Appearing in *SASO2012 Adapative Host and Network Security Workshop*, September 14, 2012

Hardware Support for Safety Interlocks and Introspection

Udit Dhawan, Albert Kwon, Edin Kadric,
Cătălin Hriţcu, Benjamin C. Pierce,
Jonathan M. Smith, André DeHon
Dept. of Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, PA, USA
Email: udit@seas.upenn.edu

Gregory Malecha,
Greg Morrisett
Dept. of Computer Science
Harvard University
Cambridge, MA, USA
Email: greg@eecs.harvard.edu

Thomas F. Knight, Jr., Andrew Sutherland,
Tom Hawkins, Amanda Zyxnfryx,
David Wittenberg, Peter Trei,
Sumit Ray, Greg Sullivan
Advanced Information Technologies
BAE Systems
Burlington, MA, USA
Email: gregory.sullivan@baesystems.com

***Abstract*—Hardware interlocks that enforce semantic invariants and allow fine-grained privilege separation can be built with reasonable costs given modern semiconductor technology. In the common error-free case, these mechanisms operate largely in parallel with the intended computation, monitoring the semantic intent of the computation on an operation-by-operation basis without sacrificing cycles to perform security checks. We specifically explore five mechanisms: (1) pointers with manifest bounds (*fat pointers*), (2) hardware types (*atomic groups*), (3) processor-supported *authority*, (4) authority-changing procedure calls (*gates*), and (5) programmable metadata validation and propagation (*tags* and *dynamic tag management*). These mechanisms allow the processor to continuously introspect on its operation, efficiently triggering software handlers on events that require logging, merit sophisticated inspection, or prompt adaptation. We present results from our prototype FPGA implementation of a processor that incorporates these mechanisms, quantifying the logic, memory, and latency requirements. We show that the dominant cost is the wider memory necessary to hold our metadata (the atomic groups and programmable tags), that the added logic resources make up less than 20% of the area of the processor, that the concurrent checks do not degrade processor cycle time, and that the tag cache is comparable to a small L1 data cache.**

***Keywords*-Processor; security; least privilege; separation of privilege; complete mediation; hardware interlocks**

I. Introduction

Three of the most basic security principles identified in Saltzer and Schroeder's seminal paper [1] are *least-privilege operation*, *separation of privileges*, and *complete mediation*. Current computing hardware [2] and software systems are extremely vulnerable in part because they violate these basic principles: 1) Processors provide a small number of privilege levels (e.g., four in the x86), of which OS kernels typically exploit only two, resulting in a privileged kernel mode that has complete authority over the system (excessive and unseparated privileges); 2) Memory systems provide isolation only by separating large address spaces (excessive privilege), and it is expensive (thousands of cycles) to switch between address spaces (discouraging privilege separation); 3) Software (including the operating system) is organized into large address spaces of code and data, presenting a large surface area for attack where any weakness can be exploited to access or subvert the whole (lack of privilege separation); 4) Both hardware and system software treat the memory as "raw seething bits" whose meaning is not understood or respected, and hence they cannot enforce the intended semantic invariants of the computation or recognize when they have been violated (incomplete mediation).

Table I
Overview of Safety Interlock Mechanisms

Problem	Mechanism	Sec.
Memory Safety	Fat Pointers	III-A
Unintended reinterpretation of data contents	Hardware Types (Atomic Groups)	III-B
Excessive privilege	Authority	III-C
Expensive privilege change	Gate	III-D
Lack of fine-grained data and data flow containment; expensive data tracking	Programmable Tags Tag Management Unit (TMU)	III-E

Moore's Law scaling has made hardware inexpensive while ubiquitous use of networked computers for personal, sensitive, and critical tasks has raised the stakes for computer security. Both effects have changed the cost structure. It is now paramount that we reduce the vulnerabilities in our computer systems, and it is inexpensive to invest hardware as part of a holistic solution to do so.

In the CRASH/SAFE project [3], we have undertaken a clean-slate co-design of a secure network host, including novel hardware, operating system, programming language, and verification strategy based on modern cost structure and capabilities. This paper describes our in-progress design of hardware mechanisms that can enforce semantic invariants in the abstraction stack (*e.g.* stack integrity, memory safety, type safety including data and instruction distinction) and allow fine-grained, lightweight privilege separation and common-case mediation. We take a security-first approach, trying to understand what abstractions we want the secure computation to enforce and possible hardware that could enforce it. This is in stark contrast to hardware-first approaches that select the security policy based on what

hardware can provide with almost no overhead. Nonetheless, we hypothesize that, once the security goals are understood, clever hardware implementations can have very modest cost. The SAFE effort builds on design elements started in the TIARA project [4].

Many issues remain open, but our work to date shows that many hardware interlock mechanisms are eminently viable using today's semiconductor technology. Specifically, in this paper, we offer the following contributions:

- We identify a set of five hardware safety interlocks that can cooperate to allow fine-grained, least-privilege and separated privilege operation with complete mediation and lightweight change of privilege (Sec. III and Tab. I).
- We characterize the requirements for the five interlocks from an FPGA implementation (Sec. V and Tab. II).
- We show how they allow introspection (Sec. IV).

Before detailing the hardware safety mechanisms, we begin in the next section by further reviewing conventional hardware and OS trends.

II. Putting the problem in historical context

The designers of today's systems violated Saltzer and Schroeder's security principles as a tradeoff to achieve acceptable performance in an era of scarce hardware resources. When the first general-purpose computers and operating systems were built, it was a challenge to build a capable processor economically; this made it difficult to allocate hardware to security. In its day, Multics [5] was considered extravagant for the inclusion of virtual memory hardware to support process isolation, and even Multics compromised by only providing enforcement rules on coarse-grained entities (*e.g.*, files), outlawing many finer-grained interactions that would be consistent with conceptual policy intent but could not properly be enforced by the hardware of the day [6].

Similarly, the commercial versions of the MIT Lisp Machine [7] were considered extravagant for their inclusion of safety features such as data type tags, bounds checking, and garbage collection support. The i432 attempted to provide a richer security model than its contemporaries including Multics, but the limited silicon budget of its day forced it to sacrifice on-chip memory (register files and caches) to accommodate the security features [8]. The resulting poor performance reinforced the conventional wisdom that rich hardware support for program semantics is not worthwhile.

The first microprocessors and microprocessor OS'es simply did without process isolation. Modern processors [2] and OSes provide only a single mechanism to applications for privilege and separation management: the user/kernel distinction managed by the virtual memory hardware, which provides page level read/write/execute access controls on a per-process basis. While this is valuable, it operates at an awkward level of granularity, controlling access to individual pages of virtual memory rather than to the semantically more meaningful unit of the individual object. The one notable instance of new hardware support for security and isolation in commodity processors is hardware support for virtualization (*e.g.* Intel VT-X [9], AMD-V). This, however, moves toward even more coarse-grained isolation, separating entire OS and runtime images.

One consequence of the large amount of state that must be exchanged to safely hand off the processor from one virtual memory context to another (*e.g.*, special processor state, register file, TLB content) is that context switches are relatively expensive (thousands of cycles); this has motivated system programmers to move code into the kernel to limit the number of domain crossings. All this code enjoys unlimited privilege, even though any given component will typically need only a very specific set of privileges.

Lack of hardware support for fine-grained security leads to several problems. Violation of intended semantics allows attackers to subvert the system (*e.g.* write beyond the end of an object, overwrite the words on stack that should not be visible or addressable, treat a data word as an indirection address or branch target, branch to and execute words that are not instructions, perform a class method on data that is not a class instance). Worse, once an error is exploited, the attackers can often accumulate privileges to take control of the entire system (*e.g.* access to read or modify all memory, access to all privileged operations: rewrite page tables, change permissions, access devices, install software).

We submit that conventional systems are insecure in part because of security–performance tradeoffs they made based on now-*obsolete* technology assumptions. Decades of Moore's Law hardware scaling have delivered at least three orders of magnitude greater hardware capacity today than when the architecture of these systems was selected. Intel first integrated support for virtual memory process isolation with the 80286 (1982), on a die containing 1.34×10^5 transistors. By 2012 Intel ships six-core Core i7 processor dies with over 2×10^9 transistors. With a competent processor now small compared to a silicon die, we are living in an era of abundant hardware resources. It makes sense to commit a modest portion of these resources to reducing security–performance tradeoffs and eliminating some classes of security vulnerabilities by *baking in* basic security principles.

III. Hardware Safety Interlock Mechanisms

The SAFE processor is a single-address-space architecture [10] with mechanisms for fine-grained protection and automatic memory management. Virtual memory is replaced with a collection of hardware interlock mechanisms that better support fine-grained, object-level privilege separation and semantic enforcement. Fig. 1 shows the microarchitecture for the SAFE processor. Along with the functional units found in a conventional processor—the integer ALU (IALU), floating point unit (FPU), branching unit (BU), etc.—the SAFE processor includes specialized functional units for fine-grained metadata validation. These units are

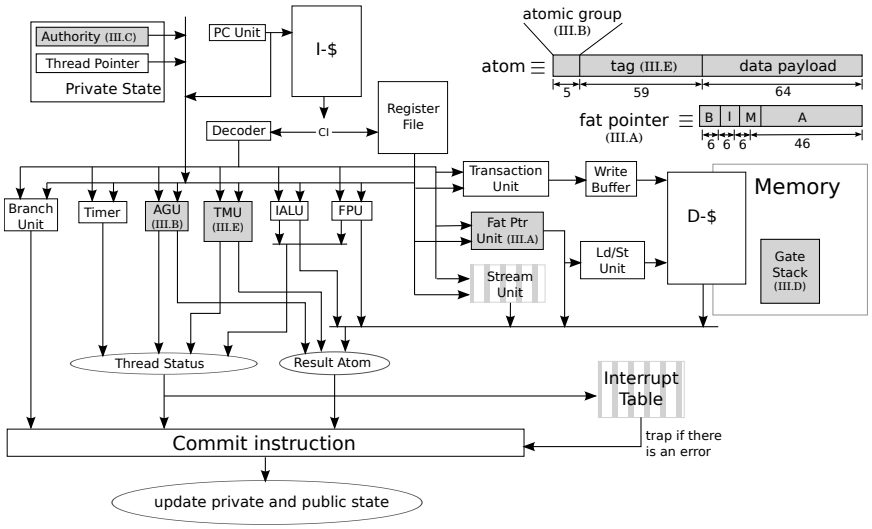

Figure 1. SAFE Microarchitecture

shown in solid gray in Fig. 1 and operate in parallel with the datapath functional units so that they do not slow down the computation in the normal case. In the rest of this section, we briefly discuss five key hardware mechanisms currently implemented in our prototype.

A. Fat Pointers

Problem: Virtual memory provides coarse-grained address space separation, but address spaces are large, encouraging applications and OS kernels to live within a single address space—no one would consider creating a new address space for one object. Consequently, within an address space, there is little or no isolation between objects. Violation of the spatial bounds of objects (*e.g.* buffer overflow) is an abundant source of bugs and vulnerabilities.

Approach: The SAFE processor incorporates *fat pointers*, pointers that include a base and bound (*e.g.*, [11]), to guarantee memory safety. Modern systems have explored the use of fat pointers in software using multiple words to represent the pointer, its base, and its bound. These schemes depend on the compiler or libraries to maintain the software pointers and incur runtime overheads of a factor of 2–3×. In contrast, we adopt Brown's approximate encoding scheme [12], which allows us to pack an entire fat pointer into a 64-bit machine word. Other efforts dealing with legacy code for unsafe languages have provided hardware support by putting the bound information in a shadow space [13].

Our scheme divides the 64-bit word into 46b for the actual address, A, and splits the remaining 18 bits among three fields: a 6b block size B, a 6b length mantissa L, and a 6b block offset F. We allocate $L \times 2^B$ words for every object, aligned to a 2^B boundary. This allows us to compute the base and approximate bound by shifts and simple arithmetic. Memory lost to alignment fragmentation is less than $2^{-(L-1)}$ (about 3% for L=6).

Implementation: The entire fat pointer is treated as an indivisible atom by the hardware; that is, the fat pointer is stored in memory as a single word, read and written as a word, and kept together through computations on the processor. Only the hardware *Fat Pointer Unit*, shown in Fig. 1, deals with the individual fields to perform updates and checks. The fat pointer unit operates in parallel with the ALU and memory operations. In the common, non-erroneous case, the validation does not delay normal operation. In our first implementation, the fat pointer unit was in the critical path of the processor. However, we have recently developed a recoding that uses the compact encoding in memory and keeps a decoded version in the register file to reduce the fat pointer decoding, update, and validation operations below 4.2 ns on the 40 nm Virtex-6 FPGA when working with the full 46b address. This is 10% slower than our other critical paths (Tab. II), so remains a target for optimization. On 32b addresses, it is below 3.9 ns. The Fat Pointer unit requires 1156 LUTs (less than the area of a floating-point adder) and adds 20 BRAMs to the 32 BRAMs required for the baseline register file.

B. Atomic Groups: Hardware Typed Memory Words

Problem: Since memory can only be addressed through fat pointers, SAFE can control the data a particular piece of code can see by controlling the fat pointers to which it has access. That is, the fat pointers serve as an object capability (*c.f.*, [14]). However, to make this sound, we must assure that it is not possible for arbitrary code to create (*forge*) a valid fat pointer. Rather, the only way to get a valid pointer should be for some appropriately privileged entity to

provide it. This is different from pointers on a conventional architecture, which can be freely created from integers.

Pointers are only one special kind of data whose creation and use it is useful to control. Some attacks on conventional processors exploit the fact that the processor cannot distinguish data and code and trick the processor into reinterpreting data as code, thereby injecting code sequences.

Approach: To prevent forging, reinterpretation, and misinterpretation of words, SAFE types every word in memory. Since the type is an indivisible part of every word, we call the typed words *atoms* (Fig. 1) and the type an *atomic group*. This allows the processor to know the intended use of each atom in memory and enforce appropriate, type-safe usage. For example, only fat pointers can be used to reference memory, only instructions can be dispatched to the processor, generic boolean operations can only be performed on integers, and uninitialized data can be distinguished from valid data. This extends the idea of a typed architecture (*e.g.*, [7]) or a typed assembly language (*e.g.*, [15]).

SAFE currently uses 5 bits to distinguish the following atomic groups: Frame Pointer, Instruction Pointer, Gate Pointer (Section III-D), Stream Read Pointer, Stream Write Pointer, Thread Pointer, Forwarding Pointer, IEEE Double Float, Integer, Instruction, Authority, Principal, Empty, Uninitialized, and Error. Pointer distinctions allow SAFE to enforce different limitations on pointer use (*e.g.* only an instruction pointer can become the program counter).

Implementation: The *atomic group unit* (*AGU*) checks each instruction to validate that the atomic group (type) of the operands is consistent with the instruction, flags an error when type mismatches occur, and selects the atomic group for the result of the instruction. The *AGU* operates in parallel with the functional units that operate on the data and with memory access. In our FPGA implementation, it only takes 135 LUTs and completes in 3.6 ns. This delay is roughly the same as the IALU, so when executed in parallel, we see no impact on performance.

C. Who is Running this Code?: Authority

Problem: To support separation of privilege and least privilege, we must support differently privileged entities. As an example, the set of unforgeable atomic groups raises the issue of who has privilege to assign a particular atomic group to some atom. A primitive memory allocator needs to create pointers, but does not need special privileges over threads. The traditional supervisor mode solution concentrates privilege in a single, monolithic authority, violating least privilege and creating a single point of vulnerability.

Approach: In our clean-slate approach, we introduce a rich notion of *authority* to the processor hardware and use it to mediate all operations. We use specially tagged (*i.e.*, distinguished atomic group) pointers to serve as authority, leaving the encoding and interpretation of authority to software. Since we use pointers, they are as plentiful as main memory allows, facilitating fine-grained subdivision of responsibility. At any point in time, the processor is running on behalf of a specific authority, which it tracks in a special processor register. Among other things, this allows the SAFE processor to restrict the ability to perform particular instructions (or even particular instructions on particular atomic groups) to specific authorities in a fine-grained manner. For example, only the *allocator* is allowed to execute the privileged `framptr` instruction, which creates a new fat pointer for a frame of memory locations.

Implementation: The main requirement is a register in the thread state to represent the authority with appropriate restrictions on how this authority can be modified. We will see that this interacts closely with gate (Section III-D) and metadata validation (Section III-E).

Open Issues: Exactly what authorities represent at higher levels is an active area of discussion. The facilities we provide are designed to support a wide range of idioms. For example, an authority might pick out a point in a hierarchy of principals, some of which *act-for* [16] or *delegate to* [17] others. Or an authority might represent a set of first-class *capabilities* to perform actions such as declassifying or endorsing certain types of information. Moreover, by using different forms of gate calls (Section III-D), authority in a running program can be managed either lexically (calling a procedure raises the current authority to include that of the procedure's creator) or dynamically (on procedure call, the authority of the caller is implicitly passed to the callee).

D. Changing Authority: Gates

Problem: To support fine-grained privilege separation, we must make the change of privileges secure and inexpensive. Since we tie privileges to authorities, that means we need to change authorities in a lightweight manner. Privilege separation in traditional systems is tied to virtual memory address contexts, which are relatively expensive to switch (*e.g.*, [18], [19]). Even on the i432, it took close to 1000 cycles to perform a privilege-changing call [8].

Approach: To support lightweight privilege change, we provide a special form of procedure call that changes the authority. We call this authority changing procedure call a *gate* since it performs the same logical function as a Multics gate [5]. Our gate is also similar to an i432 ENTER [8]. However, unlike the Multics gate or i432 ENTER, we exploit today's greater hardware capacity to make gate operations as inexpensive as an ordinary procedure call.

We represent a gate as a 3-atom object containing an instruction pointer, an authority, and an environment pointer. It is essentially a closure (combination of code and data) that has been extended with privilege (the authority). The environment pointer can be seen as capturing the lexically defined environment or the object-local data. When invoked, the processor's program counter (PC), authority, and environment pointers are changed to the values specified in

the gate, with the old values pushed onto a call stack. The hardware deals atomically with the gate: once created, it cannot be mutated, the constituent components cannot be dereferenced or extracted, and invocation is an atomic change of PC, authority, and environment.

A gate can be viewed as providing controlled access to a service or object. The data associated with the object or service can be made private to the service. The gate call changes the authority and recovers the pointers (object capabilities) that provide access to this private data. However, this access is only provided in order to run the code specified by the instruction pointer. This associated code can then be used to limit the operations allowed. For example, the allocator can provide a gate to perform allocation without exposing its internal state or giving away its privileged operations. In the extreme, if we use a separate authority for each object, gates can be seen as providing enforced data hiding and encapsulation object semantics—the only way to access the data is through the exported gates.

To ensure gate and authority integrity, the processor uses a special call stack that we call a *gate stack*. Unlike the call stack in conventional architectures, the gate stack is *not* accessible to the code. This gate stack is only used for saving and restoring PC, authority, and environment around procedure and gate calls—the compiler or programmer is responsible for local data allocations. The processor maintains a pointer to the gate stack per thread and this pointer is only manipulated by call and return instructions. We treat the gate stack specially in order to (1) assure that pointer capabilities and authorities do not flow to and cannot be modified by called routines, (2) prevent information from the caller chain from flowing to the callee, and (3) guarantee the called program can only return to defined join points with designated environments and authorities. Note that this generalizes and replaces the notion of a system call, which traditionally requires separate user and kernel stacks and a context switch.

Implementation: To perform a call we must load the 3 atoms in the gate object into the processor state, save the corresponding 3 atoms from the processor state onto the gate stack, and update the gate stack pointer. We force gate stack frames to be aligned on a 4-atom boundary, design our L1 data cache to be dual ported and multiples of 4 atoms wide, and provide a wide bus between the processor an the L1 data cache. As a result, we can perform the entire state exchange in a *single cycle*. This makes gate calls and returns no more expensive than ordinary procedure calls and returns, removing the traditional performance disincentive to privilege separation.

Here we exploit that fact that L1 data caches are on chip and already organized into wide cache lines, making the wider interface between the processor and this piece of the memory hierarchy inexpensive. Embedded memories on FPGAs are dual ported, so there is no additional cost to this. In a full custom design, we might prefer to use a separate memory bank for the gate stack or accept a second cycle to perform the write separately from the read in order to avoid making the L1 data cache dual ported.

Open Issues: There are many variants on gate calls. Should there be a way to pass limited authority into a gate? Is tail-call optimization essential, and can it be secure? When is it appropriate to restore the tag on the PC associated with the call (relevant to implicit flows in the next section)? What is the appropriate hygiene for registers across authority-changing calls? Should there be a variant that allows timeouts on gates, allowing the caller to regain control if the callee either mistakenly or maliciously fails to return control, and how should this functionality be divided across hardware and software? Is there value in "functional" gate calls, whose results can be cached in hardware? To explore such questions, we currently support a wide variety of gate calls. We expect experience and benchmarking to help identify the essential set.

While we can exchange the key processor state in a single cycle, if the code and environment for the gate are not in a cache, they will incur additional cycles to fault the data into the L1 caches. More benchmarks are needed to clarify how much this impacts effective procedure call time and drive memory system optimization to minimize the impact.

E. Metadata Validation, Propagation, and Monitoring

Problem: To limit the flow of information for secrecy and integrity, many recent security architectures (both software and hardware) have adopted dynamic information-flow tracking [20]–[24]. Most commonly, a few tag bits are associated with every word to specify if it is tainted or not (and taint level in case of multiple bits). These taint bits are consulted at various spatial or temporal events to prevent unintended data sharing or usage. Most of these systems offer a limited number of tag bits and a few policies defining taint propagation and restrictions. While this work illustrates the power of metadata tracking, combination, and checking, it remains unclear what tags, policies, or collection of policies a processor should support. Furthermore, in order to offer strong security assurances, it is necessary to track not only explicit flows of information as data is copied and manipulated but also *implicit flows* through conditional control transfers [25]; dynamic approaches to implicit flow tracking remain an active area of research both in our SAFE effort and in the larger community.

Native types at the hardware level (Sec. III-B) makes the processor aware of the most basic semantic invariants of any computation. However, programming languages and programmers will create rich, fine-grained data types that have their own semantic invariants that must be enforced (*e.g.*, an integer type for a bank account number should not be interchangeable with a integer type used for a date). Implementations that dynamically validate and enforce these

policies in software become expensive, forcing a tradeoff between good defensive programming and performance.

Approach: In addition to the atomic group (Sec. III-B), we equip every atom with a programmable tag (Fig. 1). Every instruction is validated against a set of software-programmable rules to determine if the operation is allowed and what the resulting tags should be. To support extensibility, each tag is a pointer to a data structure that can be defined and interpreted by the programmable rules. As such, it can be used to combine a collection of properties that may be used orthogonally (*e.g.* data types and usage restrictions, taint, secrecy labels, integrity labels, provenance). In our present design, we allow rich rules that take as input nine values or tags: the current operation (*e.g.* add, load, gate), the current authority, the tags on the thread pointer, the PC, the three operands, and the result from memory. In response, the rules can specify whether or not the operation is allowed and, if allowed, seven result values or tags: the new authority, the tag on the thread pointer, the PC, the written operands (up to 3), and the value to be written to memory.

Implementation: To allow this rich and extensible mediation while minimizing the impact on execution time, we add a separate functional unit, the Tag Management Unit (TMU) (Fig. 1) that runs in parallel with operations on the data. The TMU is a cache on the software rules. It matches the 9 input fields (a total of 375b for the full 46b address case) and produces the 8 output results (323b). This caching technique is similar to the one used in [24]. The TMU operates in parallel with the datapath functional units so the validation occurs without increasing processor cycle time or adding cycles in the common successful case.

To deal with bootstrapping of primitive software services, we decompose the TMU into two pieces: (1) a static rule cache for primitive services including the services necessary to update the dynamic rule cache and (2) a dynamic rule cache for extensible rules. In Sec. V we show that a 1024-entry dynamic TMU is only slightly larger than a small L1 data cache.

Open Issues: A central component of the SAFE research is what the tag rules should be and how to make them practically usable in a programming language. The richness of our current design specifically supports experimentation and extension.

The use of full pointers provides extensibility and compactness in the number of tags used, but it hides the structure of the tags (*i.e.* ability to decompose tag rules by their orthogonal fields). This loss of structure increases the effective size of the rule working set. As we get further experience with applications and label models, a key question will be whether or not this lack of structure makes the working set unworkably large.

Using a pointer the same size as main memory address pointers on every word is convenient and provides great flexibility for experimentation but incurs 100% overhead for tag storage and movement. Obvious directions to reduce this overhead include using a restricted range of the address space for the tag pointer (*e.g.* using 28b of pointer address rather than 46b) and associating the tag pointer with a larger payload than a single 64b word (*e.g.* a 128b or 256b packed word or a flexible granularity scheme such as [13], [20]).

IV. Introspection Support

The hardware mechanisms employed in the SAFE architecture facilitate efficient, targeted low-level monitoring and ease invocation of a deeper introspection when necessary.

Problem: In conventional systems, it can be expensive and inefficient to introduce inspection into every *potential* point where a monitored event may occur (*e.g.*, every write that might mutate a field in a data structure). Virtual memory tricks (*e.g.*, marking a page read-only) are often the most efficient, but are generally mismatched with the granularity of the object, leading to spurious traps.

Approach: Our flexible, fine-grained metadata tags with rich hardware matching allows us to precisely target specific scenarios and invoke software handlers only when those conditions occur. When the common case is *not* the event of interest, the TMU can validate the operation without adding cycles to slow down normal case operation, invoking the software handler only when the uncommon target conditions occur. Rich, fine-grained tagging with careful selection of tags allows us to be very specific about the case that merits monitoring.

Impact: We may introduce specific tags precisely to differentiate them for monitoring. Code-specific breakpoints can be generated with tags on the instructions. Data values can be tagged to generate traps when used in specific ways or places. Memory locations can be tagged to generate traps on their use or modification. The ability to use tags for provenance and taint tracking means tags can carry digested information about the history and trajectory of an atom that can be used in identifying and triggering specific conditions.

These traps can provide efficient triggers for reference monitor inspection [26]. Once checked, immutable data can be tagged with properties derived from the inspection, perhaps providing an endorsement that avoids the need for future inspection (*e.g.* [27]). These focused traps can also be used to log events, such as the use of particular privileges in specific contexts. Changing the tags or rules employed can be used to adapt the level and focus of the monitoring without changing the application code (*c.f.* [28]).

Open Questions: Exploiting these facilities for rich monitoring on top of security and integrity information flow raises a number of its own challenges, including: How should we make these facilities accessible to the application and system programmer? What are the best patterns for using tags, rules, and the TMU to efficiently monitor data? How can these monitors be used without leaking information (*e.g.*, [29])?

V. FPGA Prototype Implementation

We are developing the prototype SAFE processor as shown in Figure 1 using the Bluespec SystemVerilog [30] hardware description language and targeting a Xilinx ML605 Virtex 6 (40nm) FPGA development board [31] for hardware testing. Currently we have an unpipelined, functional design that executes five 5 ns cycles per operation. Mandatory address pipelining on Virtex embedded memories forces the multi-cycle operation. The number of FPGA resources consumed by individual functional units in the current prototype are shown in Tab. II. The current implementation is aimed at functional correctness, and optimization has been limited to key functional units such as the fat pointer unit (Sec. III-A).

Nevertheless, the five security modules shown in Sec. III are not large. The modules take 4825 LUTs and 66 BRAMs total, which is about 18% of LUTs and 44% of BRAMs used for the processor. Some cost for integrating these functional units shows up in the "Top" module logic area. The security modules run in parallel with the IALU, FPU, and I-Cache access, which have delays of 3.7 ns, 4.0 ns, and 3.6 ns respectively. All security units run under 3.9 ns except for the 46b fat-pointer decode that runs in 4.2 ns.

Currently we have implemented only the L1 instruction and data caches, each with a block size of 4 atoms and 1024 entries. This gives us 32KB (4K atoms) of instruction and data capacity (not accounting for the metadata) each that can be cached at any instant. These caches are backed by the main memory which lives in an off-chip DRAM. Both of these caches employ a low conflict rate multiple hash function architecture. We use a similar architecture for the TMU cache. Currently we use 28 bits for all addresses as an implementation simplification since we do not expect to address over 2^{28+3}=2 GB with the FPGA prototype. Due to the 28b addresses, the dynamic rules are stored in a 1024-entry TMU cache as complete 527 bits (inputs + outputs) vectors as opposed to 698 bits for the full address range case for each rule. This results in the current TMU being comparable in size to the small L1 data/instruction cache.

Our current multicycle hardware prototype will facilitate the transition to a pipelined version of the processor with slight re-organization of hardware units. We expect to be able to run our FPGA-based processor at a clock speed of 250 MHz with 5 pipeline stages.

Table II
Implementation Statistics for FPGA prototype of Processor

Module	Security?	LoC	LUTs #	LUTs %	BRAMs	Delay (ns)
Top (control, multiplexing)	○	1591	12200	45.7	0	-
AGU	•	138	135	0.4	0	3.6
ALU		80	852	3.1	0	3.7
Floating Point Adder (5 pipeline stages)		550	1300	4.8	0	4.0
Branch Unit		58	46	0.1	0	2.3
Fat Pointer Unit	•	79	1156	4.2	0	
update and check						3.6
decode (32b)						3.9
decode (46b)						4.2
Gate Stack	•	102	852	3.1	0	2.6
Load Store Unit		53	2	0.0	0	0.4
PC Unit	•	135	1647	6.1	0	3.7
Streaming Unit		52	48	0.1	0	2.0
Interposed Unit	•	57	5	0.0	0	1.9
Timer		85	472	1.7	0	3.1
Write Buffer (32-entry)		286	4799	17.9	0	2.9
TMU (28b tags)	•	728	1030	3.8		
Static (1024 rules)					21	3.6
Dynamic (1024 rules)					25	3.6
Register File (32 GPR)		215	1108	4.1	52	3.0
I-$ (4K atoms)		-	643	2.3	26	3.6
D-$ (4K atoms)		-	643	2.3	26	3.6
Overall SAFE Processor		**4252**	**26958**		**150**	**5.0**
Usage (xc6vlx240t device)		**-**	**18%**		**36%**	**-**

VI. Conclusions

The modern hardware cost landscape is qualitatively different from the one under which conventional processor, microprocessor, and operating systems were designed. This makes it viable to consider hardware safety interlocks that were unthinkable a few decades ago. These interlocks can operate in parallel with datapath functional operations so that they do not slow down normal operation. Furthermore, they ease the traditional tradeoff between security and performance, enabling fine-grained, object-level separation of privileges and complete, instruction-by-instruction, mediation. We have shown: (1) how bounds can be encoded in a 64b pointer word and checked in parallel with operation; (2) how words can be typed and checked in parallel with operation; (3) how the processor can always know on whose behalf it is running; (4) how privilege changes can occur at the level of a procedure call without slowing down the procedure call; and (5) how rich information flow and semantic constraints can be propagated and validated in parallel with operation. All these operations together can be implemented with 25% additional logic (less than 20% of final processor logic resources) and with an additional cache comparable to a small L1 data cache.

Acknowledgment

This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, September 1975.

[2] J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, Inc., 2003.

[3] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan, "Preliminary design of the SAFE platform," in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, ser. PLOS, Oct. 2011. [Online]. Available: http://www.crash-safe.org/sites/default/files/plos11-final_0.pdf

[4] H. Shrobe, A. DeHon, and T. F. Knight, Jr., "Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (tiara)," December 2009. [Online]. Available: http://www.dtic.mil/srch/doc?collection=t3&id=ADA511350

[5] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, July 1974.

[6] D. E. Bell, "Looking back at the Bell-La Padula model," *Proceedings of the Annual Computer Security Applications Conference*, pp. 337–351, 2005.

[7] R. Greenblatt, T. Knight, Jr., J. Holloway, D. Moon, and D. Weinreb, "The LISP machine," in *Interactive Programming Environments*. McGraw-Hill, 1984.

[8] R. P. Colwell, E. F. Gehringer, and E. D. Jensen, "Performance effects of architectural complexity in the Intel 432," *ACM Trans. Comput. Syst.*, vol. 6, pp. 296–339, August 1988. [Online]. Available: http://doi.acm.org/10.1145/45059.214411

[9] F. Leung, G. Neiger, D. Rodgers, A. Santoni, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," *Intel Technology Journal*, Aug. 2006. [Online]. Available: http://www.intel.com/technology/itj/2006/v10i3/

[10] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4, 1994.

[11] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1065887.1065892

[12] J. Brown, J. Grossman, A. Huang, and T. F. Knight, Jr., "A capability representation with embedded address and nearly-exact object bounds," MIT AI Lab, Tech. Rep. 5, April 2000, aries Project. [Online]. Available: http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-05.pdf

[13] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 103–114.

[14] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: a fast capability system," in *Proceedings of the 17th Symposium on Operating Systems Principles*, ser. SOSP. ACM, 1999, pp. 170–185. [Online]. Available: http://www.eros-os.org/papers/sosp99-eros-preprint.ps

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, pp. 527–568, 1999.

[16] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, pp. 410–442, October 2000. [Online]. Available: http://doi.acm.org/10.1145/363516.363526

[17] W. Cheng, D. R. K. Ports, D. Schultz, J. Cowling, V. Popic, A. Blankstein, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for usable information flow control in Aeolus," in *Proceedings of the 2012 USENIX Annual Technical Conference*, Jun. 2012. [Online]. Available: http://pmg.csail.mit.edu/pubs/cheng12__abstr_usabl_infor_flow_contr_aeolus-abstract.html

[18] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware," DEC WRL, Digital Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301, WRL TN 11, October 1989.

[19] T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "The interaction of architectures and operating system design," in *Fourth International Conference on Architectural Support for Programming Languages*, April 1991, pp. 108–120.

[20] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 304–315.

[21] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 85–96.

[22] J. R. Crandall, F. T. Chong, and S. F. Wu, "Minos: Architectural support for protecting control data," *ACM Transactions on Architecture and Code Optimization*, vol. 5, pp. 359–389, December 2006.

[23] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the International Symposium on Computer Architecture*, 2007, pp. 482–493.

[24] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *Proc. of the 14th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2008.

[25] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS, 2008, pp. 56–70. [Online]. Available: http://www.cs.umd.edu/~mwh/papers/king08implicit.html

[26] C. E. Irvine, "The reference monitor concept as a unifying principle in computer security education," in *In Proceedings of the IFIP TC11 WG 11.8 First World Conference on Information Security Education*, 1999, pp. 27–37.

[27] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *ACM SOSP*, October 2009. [Online]. Available: http://pdos.csail.mit.edu/papers/resin:sosp09/resin:sosp09.pdf

[28] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[29] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," in *Proceedings of the 3rd European Conference on Computer Systems*, ser. Eurosys. ACM, 2008, pp. 301–313. [Online]. Available: http://www.cs.ucla.edu/~pefstath/papers/efstathopoulos08manageable.pdf

[30] Bluespec, Inc., "Bluespec SystemVerilog." [Online]. Available: http://www.bluespec.com

[31] Xilinx, Inc., "Virtex-6 FPGA ML605 Evaluation Kit." [Online]. Available: http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm