A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information Flow

Jeremy Brown Thomas F. Knight, Jr.

1 Introduction

With each passing year, more and more valuable, confidential information is stored in government and commercial computer systems. Ensuring the security of those computer systems is a challenge with social, political, and technological aspects; computer networks, however, make the technological aspects particularly important as computer systems are exposed to assault from remote sites.

Two critical components of the technological computer security problem are access control and data dissemination control. Access control mechanisms prevent unauthorized parties from accessing (e.g. reading, modifying, or executing) confidential data or programs. Data dissemination control mechanisms prevent confidential data from being exposed to unauthorized parties, either by accident or due to malicious code which has gained read-access to the data; e.g. a malicious or erroneous program should never be able to read a "Top Secret" value and write it out as an "Unclassified" result.

In this memo we present two contributions addressing the problem of controlling data dissemination, also known as ensuring secure information flow. First, we present a sound, flexible model which dynamically ensures secure data flow with respect to a lattice-based information flow policy, with security classification on a per-word basis. Second, we present a set of hardware mechanisms, most notably the Hash Execution (HEX) unit, which enable the practical implementation of our model. We believe that recent trends in logic and memory density and costs make the architectural overhead of our mechanisms small, and that they are more than offset by the significant benefits they bring to system security.

Our dynamic strategy has several advantages over static (compile-time) verificationof secure information flow. It is not conservative in its enforcement of security policy; static verification techniques must reject some programs which are, in fact, correct, whereas our dynamic strategy will safely execute them. Also, our strategy does not require that the security tag for each storage location be fixed; instead, our mechanism allows security tags to be adjusted according to a set of rules which guarantees that no information is leaked due to those adjustments.

Perhaps the most important advantage of our dynamic strategy over static verification strategies, however, is that it reduces the size of the required *Trusted Computing Base* (TCB). A TCB consists of a set of hardware and software mechanisms which, if correctly implemented, guarantee that regardless of any other code run on the system, security will not be violated. Verifying the correctness of a TCB is laborious at best (e.g. [21, 14]), and considerable work tends to go into minimizing its size (e.g. [20].) Compile-time program verification places the verifying compiler within the bounds of the TCB; unfortunately, a compiler is an extremely large, complex piece of software. By contrast, our dynamic mechanism guarantees security with a TCB composed of only a few simple hardware mechanisms and software routines.

The remainder of this paper is structured as follows. In Section 2, we discuss a handful of previous works in the computer security field, particularly with respect to secure data flow. In Section 3, we present an abstract architecture which ensures that information flow is secure according to a lattice model; we also present a general rule enabling authorized principals to safely declassify a selected datum without unintentionally leaking information about other data. In Section 4, we describe the hash execution (HEX) unit, a general-purpose processor component with particular applicability to our security model. In Section 5, we discuss a few additional matters relating to the practical implementation of our abstract model. We conclude in Section 6.

Project Aries Technical Memo ARIES-TM-015
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA USA

Research performed under
DARPA/AFOSR Contract Number F306029810172

2 Related Work

There has been a great deal of work on computer security in general and on secure information flow in specific, and we make no attempt here at identifying all or even most of it. Instead, we focus on a few key works which have direct relevance to this memo.

2.1 Schemes for dynamic, secure data flow

Several previous works describe schemes for dynamically ensuring secure data flow.

Lampson outlines the parameters of the problem in [12], suggesting that "confined" programs or procedures must be "memoryless". Fenton [8] describes the Data Mark Machine, an abstract model which implements memoryless confinement.

The Data Mark Machine includes a return-address stack used to allow safe declassification of the program counter: an unclassified process may push a return-address onto the stack, and a classified process may then declassify itself by popping the unclassified address into its program counter.

Building on [8], Gat and Saal [10] address the additional problem of general purpose registers, suggesting that any register which has a classified value written into it in the course of a classified computation must be returned to its original value when the program counter exits the classified region.

The Hydra [23, 3] operating system takes a different approach to implementing confined procedures, eschewing classifications in favor of permissions-based confinement. The caller of a routine can insist that the called routine may only write data into storage explicitly provided to the routine by the caller; in this way, the caller has complete control over where its data (or derivatives thereof) may be stored upon return from the called routine. Since Hydra does not actually tag data with security classifications, however, if the caller accidentally provides publicly-reachable storage as an argument to the confined procedure, confidential information may easily be revealed.

On the other hand, the ADEPT-50 [22] operating system does maintain security classifications, although on a rather coarse, per-file basis. Whenever a job creates a new file, the classification of that file is at least as great as that of all files the job has previously accessed; we refer to this type of scheme, in which the classification of a job or process cannot be lowered after it has been raised, as a "high-water mark" scheme. Since the goal of ADEPT's classification scheme is merely to prevent unauthorized access, there is no attempt made to prevent classified information from being written into a previously-created, unclassified file.

The surveillance protection mechanism [11] dynamically tracks the classification of each variable and of the program counter; it assumes that only the output of a program will be visible to any other process, and refuses to display it if the program counter has been contaminated by overly classified data. The program counter classification can only increase; as with any other high-water mark scheme, the surveillance protection mechanism is therefore conservative in its security policy enforcement.

The Privacy Restriction Processor [19] attempts to ensure data flow security by associating classifications with each storage region (segment), and with each process counter. The process counter classification is monotonically-increasing; thus the PRP is another high-water mark scheme. Additionally, the PRP allows the classification of each segment to monotonically increase; because there is no moderation of when these classifications may increase, such reclassification actually enables implicit information leakage as described in [4].

2.2 Lattice-based secure information flow

Denning [4] describes the lattice model of information flow that we use as the foundation for our dynamically secure abstract architecture; she also identifies several flaws and limitations of previous schemes for dynamically ensuring secure data flow which provided inspiration and counter-examples for our design.

The sequel work [5] presents a mechanism for compile-time verification of information flow security in a program with respect to a particular security lattice; every storage location is statically tagged with its security classification.

Myers and Liskov [17, 18] present a specific (and in our opinion particularly elegant) security classification scheme; equivalence-classes on their security labels form a lattice, and thus essentially the same static certification techniques as described in [5] apply. Additionally, the structure of their security labels enables a tasteful mechanism for decentralized declassification of data by owning principals. Myers [16] describes a variation of the Java programming language which includes this security labeling scheme.

2.3 Capabilities

Finally, although we have in general made no attempt to address matters of access control, the correctness of our data-dissemination control model relies on the use of *capabilities* [6, 13] – hardware-protected pointers denoting a specific region of memory – to ensure that unallocated memory may not be observed by any process. In particular, we favor guarded-pointer style capabilities [2, 1] which efficiently encode base and bounds in the capability representation itself in a fashion which allows the capability to point directly at any word in its range, rather than just the first word.

3 Abstract Architecture

In this section, we describe an abstract architecture which dynamically ensures secure information flow when running arbitrary code. The architecture provides the following features:

- Precise (non-conservative) enforcement of a lattice-based security policy
- Dynamic security tags on a per-word basis
- Safe overwriting of values with values of different security levels
- Safe declassification of a program counter when its dependency on classified data ends (i.e. the PC is not stuck at the security "high-water mark")

At a high level, our goal is to provide confinement ([12]) by ensuring that no modifications to machine state performed by a process running with a classified program counter may be observed by processes with inferior or orthogonal classifications – these processes can thus be said to "have no memory" of the operations performed by the classified process. This imposes restrictions on when a process may overwrite an existing value both in memory, where the act of overwriting must be invisible to processes with inferior/orthogonal classifications, and in registers, where the act of overwriting must be invisible to the process itself when it has escaped from the classified region of execution.

As per Denning [4], we enforce a security policy described by a lattice of security classes; $\bot$ is the least restrictive class, and $\top$ is the most restrictive. We will denote the security class of an object o as $\underline{o}$. If information is allowed to flow from class $\underline{a}$ to class $\underline{b}$, we will write $\underline{a} \rightarrow \underline{b}$; if not, we will write $\underline{a} \not\mapsto \underline{b}$. The $\rightarrow$ relation is transitive. $\forall C : \bot \rightarrow C$, and $\forall C : C \rightarrow \top$.

The class combining operator $\oplus$ is a least-upper-bound operator on the class lattice such that $\underline{a} \oplus \underline{b}$ is the least restrictive class that encompasses all of the restrictions of $\underline{a}$ and $\underline{b}$. $\forall C : \bot \oplus C = C$, and $\forall C : \top \oplus C = \top$.

3.1 Components, conventions, and data representation

Our architecture features processes, a shared memory, input channels, and output channels.

Each process consists of a set of general-purpose registers, a program-counter, and a special register-stack whose purpose is explained in Section 3.4. The registers of a given process are private and may not be examined by any other process.

The machine word is the unit of information: each slot of memory contains one word, and each register contains one word. A word w consists of a pair $(\underline{w}, w_v)$ where w_v is the data stored in the word, and $\underline{w}$ is the security class of the word. Words are read and written atomically.

Except in special casess, in the remainder of this paper we will not distinguish between a word of memory and a register in our flow-control rules; we will simply speak of a generic word w. When we speak of writing a new value v with class $\underline{v}$ into a word w, we will generally denote the original word contents as w_i and the new word contents as w_{i+1}.

The program counter associated with a process p is tagged with a security class just like any other word; we will generally refer to the security class of p's program counter simply as the security class of p, and will denote it as $\underline{p}$. When an operation updates $\underline{p}$, we will generally denote the original class as $\underline{p_i}$ and the new one as $\underline{p_{i+1}}$.

To ensure that unallocated regions of memory may not be observed by any process, pointers to memory are represented using capabilities denoting specific ranges of memory; hardware checks ensure that no process may access a region of memory for which it does not hold a capability.[1] To disable covert channels based on order-of-allocation, the specific memory address denoted by a capability may not be observed by non-TCB code. A capability is stored as the data component of a word just as any other datum would be.

Each input channel I is tagged with a static (unmodifiable) security class $\underline{I}$. Each output channel O is tagged with a static security class $\underline{O}$.

3.2 Operations at a single security level

Many operations will not adjust the security class of the program counter.

Process p may request a memory from an allocation routine which is part of the TCB; the capability returned by the allocator has security class $\underline{p}$, as do all the words in the segment to which the capability points.

Write attempts never adjust the program counter's security class. We note here that process p may freely write a value v into word w when $\underline{p} = \underline{w} = \underline{v}$; we defer presentation of the general rule for legal writes to Section 3.6.

The primitive operation *readable?*(w) invoked by p returns true if $\underline{w} \rightarrow \underline{p}$ and false otherwise; in other words, it returns true only when p could examine the value or class of w without increasing $\underline{p}$. Invoking *readable?* does not change $\underline{p}$.[2]

[1]In point of fact, any scheme which prevented unallocated memory from being examined by any process would be adequate, e.g. a privileged "top-of-memory register" which was incremented during allocation. Capabilities, however, are the solution to so many other problems as well that we cannot resist their application in this case.

[2]We do not have an operation contrapositive to *readable?*, i.e. one which returns true if and only if $\underline{p} \rightarrow \underline{w}$. While such an operation seems attractive, in conjunction with monotonically-increasing security classes described in Section 3.6 it poses a channel for information leakage, so

Process p may examine $\underline{p}$, and compare it for equality with any $\underline{w}$ for which $readable?(w)$ is true; p may examine the flow-relationships between all classes C in the security lattice for which $C \rightarrow \underline{p}$.

3.3 Computing the security class of a result

Suppose that process p executes the sequential instruction
I_s: `c = OP(a,b)`
where OP is a non-branching operation and the instruction's storage location I_s has security class $\underline{I_s}$. Since OP is non-branching, p's program counter after the operation is independent of the values of a or b; however, it is obviously dependent upon the fact that OP is non-branching! Thus,

$$\underline{p_{i+1}} = \underline{p_i} \oplus \underline{I_s} \qquad (1)$$

Generally, we expect instructions in a given body of code to have the same classification, and thus in most practical cases $\underline{p}$ will not be changed by non-branching instructions.

The class of the operation result is computed from the classes of the operands and the process itself, i.e.

$$\underline{c} = \underline{p} \oplus \underline{a} \oplus \underline{b} \oplus \underline{I_s} \qquad (2)$$

Where the result may actually be *stored* – that is, which register or memory slot is a legal target for c – is determined by the general rule for safe writing in Section 3.6.

Now suppose that process p computes the conditionally-branching instruction
I_b: `if (a) goto TARGET`
In this case, the new value of the program counter is plainly predicated on the value of a in addition to the instruction itself, and thus

$$\underline{p_{i+1}} = \underline{p_i} \oplus \underline{a} \oplus \underline{I_b} \qquad (3)$$

This leads to two conundrums. First, if the program counter always increases in security on every branch, how do we avoid inevitably driving every $\underline{p}$ all the way up to $\top$? And second, since after a branch $\underline{p}$ is now running at an increased security level, how can it use any of the registers which were filled with lower-security values before the branch?

The answer to both problems lies with the register stack.

3.4 The register stack

The register stack is an arbitrarily deep stack of pairs of the form (register-name,value). The security class of each pair is the class of the value. A process may inspect the register stack's contents as if they were any other values, but it may not modify the stack's contents except with the following three primitive operations.

we must omit it.

3.4.1 Primitive operations

The primitive operation `PUSH_RETURN(a)` where a is an address pushes the pair (PC, a') onto the register stack, where the value of a' is the same as that of a, but $\underline{a'} = \underline{p} \oplus \underline{a}$, i.e. $a' = (\underline{p} \oplus \underline{a}, a_v)$. The program counter is incremented and $\underline{p}$ is adjusted according to rule 1 as with any other non-branching operation.

The primitive operation `PUSH_GPR(GPRID,n)` where n is some value pushes the old value o of the specified general-purpose register onto the stack in a pair (GPRID, o); note that $\underline{o}$ is the same on the stack as it was in the register – it is not combined with $\underline{p}$! At the same time as o is placed on the stack, n' is written into the register where the value of n' is the same as that of n, but $\underline{n'} = \underline{p} \oplus \underline{n}$, i.e. $n' = (\underline{p} \oplus \underline{n}, n_v)$.

The primitive operation `POP` pops the top pair (register-name,value) off of the register stack and places the popped value into the register named register-name. In the case of a GPR, this re-creates the state of the register prior to the `PUSH_GPR` operation; in the case of a PC, this sets the program counter to the address specified by the corresponding `PUSH_RETURN`.

More specifically, the effect of `POP` when a PC entry is on top of the stack is to set p PC to the address chosen at the time of the corresponding `PUSH_RETURN.` Since the destination address was chosen at the time of the `PUSH_RETURN`, it contains no information as to the activities of the process since that time, and thus it is safe for p to take on the security class of the popped address even though it may be less strict than the class of the PC value it is replacing!

Thus, we can prevent $\underline{p}$ from monotonically increasing with every conditional branch by simply preceding each such branch with a `PUSH_RETURN`, and terminating each path of the conditional with a `POP`. Let us reiterate that since the post-conditional-branch address is pushed before the conditional branch itself is executed, the popped address contains no information regarding the value branched on, and therefore its class may safely replace whatever security class was imposed on the PC by the conditional branch.

The register stack is similar to the program counter stack of the Data Mark Machine [8] and the unclassified state-restoring mechanisms of [10]

3.4.2 Process initiation, error handling, and termination

A process is started with an empty register stack. A process exits by attempting to POP an empty register stack; there is no other exit mechanism.

A default error handler invokes POP repeatedly until a (PC,a) pair is popped or the stack is empty. A programmer may install other error handlers for various error conditions; regardless, an error handler will always run with same security class, register set, register stack, etc. that the process had before it caused the error. This ensures that p's error remains invisible at any class C for which $\underline{p} \not\mapsto \underline{C}$.

3.5 Operations on I/O channels

Process p may only read values from input channel I if $\underline{p} = \underline{I}$; a read attempt which violates this rule raises an error which is handled at $\underline{p}$. Since the error is predicated on information freely available to p, the success or failure of a read leaks no new information about $\underline{I}$ to p. Values read from I are initially assigned security class $\underline{I}$.

Process p may send value v over output channel O only if $\underline{p} = \underline{O}$ and $\underline{v} \to \underline{p}$ (i.e. for p *readable?*(v) is true.) A write attempt which violates this rule raises an error which is handled at $\underline{p}$. Since the error is predicated on information freely available to p, the success or failure of a read leaks no new information about $\underline{O}$ or v to p.

3.6 A general rule for writing new values into words

In this section we present a rule allowing the class of each word w to monotonically increase over time, and formally show the rule to be safe. We also present an example which attempts to violate secure information flow using implicit flow and show it does not succeed under our rule.

3.6.1 The general rule

Reclassification Rule 1 *Process p may write a value v into word w_i when $\underline{w_i} = \underline{p}$; the write sets $\underline{w_{i+1}} = \underline{p} \oplus \underline{v}$, thus ensuring that regardless of $\underline{v}$, $\underline{p} \to \underline{w_{i+1}}$. An attempt by p to write to a word w_i for which $\underline{w_i} \neq \underline{p}$ signals an error condition which is handled at $\underline{p}$.*

To demonstrate the safety of this rule we must verify two things: first, that p learns nothing about the class or value of v from the success or failure of the write; and second, that p leaks no information about its activity to any process p' for which $\underline{p} \not\mapsto \underline{p'}$.

To show that p learns nothing about the class or value of v, we simply note that the predcondition for the write to succeed is independent of v; failure or success of the write is not predicated on any characteristic of v.

To show that p leaks no information to any process p' for which $\underline{p} \not\mapsto \underline{p'}$, we reason as follows: since $\underline{p} = \underline{w_i}$, $\underline{p} \not\mapsto \underline{p'}$ is equivalent to $\underline{w_i} \not\mapsto \underline{p'}$ which, taken in conjunction with $\underline{p} \to \underline{w_{i+1}}$, implies that $\underline{w_{i+1}} \not\mapsto \underline{p'}$. Resultingly, for p' both *readable?*(w_i) and *readable?*(w_{i+1}) are false; no information is leaked to p' via the *readable?* routine. Correspondingly, p' cannot observe either the original or the new values or classes while remaining at $\underline{p'}$, so it cannot detect the fact that they have changed.

3.6.2 A special-case rule

Arbitrary writes by processes running with class $\perp$ present a special case.

Reclassification Rule 2 *A process p running with $\underline{p} = \perp$ may write any value into any word, regardless of their respective classifications.*

A process running with class $\perp$ embodies no privileged information in its program counter; therefore, regardless of what it writes where, it leaks no privileged information by its actions. Since every write succeeds, the process learns nothing about the values being written or overwritten.

3.6.3 An example: preventing implicit flows

Denning [4] argues that there an intrinsic problem with dynamic bindings, namely that "...a change in an object's class may remove that object from the purview of a user whose clearance no longer permits access to the object. The class change event can thereby be used to leak information..."

Denning cites an example by Fenton [7]:

```
$b$ := $c$ := *false*;
if  $a$ then $c$ := *true*;
if  $c$ then $b$ := *true*;
```

Initially $\underline{p} = \underline{b} = \underline{c} = \perp$, and the constants *true* and *false* have class $\perp$. To illustrate the problem, $\underline{a}$ is any class stricter than $\perp$.

Denning points out that on a system which allows unrestricted writes to a variable as long it they monotonically increase its security class (e.g. the Privacy Restriction Processor [19],) the execution leaks the value of a into b without ever raising the class of b to match that of a.

Our rule, however, is more restrictive and therefore safe: under our rule, the attempt to write to c after branching on a fails[3], since after the branch $\underline{p} = \underline{a}$ and therefore $\underline{p} \neq \underline{a}$. Regardless of the value of a, then, c always remains false; no information is leaked from a.[4]

With respect to a similar piece of code, Myers and Liskov [17] note that while it is easy for a runtime mechanism to detect an improper information flow, if the error causes the program to abort, the program-abort or lack thereof conveys some information. Since our architecture does not allow a program to exit/abort, and error handlers run at the same security level as the process that invokes them, our architecture is not subject to this problem. [5]

3.7 Authenticated declassification

There will undoubtedly be occasions when a user may legitimately wish to explicitly assign a weaker security class to a particular datum; for instance, a computation based on classified data might yield a result suitable for public dissemination.

To support this goal, we add the notion of authenticated principals to our architecture. Each process runs on behalf of a specific principal. A principal has two powers: the power to perform certain declassifications, and the power to replace itself with certain other principals.

3.7.1 Declassification

A principal u may be authorized to perform certain declassifications; it is up to the trusted software implementing the lattice and class structures to define the declassifications allowed by each principal.

For instance, in a linear lattice, a particular principal might be authorized to declassify data from "Secret" to "Classified", but not permitted to declassify "Classified" to "Unclassified."

As another example, in Myers and Liskov's labeling scheme [17, 18], each security class is defined by nested sets of principals; a principal may, for instance, perform a declassification by removing itself from the list of a datum's owners. [6]

[3]The error handler invoked by the write failure could report the incident over an output channel O where $\underline{a} \rightarrow \underline{O}$ before allowing execution to continue through the program; this would enable debugging at a later date.

[4]As a general note, to express any if-statement "correctly" on our architecture – that is, in such a fashion that after the if-statement, $\underline{p}$ is the same as it was before the if-statement – one must precede each if-statement with a PUSH_RETURN operation, and terminate each if-statement with a POP operation.

[5]It is worth noting that a program could conditionally fail to terminate based on a secure value. Neither static nor dynamic schemes for verifying secure information flow can prevent this type of information leakage; additional mechanisms and restrictions are required. (Nontermination is just an extreme form of covert information transfer via timing channels.)

[6]The extended labeling scheme in [18] scheme is not actually a lattice; equivalence classes on the labels form a lattice. In spite of this, their declassification operations actually map in a one-to-one fashion with specific arcs in the lattice.

The act of declassification obviously reveals some information about both the declassified value and the process performing the declassification. Since a process' state could depend on information that a principal is not allowed to reveal, we must constrain when declassification may be performed. We define the following rule for safe declassification:

Reclassification Rule 3 *Each principal u is associated with a (possibly empty) set of pairs of classes $F_u = \{(\underline{a_{u1}}, \underline{b_{u1}}), (\underline{a_{u1}}, \underline{b_{u1}}), ...\}$.*

Process p running with u's authority u may change the class of word w to C if and only if $\underline{p} = \underline{w}$ and
$\exists(\underline{a_{ui}}, \underline{b_{ui}}) \in F_u$:
$\underline{w} = \underline{b_{ui}}$*and*$C = \underline{a_{ui}}$

p may change its own class under the same criteria, which simplify in this case to:
$\exists \underline{a_{ui}} \rightarrow \underline{b_{ui}} \in F_u$:
$\underline{p} = \underline{b_{ui}}$*and*$C = \underline{a_{ui}}$

Since this rule only allows a process to declassify a value when it is authorized to declassify the program counter in the same way, it reveals no information about process state which the principal is not authorized to reveal. This rule is the only rule in our architecture which allows a process p to write a value with a security classification C for which it is not guaranteed that $\underline{p} \rightarrow C$.

Note that while this rule is correct, an implementation might choose not to instantiate the set F_u explicitly, as it might be prohibitively large. For instance, in the Myers/Liskov labeling scheme, while the declassification operation "remove this principal from the list of owners" is simple to define and implement, it expresses a huge number of potential transitions – F_u must contain one pair for every possible list of owners containing the principal.

3.7.2 Principal replacement

A principal may be authorized to replace itself with another principal as the principal upon whose behalf the current process is running (the authority principal.) We implement this with a hardware-supported "role" stack, the top element of which is the authority principal.

Using this mechanism, we can implement Role-Based Access Control ([9]) – a principal may take on any of a variety of "roles" (i.e. other principals), without ever having the access-rights of more than one of those roles at a time.

The role stack is also quite useful for logging exactly which principals performed what actions in what roles

("Jeremy Brown acting as a Research Assistant acting as a Graduate Student printed 50 pages.")

Note that whenever a PC is popped from the resgister stack, the role stack must be repeatedly popped back to the same point it was at when the PC was first pushed.

4 Hash Execution Unit

The Hash Execution Unit is a generalization of the N-way associative data cache found in most simple modern RISC processors. To review, one approach to the construction of simple caches is to place them in parallel with the conventional arithmetic unit. Data values read from the register file are used both to drive the inputs of the arithmetic unit, and as a source of the cache address and write data for the data cache. Results from the cache are written back to the register file in the same way as an arithmetic unit result. Thus, loads and stores to the data cache can be performed with the same basic timing and control mechanisms as any other instruction, with the exception of cache miss operations.

4.1 Standard Cache Design

A typical implementation of a standard data cache is shown in figure 1. One register file read generates a load/store address. The load/store address is converted to a cache line address. In the simplest case this can be done by simply dropping high order bits; more complex schemes involve hashing the load address to produce a cache line address.

The cache line address is used to reference the data cache, fetching the keys and the data. The keys are compared with the load address (or a high order portion of it, if hashing has not been used). If the key and load addresses match, the data returned by the data cache is valid, and that data is sent back to the register file as the result of the load operation. A mismatch causes a cache miss, followed by a slow main memory reference and cache reload operation.

Several such data cache units can operate in parallel on the load address; if any find valid data, the cache hits. With N such units, the data cache is termed an N way associative cache, because for any given load address, there are N possible cache locations in which it could be stored.

Store operations perform a similar tag and data read operation in the data cache. A cache hit activates a subsequent data write operation in that way of the cache, which completes during register file read of the subsequent instruction.

4.2 Extension of the Way Hardware to Value Caches

Now, consider a generalization to the standard data cache. Another type of cache commonly considered is a value cache, which caches the values of functional subroutines, eliminating the necessity to recompute the results of commonly used arguments.

A simple extension of the data cache could handle the value cache as well. Instead of cacheing data based on a load address, the argument (or arguments) can be hashed, and the hash result used to index into the N way associative cache. A hit, determined by an exact match of the argument(s) to the key, results in the use of the value part of the cached data in the matched way as the function value.

4.3 Type computations

Similarly, one could combine dynamic type information (or other data tag information such as units) from multiple arguments using a hash function; perform an N way associative lookup; perform an exact key comparison; and dynamically assign a type to the data result. Optionally (and importantly) a trap can occur on the occurance of rare or improper type combination.

As shown in figure 2, the important mechanisms here are masking units to mask off the type tag field from the register words, a hash combiner, a tag test unit for comparison of the masked data with the cache key returned, and a tag insertion mechanism for reinsertion of the tag result into the ALU register result.

With this mechanism, the type information from multiple operands can be extracted, the combination looked up, the result optionally trapped, and the combined type automatically inserted into the result in a cycle time comparable to the time for a normal data cache load. Typically, this load can occur in parallel with normal ALU operations.

Note that the location and size of the type tag data within the data word is now a programmed choice, rather than a hardware fixed feature, because of the masked extraction and insertion operations.

Multiple such hash operations could be performed in parallel on a single set of data word fetch from the processor register file. The identical units could be, alternatively, used for a larger data cache with the masking and insertion logic set appropriately.

We term this basic structure a Hash Execution Unit (HEX unit), because it performs simple (type, unit, lookup) computations, or more complex (value cache) operations quickly in the common cases.

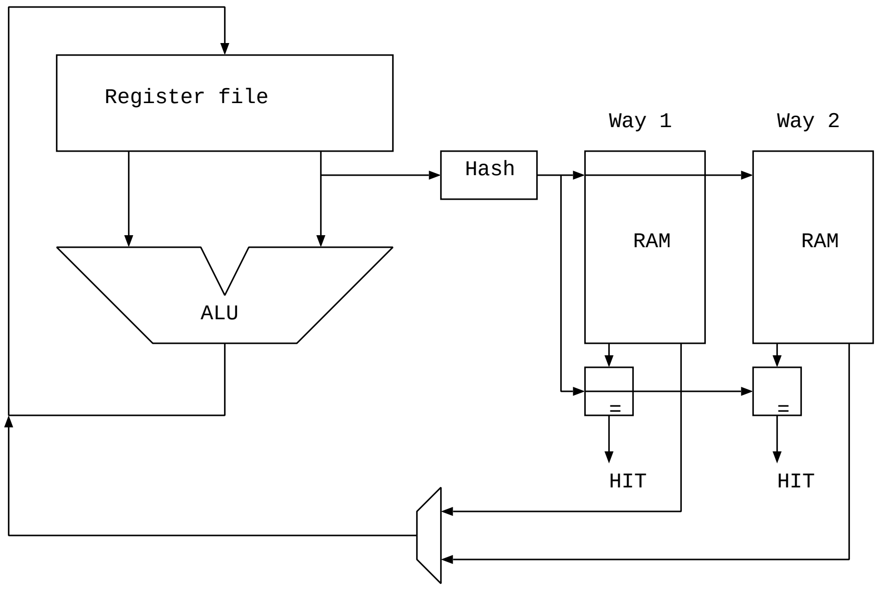

Figure 1: A typical data cache implementation.

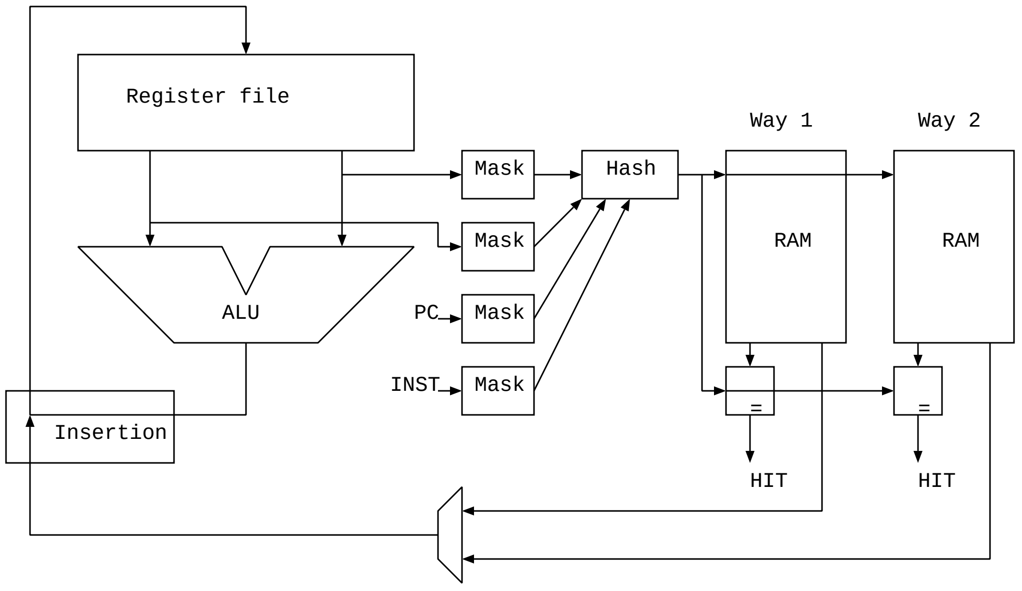

Figure 2: A hash execution (HEX) unit.

4.4 Using the HEX Unit for Security Checks

The HEX unit can be augmented slightly to implement the high performance inner loop of the dynamic security typing. The security tags from each operand are extracted by the data mask unit. Additionally, security tags associated with the program counter, and with the instruction being executed are similarly extracted. The extracted security tags are hashed together, and a portion of the resulting hashed value used to look up a security tag result. The cache keys are compared with each masked security tag (operand, PC, Instruction) to assure that the correct cache entry has been located. Under normal circumstances, the cache will hold the appropriate security level tag for the instruction result, which is then masked and inserted into the security tag field of the data written to the register file. Rarely, the cache will miss, resulting in a trap, the computation of the correct security tag being added to the cache, and the processor restarted.

The mechanism outlined provides us with a flexible, run time reconfigurable technique for quickly performing relatively complex manipulations on tag data associated with normal instruction execution. The HEX unit can be applied to make data tags, units, value cache results, object oriented method dispatches, and the security tags discussed here a practical and efficient tool in modern architectures.

5 Implementation of the model

Although the HEX unit of the previous section is the cornerstone of our proposed implementation – HEX units enable rapid computation of Equations 1, 2, and 3 in the normal case – some additional mechanism is required.

5.1 The register stack

We simulate an arbitrarily deep stack with a hardware stack of finite depth; a set of trusted routines spill the stack to memory when the hardware overflows. This is very similar to the stack implementation in the Lisp Machine [15], and also to the register windows schemes used to simulate infinite numbers of registers in the SPARC and Itanium microprocessor architectures.

5.2 Computing legal overwrites

The general rule for overwriting words, Rule 1, requires that the security class of the process p be identical to the class of the word w being overwritten. Thus, in general, we will need to examine a word before overwriting it.

When w is in memory, we avoid delays due to having to examine a word before overwriting it as follows: when we look up w in the D-cache with intent to overwrite it, we use a key consisting of both w's address and p's security bits. If address and security bits don't both match, the cache lookup fails; separate equality outputs for the address and security-bit comparisons indicate whether the failure was due to a D-cache miss or a security fault. The only slowdown due to the security check is in the case of a legitimate D-cache miss: the system must wait for the old word to be transferred to the D-cache in order to perform the security check prior to the write.[7]

When w is in a register, we handle the situation differently. In modern superscalar processors, out-of-order execution requires register renaming. When issued instruction I specifies that its result will be written into register R, in actuality a new backing store slot S_n is allocated and the name R is re-bound to point to S_n instead of its previous slot S_o. I may not produce the value intended for R for several cycles, during which time logically-subsequent instructions dependent on I's result are stalled.

In this setting, Rule 1 requires that $\underline{p_i} = \underline{S_o}$, where p_i is the PC at the time I is issued; otherwise, I must fail to write into R and an error must be raised. Due to out-of-order execution, S_o may not contain a value when I issues; thus, while I may (and should) speculatively execute, it must not actually retire (commit) until the value of S_o has been fixed.

Based on these observations, we can implement the security check of Rule 1 in two simple steps. At the time of issue, the entry for I records slot S_n as the backing store for R, and also records S_o as S_n's predecessor slot. At the time of retirement, $\underline{S_o}$ is compared to $\underline{p_i}$. If they are equal retirement succeeds; otherwise, retirement fails and an error is raised (thus aborting all instructions logically-subsequent to I.[8]) The only slowdown due to this mechanism ocurs when an instruction's retirement is delayed waiting for a value to be produced by a logically-precedent instruction; since retirement happens at the end of the processor pipeline, however, this situation will generally be extremely rare.

6 Conclusions

In this memo we have made several contributions. On the theoretical side, we have described an abstract architecture which dynamically ensures secure information flow with respect to a security lattice; we have shown that it does not leak information through implicit channels; and

[7]Actually, a system with adequate speculation support could simply perform the write speculatively, and commit only when the "overwritten" value arrives from main memory and the permissions-check has been performed.

[8] Imprecise exceptions are fine as long as POP instructions act as a barrier with respect to exceptions.

we have defined a general rule allowing authorized principals to perform specific declassifications without additional, unintentional information leakage. The architecture provides a set of features which collectively make it an improvement on all predecessors known to us:

- Precise (non-conservative) enforcement of a lattice-based security policy
- Dynamic security tags on a per-word basis
- Safe overwriting of values with values of different security levels
- Safe declassification of a program counter when its dependency on classified data ends (i.e. the PC is not stuck at the security "high-water mark")
- A small TCB

On the practical side, we have described mechanisms enabling the implementation of our dynamically secure architecture. Of particular importance is the hash execution unit, or HEX unit, a generalization of the N-way associative data cache which can be applied to make data tags, units, value cache results, object oriented method dispatches, and the security tags discussed here a practical and efficient tool in modern architectures.

We close by enumerating the elements of the trusted computing base for our suggested implementation. The TCB consists of the following policy-independent components:

- A processor featuring:
  - HEX units
  - general purpose registers
  - per-word security class annotation
  - the register stack
- trap handlers to securely swap processes and spill the register stack
- memory management routines for allocation and garbage collection

...and the following policy-dependent components:

- specification of a security class hierarchy
- specification of a set of principals
- software computing $\rightarrow$ (flows-to relation) and $\oplus$ (class combining operator) on the specified lattice
- software computing declassification authorizations for each principal

References

[1] Jeremy Brown, J.P. Grossman, Andrew Huang, and Tom Knight. A capability representation with embedded address and nearly-exact object bounds. Technical Report ARIES-TM-005, Project Aries, MIT AI Laboratory, April 2000.

[2] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–27, October 1994.

[3] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 141–60, University of Texas at Austin, 1975.

[4] D. E. Denning. A lattice model of secure information flow. *Communications of the Association of Computing Machinery*, 1976.

[5] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[6] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–12, July 1974.

[7] J. Fenton. Information protection systems, 1973.

[8] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[9] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of the National Computer Security Conference*, 1992.

[10] Israel Gat and Harry J. Saal. Memoryless execution: A programmer's viewpoint. *Software – Practice and Experience*, 6:463–471, 1976.

[11] Anita K. Jones and Richard J. Lipton. The enforcement of security policies for computation. In *Symposium on Operating Systems Principles*, pages 197–206, 1975.

[12] Butler W. Lampson. A note on the confinement problem. *Communications of the A.C.M.*, 16(10):613–615, 1973.

[13] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984.

[14] Jonathan K. Millen. Security kernel validation in practice. *Communications of the ACM*, 19(5):243–250, 1976.

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

[16] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[17] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.

[18] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

[19] L. Rotenberg. Making computers keep secrets. Technical Report MIT/LCS/TR-115, MIT Laboratory for Computer Science, 1974.

[20] M. Schroeder. Engineering a security kernel for multics, 1975.

[21] K. G. Walter, S. I. Schaen, W. F. Ogden, W. C. Rounds, D. G. Shumway, D. D. Schaeffer, K. J. Biba, F. T. Bradshaw, S. R. Ames, and J. M. Gilligan. Structured specification of a security kernel. In *International Conference on Reliable Software*, pages 285–293, 1975.

[22] Clark Weissman. Security controls in the ADEPT-50 time-sharing system. *Proc. FJCC, AFIPS*, 35, 1969.

[23] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, 1974.