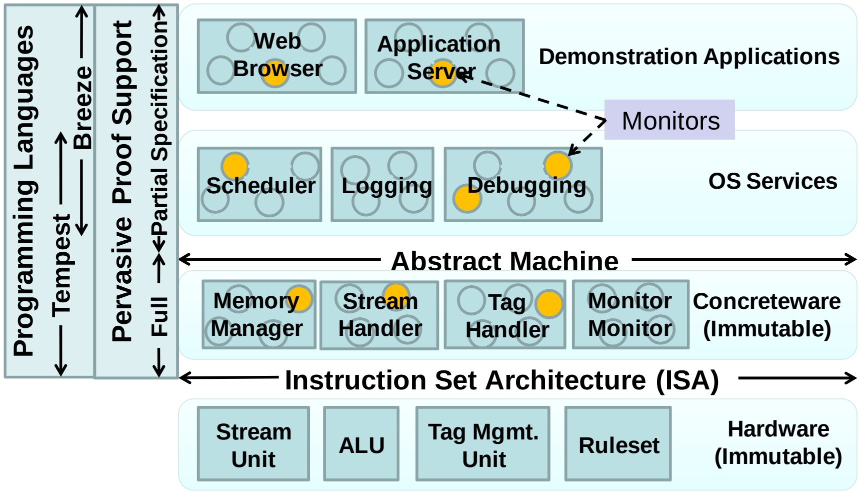
BAA Number	DARPA-BAA-10-70
Proposal Title	SAFE: A Semantically Aware Foundation Environment for CRASH
Technical Area(s)	TA1: Processor Architectures TA2: Operating Systems TA3: Formal Methods TA4: Programming Languages and Environments
Lead Organization Submitting Proposal	BAE Systems
Type of Business	Other Large Business
Contractor’s Reference Number	A-10-0071
Other Team Members and Type of Business	University of Pennsylvania (Other Nonprofit) Northeastern University (Other Nonprofit) Harvard University (Other Nonprofit)

Technical Point of Contact Dr. Bryan Loyall 6 New England Executive Park Burlington, MA 01803 E-mail: bryan.loyall@baesystems.com Phone: (781) 262-4923 Fax: (781) 273-9345	Administrative Point of Contact Ms. Tracy Vachon 6 New England Executive Park Burlington, MA 01803 E-mail: tracy.vachon@baesystems.com Phone: (781) 262-4439 Fax: (781) 262-4123

BAE SYSTEMS

Penn

SAFE:
A **S**emantically **A**ware **F**oundation **E**nvironment for CRASH



Innovations

- *Fine-grained compartmentalization* supported by hardware, with runtime intents & security interlocks without compromising performance
  - Tagged data for compartmentalization and intent
  - Programmable Rulesets
  - Hardware Tag Management Unit for complete mediation on cycle-by-cycle basis
- *Co-design for pervasive verification* defines clean semantics and omit complicating features to make verification tractable
  - Formal analysis of static guarantees
  - Hardware Rulesets ground axioms for formal analysis
  - Complete verification of abstract machine grounding security
- *Prevention-in-Depth* radical decomposition of systems into mutually suspicious components with separated privileges.
  - Neighborhood watch cross-checking of components
  - Multi-level monitoring

Impact

- Game changing: radically transform the cyber landscape to give advantage to the defenders
- Secure software: easiest to write a secure application and hard to unintentionally introduce vulnerabilities
- Resiliency to attacks and defects: strong compartmentalization limits breach impact
- Minimal reduction in performance : hardware supported security model performs rich security checking in parallel
- Solid platform: provides base for future research in secure computing

Unique Aspects

- Co-implementation of continuously available SAFE prototype enhances co-design; ideas evaluated & issues resolved rapidly
- System-wide use-cases align research dependencies & minimize incidental development mismatches
- Leading researchers in all areas addressed by SAFE with many on team spanning at least two areas facilitating co-design
- Co-design enhanced by prior research collaboration among team members; will maximize benefits of co-design while avoiding risk
- Co-implementation risk reduced by BAE Systems proven track record of delivering in DARPA systems programs

Compartmentalization, pervasive verification and prevention-in-depth shift the advantage from attackers to defenders liberating them from today’s breach-and-patch-of-the-week grind.

1.1 Innovative Claims for the Proposed Research

We propose to create SAFE, a *Semantically Aware Foundation Environment* for CRASH. Our key claim is that it is feasible to build a suite of modern operating system services that embodies and supports fundamental security principles—including separation of privilege, least privilege, and mutual suspicion—down to its very bones, without compromising performance.

Achieving this goal demands a *co-design* methodology in which all critical system layers are designed together, with a ruthless insistence on simplicity, security, and verifiability at every level—leading us to an integrated effort focusing on the CRASH topic areas of (1) processor architectures, (2) operating systems, (3) formal methods, and (4) programming languages and compilers. To make SAFE feasible, we will push the state of the art in three mutually supporting areas: *fine-grained compartmentalization*, *pervasive verification*, and *prevention in depth*.

Fine-grained compartmentalization: With modern hardware, we can design a processor that enforces fine-grained compartmentalization on an operation-by-operation basis (*complete mediation*) without significantly impacting performance (§2.4.3). The processor will natively support access control between a (virtually) unbounded set of **principals**—the entities on whose behalf the processor is running—and a (virtually) unbounded set of **compartments**–fine grained sets of related objects (§2.4.2). The processor supports a single mechanism, **gates**, to transfer control from one principal to another, mediated by **rulesets** (§2.4.2). Programmable rulesets will define the compartment and instruction privileges available to each principal. The building blocks of principals, compartments, gates, and rulesets allow us to divide the computation into small pieces, each with the minimum necessary capabilities (*least privilege*) and to decompose tasks into federations of cooperating, but mutually suspicious, components (*separation of privileges*). We will further support compartmentalization with a segregated memory model, garbage collection, and native support for inter-process streams (§2.4.5).

This foundation will allow us to radically restructure both the operating system and large-scale software applications that use it (§2.4.7). This provides two key advantages: (1) *breach containment* and (2) *decomposable verification*. That is, strong compartmentalization guarantees that the compromise of a single piece of code or principal enables limited access to the rest of the system, containing the damage it can do, thereby protecting the integrity of cooperating components so they can detect and report the misbehaving component (see *Prevention in Depth*, below). Moreover, the guaranteed non-interaction between components allows us to verify components independently (see *Pervasive verification*).

Pervasive verification: While full verification of arbitrary software systems remains an elusive challenge, specification and verification technology has improved to the point where full verification of critical abstraction layers such as garbage collection and tag management is within reach, provided we carefully design our programming languages and hardware with verification in mind (§'s 2.4, 2.4.5). Moreover, the same design features that increase security and robustness throughout the system—strong compartmentalization and continuous dynamic checking of fundamental safety properties—will greatly simplify more targeted verification of key properties in the higher layers of the OS and in applications.

The compilers for our systems programming languages will be proven correct with respect to formal models of the source and target languages, allowing reasoning at the source code level with strong assurance that the verified claims actually apply to the running code.

At runtime, the hardware will maintain rich type and provenance data, and all machine-level operations will be checked against security policies that refer to principals, operations, and per-word metadata (§2.4.3). Key axioms necessary to make static analysis decomposable and tractable will be checked at runtime by hardware-supported metadata mediation. These include the assurance of *control-flow* and *object integrity*, ensuring that procedure calls and intra-procedure branches are checked against compiler-supplied targets; that there are no references beyond an object's extent; and that object metadata integrity is maintained.

Complete verification of the abstract machine layer guarantees that the foundational services that support compartmentalization cannot be violated. Verification of safety policies on higher level software guarantees that components will not attempt to violate their assigned privileges during execution.

Prevention in Depth ensures that, above the abstract-machine layer, there is no single mechanism that, if breached, will allow arbitrary violation of security policies. SAFE achieves prevention in depth using two organizing principles, *fine-grained task decomposition* and *mutual suspicion*.

Fine-grained task decomposition utilizes the features of fine-grained compartmentalization discussed above in order to guarantee that no single entity has the privileges necessary to compromise system integrity (*separation of privileges*). *Mutual suspicion* is the principle that, despite formal verification, behavioral correctness and security policies adherence is nonetheless checked at runtime. Decomposition patterns like *proposer-verifier-implementer* provide design guidelines for organizing software with separation of privileges and mutual suspicion (§2.4.5). Analysis and decomposition support in the system programming languages will ease the task of creating segregated software and provide feedback on the quality of decomposition (§2.4.6). SAFE's programming languages will also support automatic generation and user specification of monitors to verify that components are behaving as expected.

SAFE will further enable the creation of *non-bypassable forensic trails* available for logging and runtime safety and correctness monitoring (§2.4.7). The log implementation itself will be verified for correctness.

Combined, these mechanisms minimize attack surface, offer multiple levels of security, and support breach containment and detection when penetration does occur. Furthermore, they provide a rich substrate for other CRASH contractors to experiment with approaches to diversification and sophisticated detection, diagnosis, and repair for adaptive immunity.

SAFE's team of multidisciplinary experts provides unique capabilities and synergies to achieve these breakthroughs. Our team includes world-class researchers in all four SAFE research areas, with most team members working in at least two areas, facilitating co-design. BAE Systems team members bring expertise across these areas, combined with experience in agile development of integrated research systems from numerous successful DARPA research programs. SAFE builds on strong working relationships, including nearly a year of work by Loyall, Sullivan, Knight, DeHon, Rosenberg and Anderson on the SAFE design, which advances over 4 years of prior work in security tagged architectures by Knight and DeHon [SDK09]. The team is committed to a truly clean-slate approach, vs. advancing existing research agendas, with a shared recognition that this is a once-in-a-career opportunity to reinvent computing.

Extreme co-implementation addresses the challenges inherent in constructing a new compute stack from scratch in which normal assumptions about modularity, interface stability, and layer boundaries may not apply, and individual design decisions can only be effectively evaluated by system tests. To align research interdependencies and avoid accidently conflicting implementation details, we will rapidly stand up an end-to-end experimental system guided by a jointly-designed series of use cases that focus the research from an initial thin line through the system to a fully functioning SAFE (§2.4.8).

1.2 Technical Approach

Problem: Securing large software systems is beyond the current state-of-the-art. With millions of lines of code, all of which may interact, these systems are too large for monolithic verification. Unfortunately, the mechanisms and capabilities of conventional processors, and particularly their cost structures, demand that we treat large software systems as a single, monolithic protection domain inside which arbitrary interactions are permitted. The result is well known: any breach compromises the integrity of the entire system. This provides a huge asymmetric advantage to the attackers—they only need to find one bug, while the defenders must secure millions of lines of code. This gives rise to today's breach-and-patch-of-the-week environment.

Strategy: Our strategy is to accept that we cannot produce flawless software on this scale and must instead design software systems that are more amenable to automated detection and avoidance of flaws while being more tolerant of the inevitable flaws that will exist. To do this, we must divide software into more manageable pieces and constrain their interaction—we must compartmentalize. Doing so makes verification of individual components possible. Another advantage of strong compartmentalization is that it contains the effects of breaches, preventing faulty software components from terrorizing their neighbors. Once we break free of the one-breach-violates-system-integrity model, software components can become suspicious like a systems software *neighborhood watch*. This makes it possible to design systems that detect misbehavior, prevent its propagation, and even replace faulty components.

Safety First: In today's legacy systems, functionality and performance always trump safety, reliability, and security. As a result, the programmer and the hardware must perform more work (write more code, run more instructions) to implement safe and reliable solutions. In contrast, SAFE is founded on a *safety first* principle that guides the design of mechanisms and services, the selection of implementations, and the design of programming language semantics and compilers: *Make the robust case simple to program and fast to execute.* Safety as default does not mean that we ignore performance; rather, it means that the common cases that must be supported most efficiently by hardware are the ones that ensure safety and reliability.

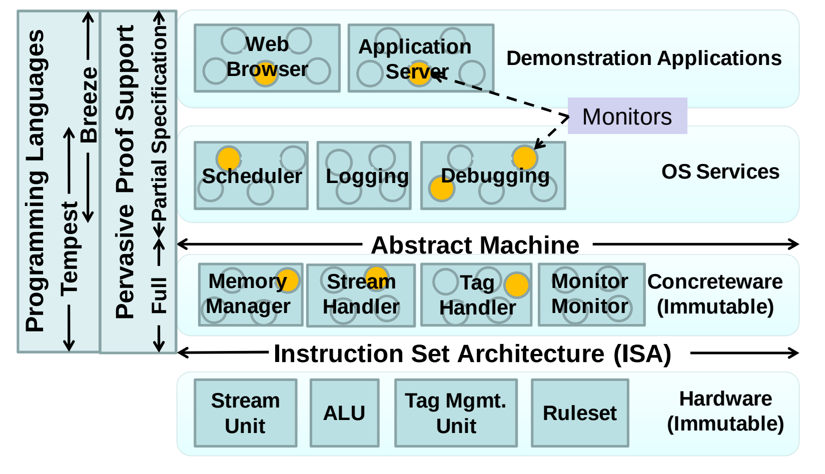


Figure 1 - SAFE Architecture

Co-design: Engineering co-design allows us to *define away* hard and intractable problems. The interfaces between system layers (*e.g.*, ISA, OS) now in use (*e.g.*, x86, shared memory, user/kernel mode) were designed decades ago without concern for security in a networked world or formal verification. Consequently, they exacerbate the challenge of providing safety with reasonable performance and ease of programming. With decades better understanding of algorithms, verification, system design, programming languages, application requirements, and programmer difficulties, we are now in a position to re-engineer the boundaries between layers to vastly simplify the task of analyzing and securing computer systems. On top of this greater understanding, decades of Moore's Law scaling has changed the tradeoffs in hardware costs, making it feasible to demand richer assistance from the hardware so that safety support need not be detrimental to performance.

SAFE Architecture: The SAFE architecture (Figure 1) defines clean interfaces between hardware, OS and applications that allow formal verification of all components, including abstract machine services, compilers and critical properties of applications.

Security begins with every word in the system. Each word has a *tag* encoding runtime metadata for its type, provenance, and other properties (§2.4.2). These tags are supported by a hardware *tag management unit (TMU)*, a software *tag handler* to support unlimited numbers of tags, and a programmable *ruleset* that allows or disallows each machine instruction, computing resultant data tags (supporting provenance). Hardware support enables fine-grained privilege separation without sacrificing performance (§2.4.3). Developers can match separation and protection to the task rather than being burdened with coarse-grained and low-performance isolation mechanisms (§2.4.2). Immutable, fully verified *concreteware* provides additional security for the abstract machine.

SAFE's high-level programming language *Breeze* has a type system that allows specification of formal safety and security policies that are statically verified (formally proven correct) *and* dynamically enforced by the underlying hardware. Proof obligations and redundant checking of policies are enforced by the TMU and by behavioral invariants compiled into separately executing monitors. A *monitor monitor* uses separately provided information from the compiler to check that the monitors are not compromised. *Tempest* allows more detailed control, with less automatic support for verification. Breeze also facilitates automatic partitioning of services and applications into interacting components and accompanying monitors (depicted notionally as sets of circles in the figure) for separation of privilege, least privilege and mutual suspicion. SAFE's *Stream Unit (SU)* and *Stream Handler* provide lightweight communication between processes to further support this modularity, making communication between principals cheap while supporting lightweight communication between processes running on different cores. Secure OS services including a least-privilege *scheduler, debugger, memory manager* and non-bypassable *logging* are implemented without a privileged kernel.

1.2.1 How SAFE provides Prevention-in-Depth for Security and Resilience

A brief example will illustrate SAFE's operation through development and execution. Standard web browsers support the integration of plugins—third-party software modules that operate with full access to the browser's address space. We assume that the browser is well-written, but plugins may be buggy or even malicious. Assume an adversary who either wants to exploit flaws in an existing plugin or get users to use a malicious plugin of his own.

Vulnerabilities in a plugin, or in the browser, might allow the adversary to take it over. Techniques such as "fuzzing" will reveal many bugs, each a possible attack vector: buffer overflows, arithmetic errors, etc. The plugin (or its interpreter) can corrupt memory by overwriting return addresses, changing function pointers, etc. If any of the bugs allows the attacker to run his own code in the browser, he can control it. SAFE blocks these attacks at the lowest levels of the system, in the ISA, Abstract Machine, and operating system. Writes outside array bounds are disallowed, return addresses are checked for validity, function pointers are not writable. Attempts to use such flaws will be trapped by the hardware and logged, bringing the attack to administrators' attention.

Consider instead a Trojan plugin, designed to collect and upload users' account names and passwords, attractive enough for users to think they want it. Today, the plugin has access to anything in the browser's address space, either directly or through its access to the document object model (DOM) supported by the browser. On SAFE, the browser is written in Breeze, which automates the partitioning of the process into distinct privilege domains. Each plugin is in a separate compartment, executing on behalf of a separate principal (§2.4.2), so it is effectively sandboxed by the hardware and operating system, not by the browser. Its only access either to browser memory or to operating system functions is through the browser's plugin API.

The plugin will require access to the DOM, including user passwords. Using compartments, the memory associated with a password string can be made unreadable outside the browser core, while still allowing plugins to get password references and pass them back to the browser. If the plugin can create a form, fill it in, and submit it—its only way to access the network—the browser core can create the DOM in the plugin's compartment, and let code execute against it only on behalf of the plugin's principal. The user's confidential information can be accessed and used by the plugin without its having any ability to read it or upload it to the network.

1.2.2 Architecture Model Brings Fine-Grained Security Concepts to Runtime

The SAFE tagged architecture is based on our previous work under the TIARA project [SDK09]. In this fine-grained protection model, *complete mediation* means allowable instructions and data access control are performed for each instruction using metadata on each word of memory.

The privileges of the entity on whose behalf a process runs are designated its *principal.* A process executes with *least privilege,* meaning its sequence of principals over its execution only have the rights they need at a given moment. Concretely, principals are unique identifiers that are given meaning through the installed *ruleset*. In a conventional operating system, each user and each service (*e.g.* network file server, web server) is a principal whereas in SAFE we use finer-grained principals and hence privilege assignment. Rather than a single user principal, the user would have sub-principals for each application or service (*e.g.* user.browser, user.mailreader, user.editor, user.fileserver), and these can further be subdivided (*e.g.* user.browser.jpeg-render) so task and service principals run with minimum privileges.

Compartments are an abstraction for collections of memory that are treated as a group with regard to access rules. Compartments can be as *fine-grained* as a single word. Compartments can be used to represent private data, data with limited sharing (*e.g.*, one principal can write and another can read), and data with restricted uses (*e.g.*, only this principal can execute). Compartments capture the process isolation and shared-memory data sharing of traditional inter-process address-space separation and mapping, but at a finer granularity and with lower overhead when switching among code and principals with different access rights.

Every word in the system has an associated *type*. Types are an abstraction capturing broad classes of data that can also be used to constrain appropriate use. Types differentiate words in memory that are used for code, addresses, integers, and floating-point values. Types further distinguish particular roles for data (*e.g.* object length, frame pointer, return address pointer).

A *tag* is a set of bits encoding the metadata associated with each word, including its compartment, its data type, and additional metadata properties from an extensible set specified by the SAFE services and applications. The metadata are given operational meaning through a *ruleset*. Rules are evaluated in parallel with computation, having minimal impact on performance; they allow or disallow each instruction based on the current principal, the instruction to be executed, and the tags of the operands. If the ruleset allows an operation, a new tag is computed for the data result. This general mechanism supports multiple forms of provenance propagation and enforcement (see §2.4.4). The Program Counter (PC) is also tagged, allowing the PC to encode the provenance of any data that was used in conditional branch instructions.

A rich, extensible set of principals, metadata categories, and rules enables compact expression of a wide range of protections in SAFE. E.g., rules can express the intended use of data (*e.g.* pointers, code, integers, return addresses), allowing SAFE to restrict the processor to only use data in meaningful ways (*e.g.* not treat an integer as a pointer or instruction).

SAFE uses a *segregated memory model*, meaning all objects exist as memory *frames* with fixed and manifest bounds. References outside the bounds of a frame are defined as errors. The model does not rely on *mutable shared memory* between processes. This segregated memory model eases verification since independent modules do not need to reason about aliasing or arbitrary mutation of data by other system components. It is also consistent with the distributed-memory models that will be necessary for large-scale, multiprocessor systems.

We provide two models for inter-Principal communication: (1) gates within a process and (2) interprocess streams. When a process needs something done for which it lacks privilege, it makes a principal-changing-procedure-call, a *gate*, and the process principal changes for the duration of the procedure call. This is appropriate when the process does not have useful work it can perform in parallel with the privileged operation. Alternately, a process can make a request to a service process over an interprocess stream. In this case, the service process executes concurrently as its own principal serving requests from many different processes.

1.2.3 CPU Organized for Reliable Execution (CORE)

Challenge: Conventional processors lack precisely specified and cleanly defined Instruction Set Architectures (ISA) that can be used for formal verification. Furthermore, the hardware does not have access to the semantic information to understand the operations it is performing, allowing it to be easily tricked into violating the intended semantics of the computation (e.g. converting integers into pointers or instructions). These processors have only a coarse notion of principals with different privileges—usually just two just two—forcing the systems layer to implement privilege division and separation in software. The large cost of changing between principals with different access rights discourages the kind of fine-grained compartmentalization necessary to support least privilege, separation of privileges, and modularity – complicating verification and discouraging safe and robust operating system design and software architectures.

Strategy: Our clean-slate strategy redesigns the processor architecture to support verification and safe operating system and software architecture design without sacrificing performance. We bring the notion of principals, types, compartments, and streams down to the hardware and support principal-changing gates.

ISA: The CORE ISA uses register-to-register, load-store, RISC-style computational instructions. It has the standard arithmetic, logic, load-store, and branching instructions and a conventional register set. It adds special processor state registers for principal, code frame base and bounds, and data frame base and bounds. All memory references and intra-procedure control flow are made relative to the base of frames and are bounds checked. Processor state captures the source of procedure calls, returns, and branches so that control flow from unintended locations can be identified and disallowed. The CORE ISA adds procedure call, entry, and return instructions that handle the definition and exchange of processor state (e.g. code and data frame bounds) including the potential change of principal. The CORE ISA includes streaming operations. Special instructions exist for data tagging, tag extraction, and TMU maintenance. These instructions are carefully mediated by tag rulesets as noted below. All instructions are tagged and all code lives in code blocks. The CORE ISA is formally defined with a mechanized semantics using the Coq proof assistant. This will help connect the CORE ISA to the verified compilers such as Tempest, the low level systems language (§2.4.6).

Table 1 - Rules Supported by CORE ISA Tag Rule Processing

Result	Query
boolean	canRead(Principal p, Compart c), canWrite(Principal p, Compart orig, Compart new_word), canUse(Principal p, ExtMetadata e), allowedInstrType(Principal p, Type type), allowedOpTypes(Principal p, Instr i, Type t1, Type t2)
Compart	writeCompart(Compart orig, Compart new_word), combineCompart(Compart c1, Compart c2)
ExtMetadata	combineMetadata(ExtMetadata e1, ExtMetadata e2)
Type	typeResult(Instr i, Type t1, Type t2)
Handler	monitor(Type t), monitor(ExtMetadata e)

Tag Discipline and Rules: On every operation (*complete mediation*), the instruction, data tags, and principal are checked for permissions (i.e., the Boolean queries in Table 1. If allowed, the data tag on the result, including the data type, compartment, and extended metadata fields, are determined based on the tag of the inputs (i.e., combineCompart, typeResult, combineMetadata). For example, when P.user.webreader performs an ADD on (network-data-for-user, integer, network-tainted, 32) and (web-local-data, integer, localdata, 1), the tagged result might become (web-local-data, integer, network-tainted, 33), assuming P.user.webreader is allowed to read network-data-for-user and web-local-data and the ruleset specifies propagating the network-tainted indication into all derived results. If any aforementioned privileges were denied, a rule violation would be flagged. A similar computation is performed for writes based on the tag of the data to be written and the tag of the data at the target memory location (i.e. writeCompart in Table 1).

Tags are immutable and not accessible to normal code. The compartment and extended metadata fields define whether or not the principal is allowed to read or write the data. The type tag further defines how the data can be used by the principal (e.g. pointers, integers, and floating-point), if at all. For example, if P.user.webreader now attempts to perform a procedure call to (web-local-data, integer, network-tainted, 33), the ruleset would flag a rule violation since integer is not a valid type for a method all. It can also be used to distinguish and limit the use of data with special rules such as bounds, return addresses, and garbage-collection forwarding pointers. One significant consequence of this is that we can limit the use of instructions, and even limit instructions on particular data types by principal, in a fine grained way (*separation of privileges*). For example, the privileged instruction for TMU-write could be given to the TMU.refill.verifier principal without giving it any privileges to retag data.

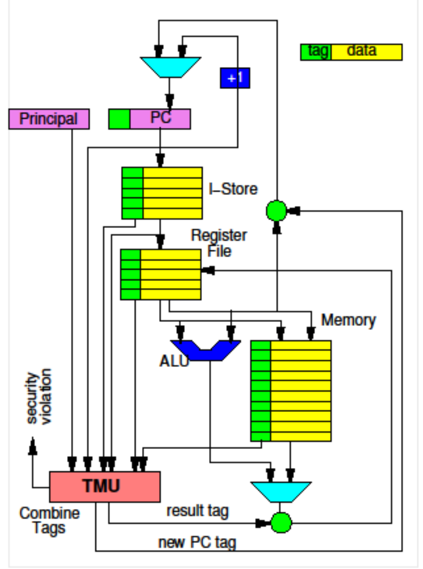


Figure 2 - Processor Datapath with Tags and TMU

Hardware Microarchitecture: To support these operations efficiently, the hardware includes a **Tag Management Unit (TMU)** and a **Stream Unit (SU)**. The TMU examines the tags on the data, the instruction, and the program counter (PC) to determine the validity of the operation and the tag for the result. The TMU operates on bit-level encodings of the tags and principals, so simply performs a memory lookup to determine permissions and result tag encodings. The TMU operates in parallel with the processor's execution path, validating the results before writeback (See Figure 2). In this way, it does not impact processor cycle time and shares well developed mechanisms for miss-speculation when violations or misses occur. The TMU can be decomposed into a number of small associative or hashed memories to hold common-case tag combinations and backed by service routines to update the TMU hardware similar to TLB or page-miss handlers (§2.4.5). The TMU refill handler is part of the Abstract Machine (§2.4.5). Preliminary estimates suggest the TMU can be less than half the size of today's 64KB L1 data caches [SDK09].

The SU performs a translation from stream handle to the current runtime mapping for the stream [DMC+06], routing stream data to memory, physical FIFOs, physically configured wire channels, Direct Memory Access (DMA) channel, or over packet networks (e.g. [KMd+06]). With parallel hardware support, it can allow stream reads and writes to complete at the speed of an L1 data cache regardless of the destination. For multi-core implementations, this can provide low-latency, hardware-mediated communications. Our previous design of an SU was for a heterogeneous architecture that included both ISA processors and reconfigurable accelerators [DMC+06].

1.2.4 Tag Rules Provide a Mechanism for Data Tracking and Safety Interlocks

Tags with rulesets are a simple, powerful mechanism with a multitude of uses (*economy of mechanism*). We have identified several uses that we will exploit immediately. Furthermore, this capability opens up a rich new space of design options, and our research agenda includes exploring the new opportunities created.

Type Interlocks: As described above, type tags allow us to define the role that words play (e.g. code, return-address pointers, bounds, etc.) and enforce these roles on a per-instruction basis.

Principal Interaction Interlocks: Compartment tags with rules are effectively a form of hardware interlock on principal and code interactions. They guarantee that information flow is limited. These can be used to lock down the interaction between components (*modularity*). This allows strong **sandboxing** of code, guaranteeing that components only interact in carefully defined ways. It can thus be used to enforce the intended modularity and interfaces in a software architecture. The rule set that defines these interactions can be small compared to the code, allowing tractable validation of gross flow and interaction properties for a large piece of code.

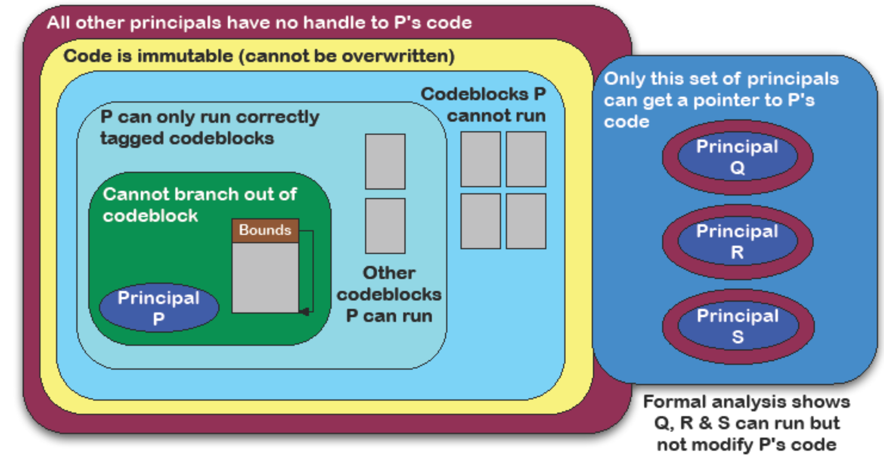

All other principals have no handle to P's code
Code is immutable (cannot be overwritten)
Codeblocks P cannot run
P can only run correctly tagged codeblocks
Cannot branch out of codeblock
Bounds
Principal P
Other codeblocks P can run
Only this set of principals can get a pointer to P's code
Principal Q
Principal R
Principal S
Formal analysis shows Q, R & S can run but not modify P's code

Figure 3 - Prevention-in-Depth Protection of Code using Tag Ruleset

The rules can be used to define multiple, independent barriers to compromising the integrity of the system (*prevention in depth*). For example, Figure 3 illustrates multiple barriers the rules can erect to prevent a principal from running unauthorized code.

Provenance Tracking: Tag combination can be used for a broad range of privacy, privilege, and integrity tracking and management. Privacy tracking and enforcement might include mandatory access control (e.g. proprietary+open→proprietary), more general security lattices, set-based privileges (e.g. JAVA), taint tracking (e.g. marking data originating from the network or touched by specific principals and preventing their use by unauthorized or inappropriate principals), and other dynamic information flow tracking and containment. Tag combination can also be used to track the integrity of data, including pedigree, confidence, and contributing sources. In general, any set of properties that can be defined as a lattice will have a well-defined composition operation and can be supported. Lattices with a small finite number of entries (e.g. 10–100) will be easily and efficiently supported with all they key combinations resident in the small hardware memories. Supporting larger lattices may require some innovation in both the hardware and TMU-miss handler implementations. Set-based privileges can also be viewed as lattices, but with large (potentially power set) lattice points. Dynamic allocation of short tag encodings to the set combinations seen in the working set should be sufficient to keep tag requirements small without impacting performance.

1.2.5 Abstract Machine Layer and Concreteware

Between the hardware ISA and applications, we include an abstract machine (AM) layer (Figure 1) that serves several important properties providing: (1) an unbounded abstraction for the hardware, (2) abstract operations where the hardware/software boundary may reasonably change from implementation to implementation, and (3) a trusted computing base.

Fine-Grained: We start from the assumption of fine-grained segregation, meaning an unbounded space of principals, types, and compartments, small compartments (as small as individual words), and fine-grained assignment of privilege (permissions can be granted to individual instructions on specific types of data). This basis allows us to broadly explore the correct granularity driven by the co-design of services, applications, languages, and verification. This is in contrast to prior work that took hardware limitations on granularity or perceived hardware costs as the starting point that constrained design (*e.g.* [Bel05, SBL89]). The combination of virtualization layers, modern hardware availability, and the broad success in microarchitecture design for inexpensively supporting virtualized resources give us confidence that we can support these rich spaces and make their ultimate hardware implementation cheap. If necessary, techniques such as those used in HardBound [DBMZ08] and pointer swizzling can allow each word in memory to have a unique compartment, although we do not expect to need that extreme. We expect most use of this flexibility will be via the compiler during code generation.

Unbounded: A physical machine implementation will have finite memory, a finite TMU capacity, finite number of tag bits, and finite hardware support for streams. The AM provides a machine view where these are unlimited. These demand handlers to support this abstraction, including garbage collection and allocation support for memory, TMU miss and refill handling for the hardware TMU cache, translation of compartments and types to the physical tag encodings, and allocation, buffer expansion, reclamation, and physical DMA management for streams. These handlers are implementation-specific code that serves an analogous role to microcode or PAL code. To protect the abstraction, components that attempt to allocate excessive amounts of resources (e.g., which try to use up the entire memory of the machine) are descheduled or killed.

Unbounded Memory, Garbage Collection: Memory management both increases programmer's burden and is a source of security bugs. Explicit memory management also complicates decomposed verification. We use garbage collection to eliminate this class of errors, support verification, and reduce the programming burden. We make sure the hardware supports garbage collection efficiently, and we provide security hardened versions at the AM layer.

To support both the needs of the application and the needs of the system, we support two levels of memory management. At the system level, we support *compartment-level* management. This allows the system to reclaim resources as principals exit the system and no longer need their data. Garbage collection at this level is provided by a small amount of concreteware (below). Here, garbage collection is coarse-grained (regions v. data elements) and the collector does not need access to read the data contained in these regions, only to reclaim it, avoiding the risk of security compromises through the compactor. Simple, robust algorithms manage regions, avoiding complex performance engineering. In the TIARA effort [SDK09], we showed how a proof-of-concept global garbage collector (a more complicated task) could be decomposed into mutual suspicious components to avoid single-point-of-failure vulnerabilities. *Within* a compartment, standard garbage collection technology, including application-specific garbage collection, can be used. This collector can employ techniques from the TIARA garbage collection to further decompose the privileges required by the collector. SAFE will provide a default intra-compartment garbage collector at the AM level, as well as hooks for application-level garbage collection.

TMU Miss Handler: On a TMU miss, the miss handler will consult the full ruleset and determine if the operation is allowed and the appropriate tag results for the operation. Determining the appropriate tag may include computing the least-upper-bound meet on a lattice, computing set unions, and determining membership in aggregate groups. If the resulting tag does not have a short encoding, either because it has not been encountered, yet, or had to be recycled due to the limited space of physical encodings, the miss handler requests a new encoding from the tag manager. If allowed, the appropriate entries are installed in the TMU, displacing current entries as necessary. If the operation is not allowed, the handler passes control to the appropriate handler for the violation.

Concreteware: These handlers play essential roles in maintaining the semantic abstraction and root of trust for the system. As such, they demand special scrutiny and care to assure that they do not constitute a single-point of failure or weak link in the system. We call these *concreteware* to emphasize that while this layer may be instructions running on top of the processor, it is hardened code that is not free to change in an operational system. To harden the code, we employ a combination of: (1) immutable code, (2) full (not partial) verification on these tasks, and (3) separation of privilege decomposition into co-operating, but suspicious principals using patterns such as *distributed invariants* [CW87], *the satellite pattern* [SDK09], and *proposer-verifier-implementer*.

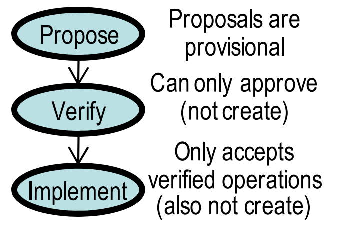

Propose
Proposals are provisional
Verify
Can only approve (not create)
Implement
Only accepts verified operations (also not create)

Figure 4 – Proposer-Verifier-Implementor

To illustrate these patterns, in proposer-verifier-implementer (Figure 4) we separate a critical task into one entity that is only privileged to propose an action (proposer) and a second that has the critical privilege of verifying and validating the action (verifier), but does not have enough privileges to create its own actions. The verifier's only choice is to accept or reject an action proposed by the proposer. In this way, both must be compromised to perform some intended operation. In the TMU refill, the implementer would be the actual hardware which will only take commands from the verifier. The proposer suggests the TMU operations, and the verifier validates they are consistent with the policies before passing them along to the TMU (implementer). This pattern is also used in cases where data is retagged, including the creation of tags from data. Unlimited retagging could be a single point of failure, so fine-grained privilege assignment allows us to separate who can perform what kinds of retagging. In these cases, the proposer can retag to a provisional tag and the verifier can upgrade the provisional tag to a real tag, but can only perform the upgrade operation. The verifier cannot create tags on its own.

SAFE also includes a *monitor monitor*, implemented in concreteware, that uses separately stored registration information provided by the compiler to confirm that the proper configuration of monitors is always executing in SAFE. This addresses attack vectors such as one where an intruder first breaches one or more monitors before attacking the components.

1.2.6 Systems Programming: Tempest and Breeze

Two systems programming languages, *Tempest* and *Breeze*—together with their compilers— are key elements of SAFE. Tempest, a variant of C designed for formal analysis, plays two different roles: (1) it is used as an intermediate language for compiling Breeze, in which the bulk of SAFE is written, and (2) it is the implementation language for a small amount of low-level "bit twiddling" code that needs direct access to hardware resources.

Breeze provides higher-level abstractions than Tempest (e.g., closures), hides low-level machine details (e.g., memory allocation, memory layout, tag management), and automatically enforces security policies with its type system, making it more convenient and productive than Tempest. This two level architecture ensures that we can write and reason about most code at a high-level of abstraction (Breeze), but where necessary, punch through these abstractions at the cost of constructing explicit proofs (Tempest). Both languages will have formally defined operational semantics to support verification both of the compilers and of applications written in the languages.

Tempest: Low-Level Programming with Explicit Proofs. The Tempest design is based on Leroy's C-light [Ler06], allowing re-use Leroy's CompCert verified optimizer for C-light, by porting its verified code-generator to our ISA accelerating our SAFE research and initial end-to-end SAFE prototype, and providing reasonable performance for our compiled code.

To extend to a full Tempest, we must extend the formal semantics with the new abstractions provided by the SAFE ISA (e.g., principals, tags, compartments, etc.), and extend to require a proof that universal security properties (e.g., object-level integrity) are respected.

Breeze: A High-Level Systems Programming Language. Most of our systems-level services will be written in Breeze, a *value-oriented*, mostly functional language, with an emphasis on co-design for verification. (As a general-purpose, high-level language, Breeze will also be useful for application-level programming, as a complement to more specialized application-level programming languages from other CRASH teams.) Its design will be optimized to facilitate simple, modular proofs, drawing in part on functional language concepts.

The SAFE architecture emphasizes multiple, independent compartments of memory, with hardware support allowing computations to shift among its various compartments in a lightweight way. Accordingly, Breeze will include linguistic mechanisms for specifying a program's interaction with its various compartments—a form of modularity that, in keeping with our prevention-in-depth approach, can be checked at three distinct times: (1) at compile time, (2) with compiler-inserted checks at run-time, and (3) with hardware-based checks as a backstop.

Breeze, like ML, Haskell, and Typed Scheme, will feature an expressive static type system, ensuring that many errors will be caught during compilation. We will incorporate three significant advances over conventional type systems to directly address security policies and/or simplify reasoning: (1) We will incorporate a *type-and-effects system* (in the spirit of FX [JG89]) so that we can make a clear distinction between functions that may have side effects and those that are pure; our libraries will be designed to encourage "purity" by default. (2) We will incorporate an *information-flow type system*, following on the ideas of Myer's Jif system [SM03] and the

Flow Caml system [Sim03]. Unlike those, we will track information-flow dependencies at *both* compile-time and run-time. The research challenge is finding a balanced, effective approach that leverages the strengths of both techniques. (3) We will incorporate simple forms of *refinement* and *session types*. Refinement types (in the style of Sage [GKT+06]) take the form {x : T | P (x)} where P is a boolean predicate, written in Breeze, over the base type T . This allows very precise behavioral contracts to be declared statically and either checked at run time or statically and removed from the compiled binary.

Breeze will include support to automatically generate monitors for components from high-level descriptions of correct behavior, invariants and policies, as well as for unaddressed proof obligations and axioms. Monitors run with privileges no greater than the components with which they are associated. Developers can also author monitors to check desired application/service-level properties.

Automating Least-Privilege Configuration: We recognize a danger that fine-grained, least-privilege protection domains lead to excessive system complexity. Without support for automatically decomposing programs, a developer will naturally tend towards an architecture with a handful of relatively large-grained protection domains. We have four strategies for mitigating this problem. First, we can take advantage of declarative linguistic constructs to specify *protection contours* at the source language level. The compiler can use this information on contours to automatically partition code into separate privilege domains similar to the approach used in Jif-Split [CLM+07]. Second, we can develop program analysis techniques, both static and dynamic, for determining the *privilege footprint* of contours (i.e., what is the maximal set of privileges that each component can achieve). Third, we can develop program analysis techniques for assessing the *Byzantine breach tolerance* of decomposed subsystems. The *Byzantine breach tolerance* is the maximum number of components an adversary can select for subversion without achieving the compromise goal. And fourth, we can provide libraries that embody common patterns of decomposition, such as *proposer-verifier-implementer* (§2.4.5), in an easy-to-use form.

1.2.7 Operating System Services

The SAFE Operating System (SOS) embodies extreme *de-kernelization*. Several benefits ensue. First, *protected state information is decentralized*, since the hardware protects state by dividing it into separate compartments and allowing or denying access based on fine-grained policies. This decentralization forces either an improbable chain of successes by an attacker, or an improbable series of internal failures before the overall system fails. Second, *fine-grained enforcement enables support for least privilege*. Third, a modular structure supports *redundancy* and *diversity*.

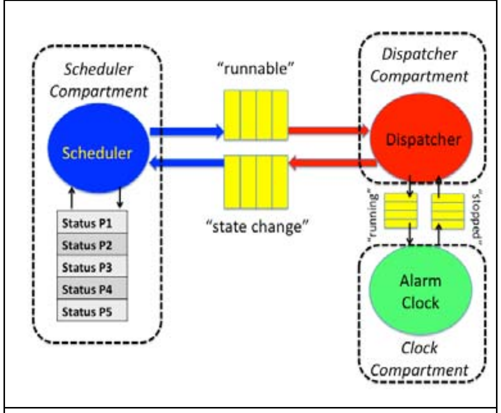


Figure 5 – Dekernelization into least privilege components isolates impact of any component breech

As an example of decomposition, consider the management of multiple competing user processes by a CPU scheduler (Figure 5). In outline, the scheduler maintains a list of processes, their scheduling status (blocked, runnable, running, *etc*) and scheduling-relevant resource consumption statistics (*e.g.*, CPU time used). The scheduler does *not* need access to process memory, inter-process communication (IPC) queues, or logs. A list of runnable processes can be sent via IPC to a dispatcher that places processes onto an available CPU. The dispatcher notifies the scheduler of state changes. The dispatcher only needs access to process state necessary to associate a runnable process with a processor, and may in turn call another "alarm clock" process with an identification of the now-running process so that it can be interrupted. Mutual suspicion amongst these elements is implemented with logic derived from valid state transitions and a reference clock. For example, if the scheduler does not see a transition from runnable to running, it might suspect the dispatcher, while if the transition from running to runnable does not occur, the alarm clock might be suspect. Fine-grained privilege assignment means we can give parts of the scheduler the ability to condemn and reclaim memory (e.g., to cleanup after process termination) without giving it the ability to read or write the memory.

IPC is performed with typed messages, ideally passed from sender to receiver through streams (§2.4.3). Two new technical ideas that become possible with typed message queues and gates are *least-privilege logging* and *least-privilege debugging*. Least-privilege logging is done by interposition, essentially a wiretap on the sending side of a typed message queue which is communicated to an I/O activity. This interposition structure enables the creation of non-bypassable forensic trails, valuable both for logging, rollback and recovery, and for runtime safety and correctness monitoring.

Least-privilege debugging provides inspection and intercession capabilities consistent with the principal executing an application. If a user has the capability of editing application source code, recompiling, and re-starting, then complete intercession (dynamic modification of a running application) is possible via a debugging interface. On the other hand, if the current user is not able to edit and recompile, then the debugging interface provides straightforward introspection via the logging interface described above.

Typed message queues can also be used to implement blocking, wakeup, and periodic alarms. This eliminates unscheduled device interrupts (except for that of the reference clock), easing verification and greatly simplifying coding device drivers (typical UNIX driver code is about 50% interrupt management).

SAFE's rich metadata tags are only meaningful when they are respected, i.e., their integrity is preserved. Inside a SAFE system, this can be guaranteed by the hardware and software support mechanisms described here. Outside a SAFE system, privacy of data and integrity of tags must be preserved. All I/O to and from tagged storage, including to secondary storage such as disk drives and perhaps even off-chip memory (e.g., DRAM), is performed through a gateway process that encrypts (privacy) and cryptographically hashes (integrity) tagged data on the way out, and decrypts and validates it on the way in. During development of SAFE, I/O will be serviced with a commodity processor and conventional OS acting as an I/O processor.

1.2.8 A novel approach to co-implementation guides co-design and integration

Construction of a CRASH prototype is a considerable challenge because it realizes a complete compute stack with component parts having numerous co-dependencies that evolve simultaneously. Our agile co-design/co-implementation cycle addresses this challenge with *continuous system builds* focused by a series of use cases of increasing sophistication (Table 3). We will rapidly build an initial end-to-end system leveraging baseline technologies and a rapid prototyping approach (§2.9.1). We will continually integrate research results into this evolving system as they are developed, maintaining a functioning end-to-end system at all times. Each SAFE prototype application will be associated with a suite of system tests that can be exercised in deterministic ways and measure progress toward a mature CRASH system.

1.2.9 SAFE is designed for transition

The security of computing and cyber-physical systems is considered one of our nation's greatest vulnerabilities. SAFE aims to virtually eliminate all known host-based security vulnerabilities, and SAFE technology will apply to a wide range of platforms—embedded to supercomputer, single processor to multi-core. SAFE subsumes the encapsulation provided by virtualization, providing sandboxing capabilities for encapsulating legacy codes or hardware emulations (§2.4.4). Partial results in SAFE, e.g. Breeze supported static verification with dynamic monitors for current processors, or hardware-enabled, limited systems in embedded FPGA-based systems, could be applied early to targeted needs of the warfighter. BAE Systems' deep knowledge of security concerns for military systems will inform our design of SAFE, and our leadership role in a number of these may offer targets of opportunity for early adoption of SAFE technologies. For example, we maintain all the computer hardware and software for the Compass Call Airborne Electronic Warfare platform, which employs an extensive mix of general-purpose and embedded hardware and is considered a prime target for cyber attack. Mechanisms being developed on SAFE could substantially improve the security of these systems.

1.2.10 How SAFE Prevents/Mitigates Current Attack Vectors

The SAFE architecture prevents the vulnerabilities caused by software defects that are the basis for many of the exploits on computer systems today, and prevents or mitigates human factors attacks, where the user is lured into installing undesirable behavior into their system. Table 2 gives an overview of seven classes of attacks, including particular instances, and SAFE's methods for preventing or mitigating them.

Table 2 – How SAFE addresses major classes of attacks

Prevented attack vectors	
Memory/stack corruption	Examples: buffer overflow, array index out of bounds, printf format attack. SAFE: provably correct system services, pointer-less AM, separation of code and data, and processor state, instructions, that limit branching control flow and procedure calls to be consistent with compiled program.
Heap/semantics faults	Examples: null pointer dereference, use-after-free, double-free, negative allocation. SAFE: HW/ISA memory access semantics, AM garbage collection, provably correct code, security violation traps, secure atomic memory allocation routines
Cross-thread semantic faults	Examples: race conditions, live-/deadlock. SAFE: no shared mutable memory between concurrent threads.
Black box checker errors	Example: Unicode parsing bug. SAFE: access control provides fine-grained and flexible limits, eliminating the need for black box security checkers.
Execution redirection	Examples: return-to-libc, heap spray, function pointer modification. SAFE: branches are limited to be within code segments and must be from allowed branch sources, high-level abstraction layer and tag ruleset precludes pointer arithmetic and overwrite, separation of code and data.
Mitigated Attack Vectors	
Dynamic behavior introduction	Examples: cross-site scripting, sandbox escape, SQL injection. SAFE: access control based on execution thread provenance.
Human factors attacks	Examples: spam, Trojan horse. SAFE: least privileges in a root-less, kernel-less system, safe installer verifies application conforms to accompanying behavior proof, full trace of execution.

1.3 Prior Work

We base much of the SAFE design on our earlier work in the Trust-Management, Intrusion-Tolerance, Accountability and Reconstitution Architecture (TIARA) [SDK09]. TIARA focused on applying best-of-breed technical solutions to a new hardware software architecture. In contrast to x86 (but similar in notion to the i432), TIARA's Security Tagged Architecture (STA) provides extremely fine-grained access enforcement, while the Zero-Kernel Operating System (ZKOS) exploits STA in a privilege-separation architecture intended to be least-privilege. A principal contribution of TIARA is its demonstration that hardware has evolved more than sufficiently to support advanced abstractions with adequate or superior system performance. SAFE will extend the TIARA technical study to the creation of a full system design extending the initial designs in TIARA's core areas of STA, ZKOS, with extension of co-design to include the programming language and formal verification components and additional facilities for resilience such as prevention-in-depth.

Prof. Smith has extensive experience at the hardware/software boundary. His group prototyped a highly parallel network adapter and OS support for novel queue management and "clocked interrupt" event signaling [ST93]. The SwitchWare active networking project provided safe general-purpose computing capabilities for routers. SwitchWare [AAA+90] used programming language techniques such as module-thinning in the ALIEN active loader; developed techniques for dynamic software updates; and developed resource-bounded programming languages, notably SNAP. SwitchWare also gave rise to EROS [SSF99], the Extremely Reliable Operating System. EROS demonstrated conclusively that a carefully constructed capability system need not suffer in IPC performance, while providing the least-privilege advantages inherent in capability systems. EROS is also notable in that it served as the basis for a proof of confinement [SW00]. Smith also led the development of Cognitive Networking at DARPA, with the SAPIENT [Smi09] and ACERT programs. The SAPIENT program's dynamic protocol composition approach represents a powerful approach to adaptive software systems.

Professor DeHon brings deep experience in co-design from physical substrate to algorithm implementation with emphasis on automatically mapping designs to the architecture. As a member of the TIARA team he led the development of Security Tagged Architecture, including the ISA, rule set, and an initial design of the hardware implementation. He has been recently involved in the development of techniques for mapping reliable and predictable computations to highly defective (10% of the elements unusable) technologies and high variation technologies (sigma=38%). He has been deeply involved in addressing problems across the abstraction stack from device physics to programming languages and applications, and in developing or adapting the automation necessary to fill the gap between high level expressions of computations and their efficient realization in hardware.

Professor Pierce has an established track record in combining theoretical foundations with practical engineering and implementation. Under the Manifest Security (NSF) project he is applying tools of language-based security and authorization logics to the development of new methods for securing "open platforms" such as web browsers. This technology is directly applicable to SAFE; in browsers scripts from untrusted sources must be run in a way that permits limited, well-defined interactions but the results of malicious scripts are strongly compartmentalized. Under the Network Opposing Botnets (NoBot) project (sponsored by ONR) he is developing techniques for detecting and mitigating botnet activity in a host computer using a combination of technologies based on programming languages and networking. Under the Contracts for Precise

Tyes NSF project he is investigating the use of software contracts for enforcing precise program invariants using dynamic checks instead of or in conjunction with static type analysis.

Professor Shivers has done research in the design and implementation of programming languages, analysis techniques for higher-order functional languages, and the interaction of programming languages with the operating system. Under the DARPA funded Express project, Dr. Shivers showed how to run an advanced functional language (Standard ML) on "bare hardware" so that the programming language becomes the operating system. His pioneering work on higher-order flow analysis is directly related to the design of SAFE and the goals of the CRASH program as it is one of the most important techniques for compile time analysis of the behavior of computer programs.

Tom Knight has a long history of building advanced, reliable computing systems. In the mid '60s, he developed the ITS time shared operating system for the PDP-10, which eventually achieved year-long continuous operation on the relatively new and unreliable hardware of the day. His MIT master's thesis on the hardware design of the Lisp Machine, allowed construction of one of the most successful dynamically type-checked and garbage collected programming environments. His subsequent founding of Symbolics Inc., led to further implementations of type checked machines including the 3600 and Ivory microprocessor. He developed aggressive parallel machines, such as the MIT Connection Machine, the Cross-Omega machine, and invented the idea of transactional memory, solving difficult problems in sharing data between parallel execution threads. His development of novel approaches for reconfigurable tag handling led to his recent work on TIARA, a proposed fine grained protection and reliability architecture.

Dr. Rosenberg has a wealth of commercial experience in massively parallel computer design, programming environments and tools, information security, parallel processor architectures and associated programming models.

Morrisett developed Typed Assembly Language (TAL) and the technology needed to compile programs to assembly code, and still prove that the resulting code is type-safe [MCG$^+$99, MWCG99]. TAL has had a strong influence on industry, particularly at Microsoft where their Singularity Kernel and Spec# programming environment use the ideas behind type-preserving compilation and typed assembly. TAL was also used by many research projects, including the PLAN work at Penn.

Morrisett also developed the Cyclone Safe C language which has been used in a range of research projects, from hardware code generation tools at Cornell, to device drivers in the Minix-3 kernel. Cyclone provides a strong type-safety guarantee, while also providing programmer-controlled access to low-level issues such as data representation and memory management.

Most recently, Morrisett's group built (a) a verified compiler for Core ML and (b) a verified incore database management system. They developed new extensions to the Coq proof assistant to support writing and reasoning about imperative programs in a modular, and scalable fashion.

Under the PittSFIield project Morrisett's group developed the software-fault isolation (SFI) technology now used in Google's Native Client. SFI provides a basic security guarantee for legacy code, but until this work, had proven too expensive to use in CISC-based systems such as the x86. This work received a "best-paper" award in the 2006 Usenix Security Symposium.

1.4 Comparison with Current Technology

UNIX-like systems [Rit74] are implemented primarily in the C programming language, which is inherently unsafe but portable across hardware. C's portability led to widespread use, but ruled out co-design for building strong security models. A completely contrary approach was tried in the Intel i432 processor and iMax operating system [Org83], which used a descriptor-based architecture and enforced protection at an extremely fine-grained level (on each memory reference). The programming language technology (Ada) was advanced for its time; the i432 provided hardware garbage collection, and provided high-level abstractions such as asynchronous inter-process communication (IPC). The logical support for objects was both attractive from a software engineering standpoint (as software module structure could be enforced by hardware mechanism) and also served as a basis for least privilege. The i432 and its hardware/software co-design succumbed to difficulties [Col88] such as inadequate use of caching technologies, inadequate memory bandwidth (e.g., a 16-bit memory bus supporting a 32-bit architecture), poor decisions in the ISA and poor code generation by the Ada compiler. All issues except the last (which can be addressed with co-design and measurement) are due in retrospect to inadequate hardware capacity, a problem we do not have. It is notable that some evolutionary developments in computer architecture (e.g., RISC [Rad82,PH96]) co-evolved with advances in memory architecture (large caches and register files) and compiler technology (register allocation techniques [CAC+81]).

Previous machines have employed tagged data, including Burroughs-Unisys MCP/AS [Org73, HLSG87], the Intel 432 [Org83, Int81], multiple generations of the Symbolics LISP Machine [Moo85, BCC$^+$87], IBM's System 38 [Lev84] and the Cambridge CAP machine [WN79]. Many of these machines employed a hardware/software architecture whereby access control to objects required possession of a hardware *capability* to perform the controlled action. The Burroughs machine used tags for security, but security guarantees required that all code be written in a high-level language and compiled by their compiler, a requirement that was not otherwise enforced by the system. System 38 gained considerable protection benefits from a single tag bit. The LISP Machine used tagging primarily for efficient support of LISP and graceful detection and handling of runtime errors (*e.g.* unintentionally treating a floating-point value as a pointer). Although security and isolation were not design goals of the LISP Machine processors or OS, this use does offer a practical improvement to security, minimizing the likelihood a latent programming bug can be exploited to create a breach. In contrast, our design exploits today's lower relative cost of hardware to support a large metadata space and an implementation capable of providing rigorous runtime security checking for all programs. This enables a new operating system model and close coupling with modern formal methods to reallocate roles in the design tradespace.

Virtual machine separation (e.g., [Cre81, RG05, BDF$^+$03]) is a technique employed today in an attempt to deal with the inherent insecurity of conventional systems. This provides a coarse-grain form of separation, where a user or application gets its own operating system. In contrast, we provide stronger isolation at finer granularity allowing application sandboxing and carefully controlled interaction between components. Satellite servers for our decomposed OS services can be used both for further prevention-in-depth and isolation of applications and interfaces between different system APIs.

The trend to microkernels (*e.g.*, [GDFR90]) embraced the idea of reducing the kernel, separating out services as separate processes isolated from each other. Performance concerns with commodity hardware kept the separate components relatively large; performance cost still limited adoption. More recently Tanenbaum [THB06] suggests the modern importance of security makes it worthwhile to sacrifice some performance for security.

The need for strong isolation has been recognized for embedded systems, leading to work on separation kernels (including the work on MILS [Rus81, AFTO04, NLI06, DNIL07]) which has largely focused on software isolation, with the usual performance considerations discouraging fine-grained compartmentalization. Our work can be seen as allowing these philosophies to be carried to the extreme by supporting them with appropriate fine-grained hardware isolation.

The Multics Kernel Project identified how the Multics kernel might be reduced to tens of thousands of lines of code with the promise that future kernels might be formally validated [SCSW77]. PSOS developed the design for a provably secure operating system that also exploited tagged data [FN79, NF03]. LOCK targeted a small security kernel with a hardware coprocessor to enforce security and judicious use of formal validation [Say02, Smi01, SBL89]. These systems made the necessary compromises to coarse-grained control (*e.g.*, pages and files), to the hardware of the day, and to the available verification technology.

Bell and LaPadula defined a formal security model for Mandatory-Access Control (MAC), including details of an implementation for Multics [BL75]. Hardware capability and performance limitations of the machines forced their model and rules to only work with coarse-grain entities (*e.g.*, files), outlawing many finer-grained interactions which would be consistent with conceptual policy intent, but could not properly be enforced by the hardware of the day [Bel05]. SELinux [GHRS05] shows how MAC could be added to Linux, but lacks the more formal basis of MAC in Multics and suffers from many of the underlying weaknesses of the Unix OS.

The Orange Book [oD85] prescribed verification from the beginning of the design for the highest security levels. While the philosophy and goals were correct, the verification technology and computing capacity of the day was inadequate for the task of validating fully capable systems. With advances in computational power and verification technology, including formal language design, program logics, mechanized proofs, proof assistants, and SAT solvers, we now have sufficient technology to automate the necessary proofs if we carefully co-design the language, operating system, and hardware architecture to work with the available verification technology.

Our techniques are consistent with industry hardware security efforts such as the Trusted Computing Group's *Trusted Platform Module*, a hardware module that provides a *hardware root of trust* useful for secure bootstrap [AFS97] and recovery [AKFS98]. The TPM's logical separation of functionality, with a separate unit validating operations, is consistent with our architecture.

The Spin [BCE+95], KaffeOS [BH05], and Singularity [FAH+06, WYA+07] projects each developed an operating system that inspires some of our design. In particular, these OS's used a type-safe high-level language to ensure basic control-flow and object-level integrity properties. Singularity uses separate address spaces communicating only via message passing, and provides convenient language mechanisms (tracked pointers) for supporting zero-overhead data movement. The security of these systems depended crucially on the correctness of the compiler, the type-checker, and the run-time system. These components were *not* formally verified and indeed, had bugs that an attacker could use. In contrast, we will formally verify the soundness of our type systems and compilers. Furthermore, these systems were designed to reduce overhead, instead of providing protection in depth. For example, they did not take advantage of (existing) hardware protection mechanisms, as we are proposing.

1.5 Papers of Previous Relevant Work

The following 3 papers are included to demonstrate previous work relevant to this BAA.

1. Shrobe, H., Knight, T., & DeHon, A. (2007, May 30). TIARA: *Trust Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture*. Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory.

2. Malecha, G., Morrisett, G., Shinnar, A., & Wisnesky, R. (2010). *Toward a verified relational database management system*. In POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 237-248.

3. Alexander, D. S., Menage, P. B., Keromytis, A. D., Arbaugh, W. A., Anagnostakis, K. G., & Smith, J. M. (2001, March). *The Price of Safety in an Active Network*, Journal of Communications and Networks, Vol. 3(1), pp. 5-18.