do so. All operations on data are ultimately performed by dispatching to a method based on the data type and security contexts of the operands. TIARA extends this notion to include the principal in the method dispatch. Thus a method is dispatched to only if the privileges of the Principal are consistent with the data types of the operands; otherwise an illegal operation is signaled. Role-based access control systems (and a variety of other access control schemes) are easily mapped into this framework: The role is encoded in the Principal; The type hierarchy is encoded in the object-type lattice; The operation type is encoded in the method-name. In our model, the MOP and its use of meta-classes and wrapper methods allows us to simultaneously implement a variety of access control policies. Furthermore, these are all implemented through method dispatching and this is efficiently supported by the HEX hardware.

In many ways, TIARA is similar to a capability architecture [32, 27, 19, 42]. However there is an important distinction. Capability systems mix together an object reference with the set of access rights; thus, when a process transmits a capability from one component to another, it transmits not only knowledge of the referenced component, but also a set of rights to access that component in particular ways. This leads to problems with managing revocation of privileges and with controlling the amount of privilege that is delegated. Although there are ways of dealing with these issues in appropriately structured object oriented capability systems [28], in our view these impose unneeded complexity.

TIARA separates knowledge of an object (which is carried in a normal object reference) from rights to access that object (which is represented in the Principal Register of a process). One can think of this in terms of Lampson's notion of an access control matrix [24], which uses one dimension to represent Principals and the other other to represent Objects. Each cell of the matrix represents the access rights of a principal and to an object. Capability system cut this matrix up into columns while Access-control lists cut it up into rows. TIARA, instead, operates on the matrix directly, since it represents the principal separately from the objects that are being accessed.

3.9 The Plan Layer

The Plan Layer is responsible for monitoring whether a process is behaving as intended; the design of this layer derives from our work on the AWDRAT [46] system within the DARPA SRS program. This layer is driven by an abstract model, or Plan, of the intended computation; this consists of a hierarchical block diagram of the computation, annotated with data- and control-flow links between the components and Pre- and Post-conditions around each component. This abstract description drives the synthesis of wrapper methods that are used to gather information on entry and exit from the relevant computational components and to check that this information is consistent with the constraints of the Plan. The previous layers check that every operation is permitted and semantically meaningful within its local context; this layer checks that the global constraints of the computation are respected. In addition, the plan layer is responsible for capturing and provisioning redundant copies of data to support recovery from attacks. This backup data is used by the plan layer, together with its model of the computational process, to facilitate the reconstitution of a consistent state of the computation from which the computation may proceed.

3.10 The Data Provenance Layer

The Object layer guarantees that only semantically sensible operations may be performed; the Access Control layer strengthens the guarantee to rule out operations that are inconsistent with access policies.

The data provenance layer provides for accountability by tracking how the current state of each object is dependent on the state of other objects. The HEX hardware performs the most basic part of this operation since it performs fine-grained "tainting" of data (e.g. it maintains the security context for each individual word). At a higher level, the data provenance layer builds links that connect an object to those other objects that were used in computing its current state. These links represent dependencies and the dependencies form a network similar to those of "Truth Maintenance Systems" (TMS) and Bayesian networks used in Artificial Intelligence.

These capabilities, like those of the Access Control layer, are implemented by wrapper methods that work in tandem with the hardware's HEX units.

3.11 The Application Substrate Level

The Application substrate level provides a variety of services that are supported by the dependency network built by the data provenance layer. As mentioned in section 3.10 this dependency network is similar to the data structures of a Truth Maintenance System (TMS). One of the key capabilities provided by a TMS is "assumption retraction" which invalidates all statements that depend on an abandoned assumption. The equivalent operation for the application substrate Layer is performed whenrp some datum is identified as having been corrupted by an attack; at this point the Data Provenance dependencies are traced to identify all data whose current state was derived from the corrupted datum and to mark these data as untrusted.

The dependency network is also similar to the data structures of Bayesian networks and it is used for probabilistic reasoning. The Data Provenance layer dependency network can also be used to support such reasoning: Each raw observation, whether it comes from sensor data or from human sources, is accorded a degree of trust, represented as a probability. For each step in the processing of intelligence data, the data provenance layer builds a link in its dependency network which can be interpreted a stating how the trustworthiness of the conclusion depends on the trustworthiness of the inputs as well as on the reliability of the interpretation step itself. Bayesian inference algorithms in the application substrate layer estimate the trustworthiness of the final conclusions (including competing conclusions) and update these estimates when the estimated reliability of any input changes.

The final service provided by the application substrate layer is the ability to identify all elements of an Active Briefing Book document that exceed the privileges of the current viewer of the document and to aid in the downgrading of these elements. For example, high resolution imagery can be "deres'd" to make it viewable at a lower classification level; other information may need to be excised. Because the application substrate layer has the ability to examine the security-context tags maintained by the hardware it can easily identify these elements.

3A Results of related prior and current government-funded R&D efforts

Howard Shrobe

Shrobe has led several DARPA sponsored efforts in the areas of information survivability, software and hardware architectures. He was the PI of the Automatic Trust Management for Autonomous Survivable Systems project in the OASIS program [43, 45, 44] and PI with Dr. Robert Balzer of the AWDRAT (cognitive immunity) and PMOP (insider threat) projects in the Self Regenerative Systems program [46]. He was also PI with Jon Doyle in the CyberPanel program [14, 13]. His research on self-adaptive software and plan driven computing has been funded by the DARPA EDCS and ANTS and Pervasive Computing programs [23, 18, 17, 37, 38]. Shrobe was also the PI of a project in the DARPA Intelligent Collaboration and Visualization program during which he developed an application similar to the Active Briefing Book concept described in section 2.2

Thomas Knight

Knight is responsible for the microarchitecture, physical design, construction and debugging of the first lisp machine systems in 1977, sponsored initially by unrestricted funds from IBM, and later by DARPA funding [36]. He led a detailed architectural, physical, and software design of the cross-omega, an SIMD architecture with highly efficient interprocessor communications in collaboration with General Electric Research, but further architecture development was not funded by DARPA. His concepts for high performance interconnect later were patented as the "multibutterfly" and became the central design principle of the DARPA-sponsored TRANSIT [5] and METRO [7] architectures. For these designs, novel chip-chip signalling technologies, cooling, and packaging techniques were developed in addition to the high level architecture. Knight is also responsible for the first transactional memory system design [22], now the centerpiece of much modern architectural research. More recent work under the DARPA Data-Intensive program has centered on techniques for efficiently implementing provenance based computation and truth maintenance systems, and for improving the representations of object structures and pointers in symbolic language implementations for efficient hardware implementation [4, 3].

Andre DeHon

DeHon has led the development of novel, high-performance chips, designs, and systems, including CAD and compilation software to support them over a number of government-funded research efforts. Early efforts include the implementation of the METRO, a low-latency fault-tolerant router [7], MLINK network interface, and MBTA parallel processing system (DARPA embedded microsystems). He led the development of several novel, spatial reconfigurable (FPGA-like) architectures including the DPGA [47], TSFPGA, MATRIX [29] (under DARPA embedded microsystems) and the HSRA [48] (under DARPA ACS program); all these designs included custom VLSI layout and fabrication. The MATRIX effort led to the foundation of Silicon Spice, which was later acquired by

Broadcomm. He also led the development of the SCORE stream-based scalable compute model for spatial computing, including compilation and run-time software support [8] (under DARPA ACS and NSF ITR). He has further overseen the development of numerous high-performance designs built on top of commercial FPGAs, including pipelined saturated accumulation [35] and sparse-matrix vector multiply [9] (under NSF ITR). Current work includes the development of an FPGA-based Graph Machine [10] (under DARPA ACIP) and architectures for molecular-scale electronics [6] (under DARPA Moletronics and MoleApps).

4 Evaluation Approach

During the course of the 18 month effort we will attempt to implement the core of the entire stack shown in figure 2. We will develop a minimal concept demonstration of an application similar to the active briefing book concept described in section 2.2 and we will use this to drive the underlying hardware and software system. We will then attempt to demonstrate, study and evaluate several key properties of the TIARA system.

Our work will be structured into three major tasks: Hardware, System Software (corresponding to the Object Abstraction, Operating System, Meta-Object and Access Control levels of figure 2) and Application Middleware (corresponding to the Plan, Data Accountability, and Application Substrate levels of figure 2).

We assume that there will be PI meetings at roughly 6 months intervals and accordingly our plan calls for 3 major cycles of development and evaluation as shown in the Chart of Milestones in section 5. In each cycle, we will design and implement enough of each of the three major task areas of the TIARA system to facilitate study and document key properties of each component as well as those of the overall system. Each cycle will revise the designs based on knowledge gained from the previous cycle and will then further enhance the design in order to reach fuller capability. By the end of the 18 months cycle we do not anticipate having a fully integrated, production ready system; we do, however, anticipate having a rich enough version of each major area so that we can collect useful data, make informed arguments about the overall safety properties and data accountability capabilities of the system.

4.1 Hardware

We will design the TIARA hardware using both simulations and physical realizations in Field Programmable Gate Array (FPGA) technology. The main data path of the processor will be an extremely simple RISC-like processor. Our focus will be on the features peculiar to TIARA. There are three major sub-tasks associated with this task:

1. Design and Simulation
2. Physical Realization in Field Programmable Gate Array (FPGA) technology
3. Testing and Analysis

The major issues of interest include:

- What are the space and time overheads for TIARA features?
- What is a good size for the security-tag field of a machine word?
- Should the HEX unit be decomposed in to several special purpose HEX units, and if so how many serving what purposes?
- How large do the HEX units need to be?

The first design iteration will focus on the HEX units, and associated hardware (e.g. management of the Program Counter tag). In the second iteration, we will include a trial FPGA implementation so the we may assess the delay and space associated with the HEX and we will use standard, off the shelf FPGA designs for the normal processor core. In the third iteration, we will include TIARA specific design elements to support memory management, bounds checking and garbage collection, and method dispatch.

The final iteration will allow us to collect data from a system running the complete (but bare-bones) stack of software.

We will document the results of each iteration and present these at PI meetings.

4.2 System Software

The system software includes both components that are typically found in an operating system (e.g. Scheduler, virtual memory manager, device drivers) and many that are not. These include the features that contribute to the Object Layer (e.g. the garbage collector, the class system), the Meta-Object layer (e.g. the meta-class system, wrappers) and the Access control layer (e.g. implementation of access control formalisms via wrappers).

There are 3 sub-tasks associated with this task

1. Design and Prototype the several layers of system software.
2. Outline a Proof of the key security and accountability properties delivered collectively by the system software and hardware.
3. Testing and analyzing the vulnerability of the system software under a variety of attack scenarios and also testing how well the system software can implement a variety of access control and data flow formalisms.

The key questions to be addressed are:

- How should the core OS layer components be modularized? What internal data does each need to manage? What are the invariant conditions that each needs to maintain? What are the overall invariants that each needs to adhere to?
- What set of security contexts needs to be reserved to support the System Software's internal needs?
- How can we guarantee that the protections of the OS layer are not bypassable?
- How well can the wrapper mechanisms implement access control and information flow formalisms both of our own design as well as those of other participants in NICECAP?

As with the hardware design, we will organize this effort into 3 iterations. The first will focus on the Object Layer and OS layer capabilities since these impact the hardware design most directly. The second iteration will focus on the Meta-Object layer and the exploration of a simple access control formalism of our own choosing. In the third iteration we will include consideration of other formalisms.

We will document the design of the system software layer. We will first evaluate the security properties delivered by the hardware and system software constructing a detailed but informal proof of these properties that could serve as the outline of a more formal proof. We we also evaluate these properties through use of simulated attacks, assessing how (in)vulnerable the system is. Finally we will also assess how well the TIARA system serves as a base for implementing access control and information flow systems, by testing at least one such formalism within the TIARA framework (we are aware of a proposal from Kestrel Institute which would serve this purpose well were it also selected as a NICECAP project).

4.2.1 Application Middleware

The application middleware includes the software components at the Plan, Data Accountability, and Application Substrate levels. These collectively support: The statement and enforcement of overall application system flow properties; The collection of application data provenance information; The use of higher-level data provenance and hardware-based security context information to support application level containment of privileged information as well as the management of trustworthiness assessments of application data.

This task has two sub-tasks:

1. Design and Prototyping of the application middleware components
2. Experimentation and Analysis of Application middleware components

We will conduct these tasks in two cycles, delaying the initiation of the task until after the system software has progressed enough to influence the application-oriented middleware. For programmatic reasons we may implement this layer of the system in Common-Lisp, but in a way that is faithful to the underlying TIARA model.

The major issues of concern are:

- Can we express a “Plan level” description of an application with adequate fidelity but without imposing unsupportable extra burden on application developers?
- Can we implement the flow control constraints expressed at the plan level without excessive overhead?
- Can we capture rich enough data provenance without excessive overhead?
- Can the captured provenance information be rich enough to support the needs of applications geared to NIC needs such as the“Active Briefing Book” concept?

To study these issues we will experiment with a bare bones, concept demonstration, implementation of an “Active Briefing Book” application. We will experiment with building “Plan Level” models of this application, measure the overhead imposed by the wrappers and test the effectiveness of the framework.

We will document both the designs and the results of these experiments.

5 Schedule and Milestones

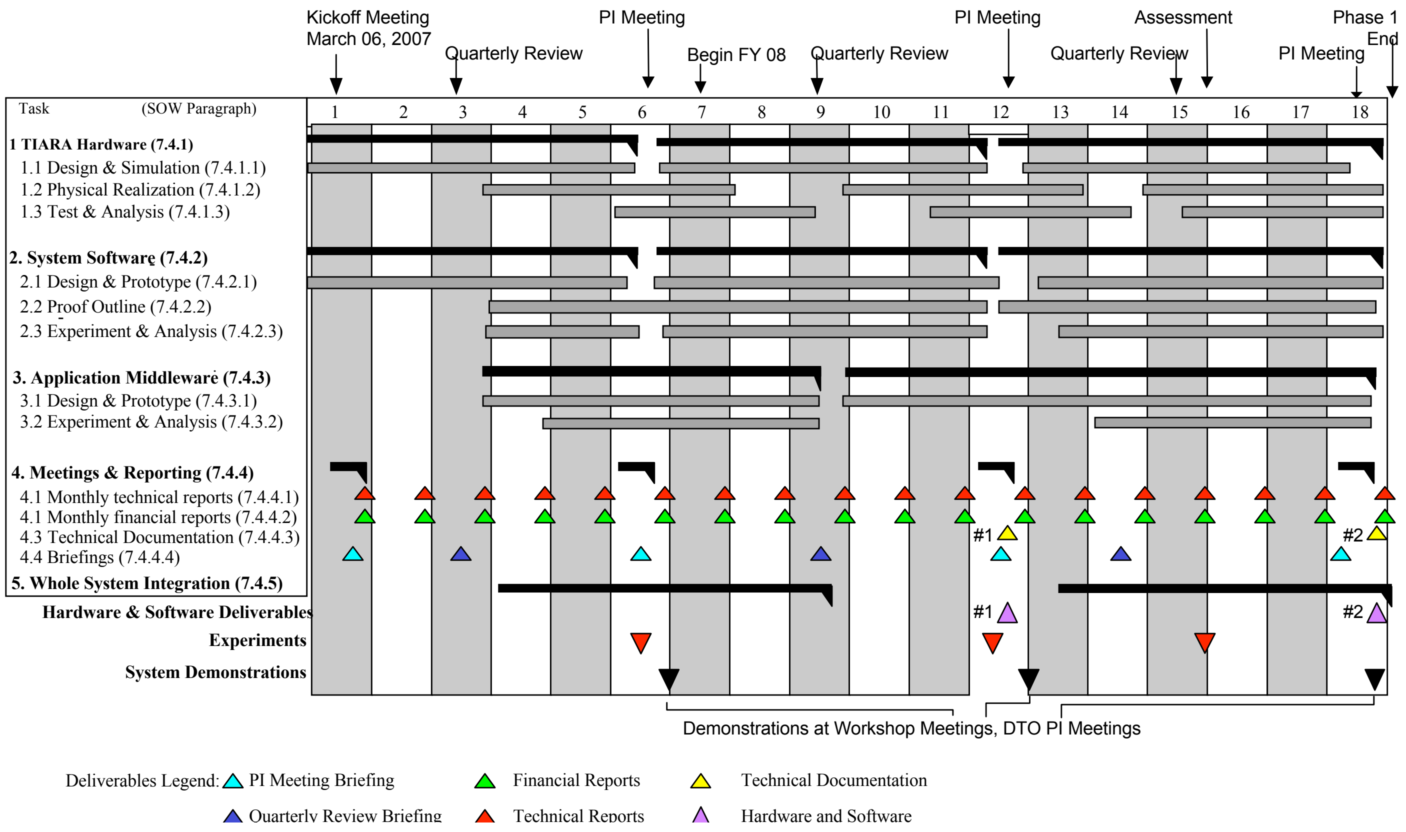

Figure 4: Tasks and Milestones

The schedule chart is shown in figure 4. The tasks correspond to those described in Section 4 and in the Statement of Work. The major milestones correspond to PI meetings (assumed to be every 6 months) and to the initial evaluation at month 15 mentioned in the BAA.

6 References

[1] John Barkley. Implementing role-based access control using object technology. In *First ACM Workshop on Role-Based Access Control*, November 30 – December 1 1995.

[2] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretations. Technical Report Tech. Report MTR-2997, MITRE Corp., July 1975.

[3] Jeremy Brown, J.P. Grossman, Andrew Huang, and Jr. Thomas F. Knight. A capability representation with embedded address and nearly-exact object bounds. Technical Report Aries Project Technical Report 5, MIT AI Lab, April 2000.

[4] Jeremy Brown and Jr. Thomas F. Knight. A minimal trusted computing base for dynamically secure information flow. Technical Report Aries Project Technical Report 15, MIT AI Lab, Novemeber 2001.

[5] Frederic T. Chong and Jr. Thomas F. Knight. Design and performance of multipath min architectures. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 286–295, New York, NY, USA, 1992. ACM Press.

[6] André DeHon. Nanowire-Based Programmable Architectures. *JETC*, 1(2):109–162, 2005.

[7] André DeHon, Frederic Chong, Matthew Becker, Eran Egozy, Henry Minsky, Samuel Peretz, and Thomas F. Knight, Jr. METRO: A Router Architecture for High-Performance, Short-Haul Routing Networks. In *ISCA*, pages 266–277, May 1994.

[8] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream computations organized for reconfigurable execution. *JMM*, 30(6):334–354, September 2006.

[9] Michael deLorimier and André DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *ISFPGA*, pages 75–85, February 2005.

[10] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomás E. Uribe, Thomas F. Knight, Jr., and André DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *ISFPCCM*. IEEE, 2006.

[11] D. E. Denning. A lattice model of secure information flow. *Communications of the Association of Computing Machinery*, 19(5):236–243, May 1976.

[12] P. J. Denning. Fault tolerant operating systems. *Computing Surveys*, 8(4):359–389, December 1976.

[13] Jon Doyle, Isaac Kohone, William Long, Howard Shrobe, and Peter Szolovits. Agile monitoring for cyber defense. In *Proceedings of the Second Darpa Information Security Conference and Exhibition (DISCEX-II)*. IEEE Computer Society, May 2001.

[14] Jon Doyle, Isaac Kohone, William Long, Howard Shrobe, and Peter Szolovits. Event recognition beyond signature and anomaly. In *Proceedings of the Second IEEE Information Assurance Workshop*. IEEE Computer Society, June 2001.

[15] C. Baker et al. The symbolics ivory processor: a 40 bit tagged architecture lisp microprocessor. In *Proceedings of the 1987 IEEE International Conference on Computer Design*, pages 512–515, October 1987.

[16] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, October 13–16 1992.

[17] Krzysztof Gajos, Harold Fox, and Howard Shrobe. End user empowerment in human centered pervasive computing. In *Pervasive 2002*, 2002.

[18] Krzysztof Gajos, Luke Weisman, and Howard Shrobe. Design principles for resource management systems for intelligent spaces. In *Proceedings of The Second International Workshop on Self-Adaptive Software*, Budapest, Hungary, 2001. To appear.

[19] A. H. Karp, R. Gupta, G. J. Rozas, and A. Banerji. Using split capabilities for access control. *IEEE Software*, 20(1):42–49, January 2003.

[20] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Number ISBN 0-201-17589-4. Addison-Wesley, 1989.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[22] Thomas F. Knight. An architecture for mostly functional languages. In *Proceedings of ACM*

Lisp and Functional Programming Conference, pages 500–519. Aug 1986.

[23] Robert Laddaga, Paul Robertson, and Howard E. Shrobe. Probabilistic dispatch, dynamic domain architecture, and self-adaptive software. In Robert Laddaga, Paul Robertson, and Howard Shrobe, editors, *Self-Adaptive Software*, pages 227–237. Springer-Verlag, 2001.

[24] Butler W. Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. (reprinted in Operating Systems Review, 8,1, January 1974, pp. 18 - 24).

[25] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[26] Carl E. Landwehr. Formal models for computer security. *ACM Comput. Surv.*, 13(3):247–278, 1981.

[27] H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[28] M. Miller, K. Yee, and J. Shapiro. Capability myths demolished, 2003.

[29] Ethan Mirsky and André DeHon. MATRIX: A Reconfigurable Computing Device with Configurable Instruction Distribution and Deployable Resources. In *Hot Chips Symposium 1997*, 1997.

[30] David A. Moon. Garbage collection in a large lisp system. In *Proceeding of the ACM Conference on Lisp and Functional Programming*, pages 235–246, 1984.

[31] David A. Moon. Architecture of the symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, 1985.

[32] A. S. Tanenbaumand S. J. Mullender and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[33] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[34] E.I. Organick. *The MULTICS system: An examination of its structure.* The MIT Press, 1972.

[35] Karl Papadantonakis, Nachiket Kapre, Stephanie Chan, and André DeHon. Pipelining Saturated Accumulation. In *ICFPT*, pages 19–26. IEEE, December 2005.

[36] T. Knight et al. R. Greenblatt. The lisp machine. In *Interactive Programming Enviornments.* McGraw-Hill, 1984.

[37] P. Robertson, R. Laddaga, and H. Shrobe. *Self-Adaptive Software.* Springer-Verlag, 2000.

[38] P. Robertson, R. Laddaga, and H. Shrobe. *Self-Adaptive Software: Applications.* Springer-Verlag, 2002.

[39] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[40] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[41] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3), March 1972.

[42] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[43] Howard Shrobe. Model-based diagnosis for information survivability. In Robert Laddaga, Paul Robertson, and Howard Shrobe, editors, *Self-Adaptive Software.* Springer-Verlag, 2001.

[44] Howard Shrobe. Computational vulnerability analysis for information survivability. In *Innovative Applications of Artificial Intelligence.* AAAI, July 2002.

[45] Howard Shrobe. Model-based diagnosis for information survivability. In *Proceeedings of the International Workshop on Principles of Diagnosis, DX-02*, May 2002.

[46] Howard Shrobe, Robert Laddaga, Robert Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, and Tim Hollebeek. AWDRAT: A Cognitive Middleware System for Information Survivability. In *Innovative Applications of Artificial Intelligence.* AAAI Press, July 2006.

[47] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[48] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid

Rowhani, Varghese George, John Wawrzynek, and André DeHon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *ISFPGA*, pages 125–134, February 1999.