C-level Tag-Based Security Monitoring

Andrew Tolmach, Portland State Univ
Sean Anderson, Portland State Univ
Catalin Hritcu, INRIA
Benjamin Pierce, UPenn

HOPE project for DARPA SSITH program
(SAFE project for DARPA CRASH program)
(a cast of 1000s)

Dagstuhl Seminar on Secure Compilation, May 2018

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0011.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

Idea in a nutshell

- Many useful safety "μpolicies" can be enforced by reference monitors based on HW metadata tags
  - E.g. info flow control, memory safety, control-flow integrity,…
  - Tag-based policies are specified and enforced at level of HW ISA
- Can we harness tag hardware for efficient enforcement of safety properties defined at level of C code?
  - Add reference monitor points to C semantics
  - Customize by per-system or per-program rules
  - Compile to ISA-level tags for runtime enforcement
- Some possible uses: fine-grained IFC, compartment enforcement, access control, trapping C undefined behaviors

Outline

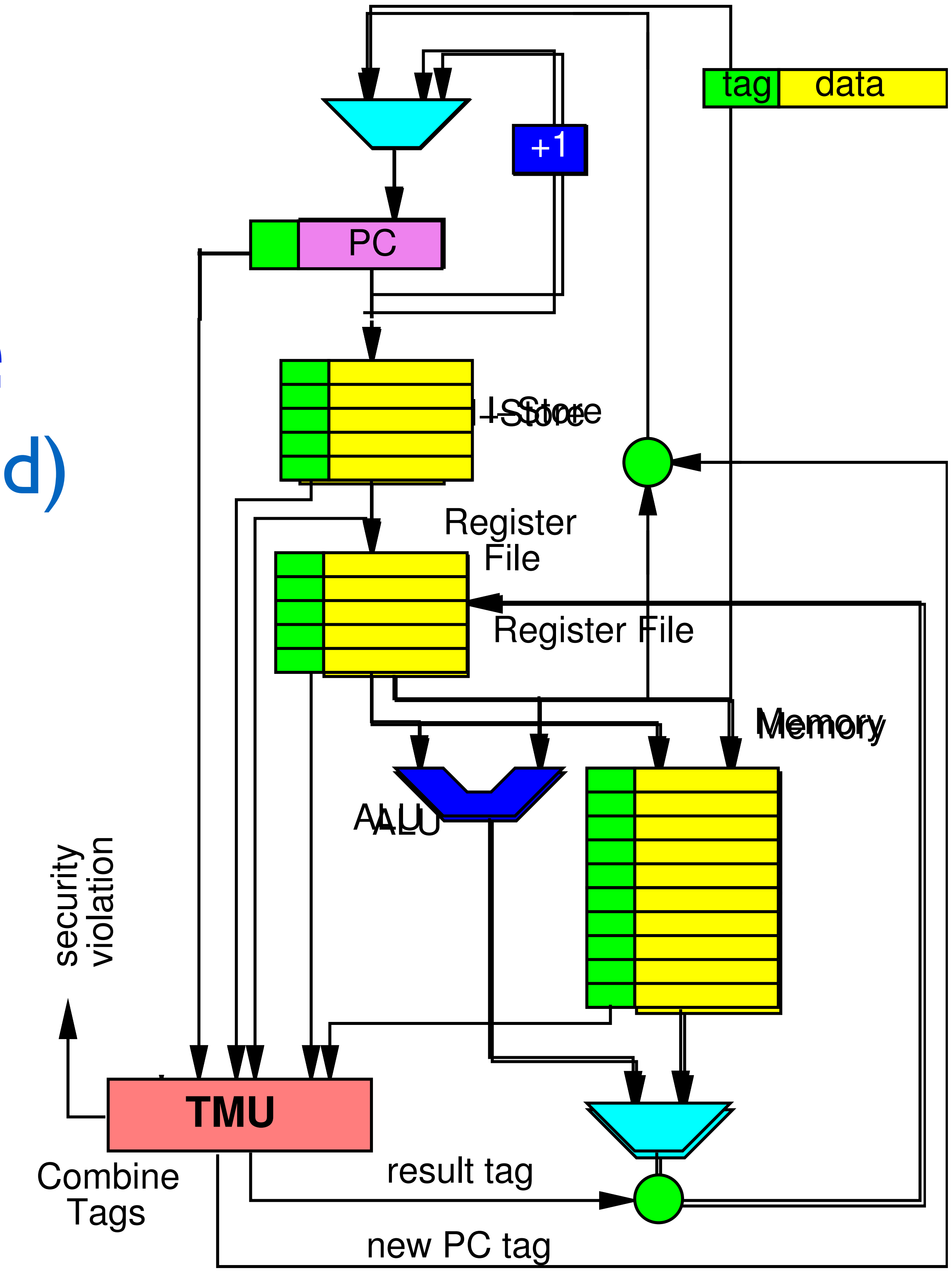
- SAFE/PUMP/PIPE tag hardware architecture
  - Example ISA-level μpolicies
- C-level monitoring
  - Applications
- Compilation scheme
- Open questions

Tag Architecture
(simplified)

Typical RISC CPU

+ large tag on every word

+ tag management unit (rule cache)



Tag Management Unit

acts like a cache

tag is arbitrary bit vector,
usually address
of a data structure

key

(opcode,
pc-tag,
instruction-tag,
register-operand1-tag,
register-operand2-tag,
memory-operand-tag)

→

result

(new-pc-tag,
result-tag)

if key is not present, control traps to tag miss handler

Tag Miss Handler

- Ordinary machine code that lives in privileged code space or on a special co-processor
- Takes missing key as input
- Executes tagging decision algorithm
  - Hardware is completely independent of this algorithm
- EITHER generates result tags (& stores in TMU cache)
  - Instruction that faulted can then complete
- OR discovers security violation and fail-stops the process (or whole processor)

Anatomy of a policy

- Set of tags for labeling registers, memory, PC
  - Can be discrete symbols, numbers, or addresses pointing to arbitrary data structures
- Rules for checking and propagating tags as the machine executes each instruction
  - Rules are just arbitrary code and may maintain persistent internal state
  - But they should be functional (given input tags always produce same output tag)
- Initial configuration
  - Tags on memory contents; tag rule state

Ex: Dynamic Info Flow Control

- Goal: prevent leakage of high-security info
- Tags = Security labels from a lattice
  - Initial memory values and pointers are labelled
  - PC carries "current" label
- Rules:
  - Instructions that move values propagate labels
  - Binary operations compute lattice join of labels
  - Conditional jumps raise PC label level
  - "No sensitive upgrade" — stores are prevented if pointer or PC is higher than old value

Ex: Heap Memory Safety

- Goal: prevent heap buffer overflows
- In-register tags VTag = NotPtr | Ptr Region#
- In-memory tags MTag = (Region#,VTag)
  - Each call to malloc generates a fresh integer region# tag R
  - Newly allocated memory cell values are tagged with (R,…)
  - Pointer to new region is tagged Ptr R
- Rules:
  - Load and store instructions check that address pointer is tagged Ptr R and the referenced memory cell value is tagged (Ptr R,…)
  - Pointer arithmetic instructions preserve Ptr R tags

Ex: Control-flow integrity

- Goal: make sure that every executed jump instruction follows a specified control-flow graph edge
  - e.g. produced by the compiler
- Tag = Data | Code *addr* | Code ⊥
  - Code at location *addr* is given tag Code *addr* iff it is the source or target of a legitimate CFG edge
- Rules:
  - Normally, PC tag is Code ⊥
  - On a jump, check that tag of jumping instruction has the form Code *addr*, and copy it into PC tag
  - If PC tag is Code *saddr* and current instruction is tagged Code *taddr*, require that (*saddr* →*taddr*) is an edge in the CFG

Ex: Dynamic Compartments

- Idea: Divide process memory into set of disjoint compartments which are protected from each other
  - Code in one compartment can jump or write to other compartments only at a pre-defined set of addresses (an interface)
- Tags = sets of compartment IDs
  - PC is tagged with {current compartment}
  - Each memory location is tagged with set of compartments that can validly access it
- Rules:
  - On each write and after each branch, compare PC tag with tag of memory location being written or executed

Composing policies

- Policies are easily composed when they are essentially orthogonal
  - e.g. A = Memory safety and B = CFI
  - Make tags be pointers to pairs (Atag,Btag)
  - Operations are allowed only if both policies say OK
  - When policies interact, things are not so simple…

Tag system performance

- Current designs cost ~100% extra area and ~50% extra power
- Runtime overhead depends on cache hit rate
  - Varies widely for different choices of policies and program patterns
  - Simulations using SPEC2006 benchmarks enforcing a fairly rich composite policy show <10% added run time for most programs
- Keeping number of "live" tags low is essential

What could be better?

- Policies are tedious to specify at the ISA level
- Some policies are impractical on unstructured code
  - e.g. simple IFC induces label creep
- Inconvenient to express per-program policies
- Although machine-level checking seems very strong, in practice we rely on compiler tool chain to give us good initial tags
- So, what if we include compiler in TCB and try to express policies in a "C-level" way?

C-level tagging

- Express tagging policy at level of C expression operators and control structures, rather than of machine instructions
- Attach tags to C “program counter,” values, memory locations (globals, malloc’ed heap records, …), functions, …
- Tag transfer functions are invoked at fixed set of points in C execution semantics
  - instead of at each instruction
  - similar to aspect-oriented programming “advice” points

What it looks like

- To use tagged C for a specific policy:
  - define vocabulary of tags and tag operators (just as for machine-level tagging)
  - instantiate all the transfer functions.
- To use it for a specific C program:
  - specify tag information for (at least) link-level C entities including functions and globals
  - ideally we do not change the C code
- Policies can be specified per system, per module or even per function

Example

One clause in C statement semantics (monadic style)

```
| IfS e s1 s2 =>
    v@v_tag <- eval e;
    old_pc_tag <- get_pc_tag;
    new_pc_tag <- ifSplitT v_tag old_pc_tag;
    set_pc_tag new_pc_tag;
    if v then exec s1 else exec s2;
    new_pc_tag <- ifJoinT v_tag old_pc_tag new_pc_tag;
    set_pc_tag new_pc_tag
```

this is designed once and for all

An instantiation for IFC tags:

```
Tag = LOW | HIGH

ifSplitT v_tag old_pc_tag := ret(v_tag \/ old_pc_tag)
ifJoinT v_tag old_pc_tag new_pc_tag := ret old_pc_tag
```

this is written once for each policy (in some suitable policy language)

Example

One clause in C expression semantics (monadic style)

```
| PlusE e1 e2 =>
  v1@t1 <- eval e1;
  v2@t2 <- eval e2;
  t <- plusT t1 t2;
  ret (v1+v2)@t
```

this is designed once and for all

An instantiation for IFC tags:

```
Tag = LOW | HIGH

plusT v1_tag v2_tag := ret (v1_tag \/ v2_tag)
```

these are written once for each policy

An instantiation for memory safety tags:

```
Tag = NotPtr | Ptr region

plusT v1_tag v2_tag :=
   match v1_tag,v2_tag with
   | NotPtr, NotPtr => retT NotPtr
   | Ptr a,  NotPtr => retT (Ptr a)
   | NotPtr, Ptr a  => retT (Ptr a)
   | _,      _      => failT
   end.
```

What is it good for?

- Can express policies that depend on structured control flow, such as fine-grained IFC within a procedure
- Function-level tags are natural way to enforce access control on resources
- Can express a variety of compartmentalization schemes
- Selective detection of C undefined behaviors, e.g. for pointer safety
  - Assume very permissive compiler and use tag policies to enforce desired level of standards compliance

Compilation

- How to go from tagged C to tagged machine code?
- Basic idea: specially tag the instructions in the generated code to indicate their C-level role
  - Machine-level transfer functions for these special tags are built directly from the C-level transfer function
  - Probably must modify compiler (to generate appropriately tagged instructions)
  - Compilation scheme is independent of policy (although policy-specific schemes might give better code)

Compilation Example

One clause in C expression semantics

```
| PlusE e1 e2 =>
  v1@t1 <- eval e1;
  v2@t2 <- eval e2;
  t <- plusT t1 t2;
  ret (v1+v2)@t
```

Corresponding clause in C expression compiler

```
compileExp (e: Exp) : (reg * list Inst) =
…
| PlusE e1 e2 =>
  let (r1,code1) := compileExp e1 in
  let (r2,code2) := compileExp e2 in
  let r := fresh_reg() in
  (r, code1 ++
      code2 ++
      [MovI r1 r @ Icopy,
       AddI r2 r @ Iplus])
```

Machine-level rules (defined once and for all)

```
MovI,tpc,Icopy,t1,_,_,_ -> tpc,t1,_
AddI,tpc,Iplus,t1,t2,_,_-> tpc,plusT t1 t2,_
```

```
| IfS e s1 s2 =>
   let (r,is) := compileExp e in
   let rt := fresh_reg() in
   let rt' := fresh_reg() in
   let is1 := compileStm s1 in
   let is2 := compileStm s2 in
   is ++
   getpctag rt ++
   combine r rt IifSplitT rt’ ++
   setpctag rt’ ++
   [BifI r (length is2+1) @ Idontcare] ++
   is2 ++
   [BrI (length is1) @ Idontcare] ++
   is1 ++
   combine rt rt’ IifJoinT rt ++
   setpctag rt
```

```
getpctag r := [ConstI 0 r @ Igetpctag]
setpctag r := [ConstI 0 r @ Isetpctag]
combine r1 r2 I r3 := [MovI r1 r3 @ Icopy] ++
                      [MovI r2 r3 @ I]
```

Machine-level rules (defined once and for all)

```
ConstI,tpc,Igetpctag,_,_,_,_      -> tpc,tpc,_
ConstI,_,Isetpctag,new_tpc,_,_,_  -> new_tpc,new_tpc,_
MovI,tpc,Icopy,t1,_,_,_           -> tpc,t1,_
MovI,tpc_,IifsplitT,t1,t2,_       -> tpc,ifSplitT tpc t1 t2
MovI,tpc,IifjointT,t1,t2,_        -> tpc,ifJoinT tpc t1 t2
```

Attacker Model and TCB

- Initial assumption is that all object code is produced by this compiler, and we rely on correctness of compiler and linker/loader
  - Trust; ideally verify.
- If our C code might have undefined behaviors, we may wish to guard against these using a combination of C-level policies and some "built-in" ISA-level policies, e.g. RWX permissions, correct call bracketing, CFI, …
- Those ISA-level policies will also be needed if we want to link against code of unknown provenance

Some Open Questions

- How much should we modify C code?
  - e.g. tags on parameters, local variables?
  - is it realistic to deal with dusty decks anyhow?
- Could we use software enforcement for C level tags?
- What is a good C component specification scheme?
- Who is the "security engineer" responsible for applying policies?