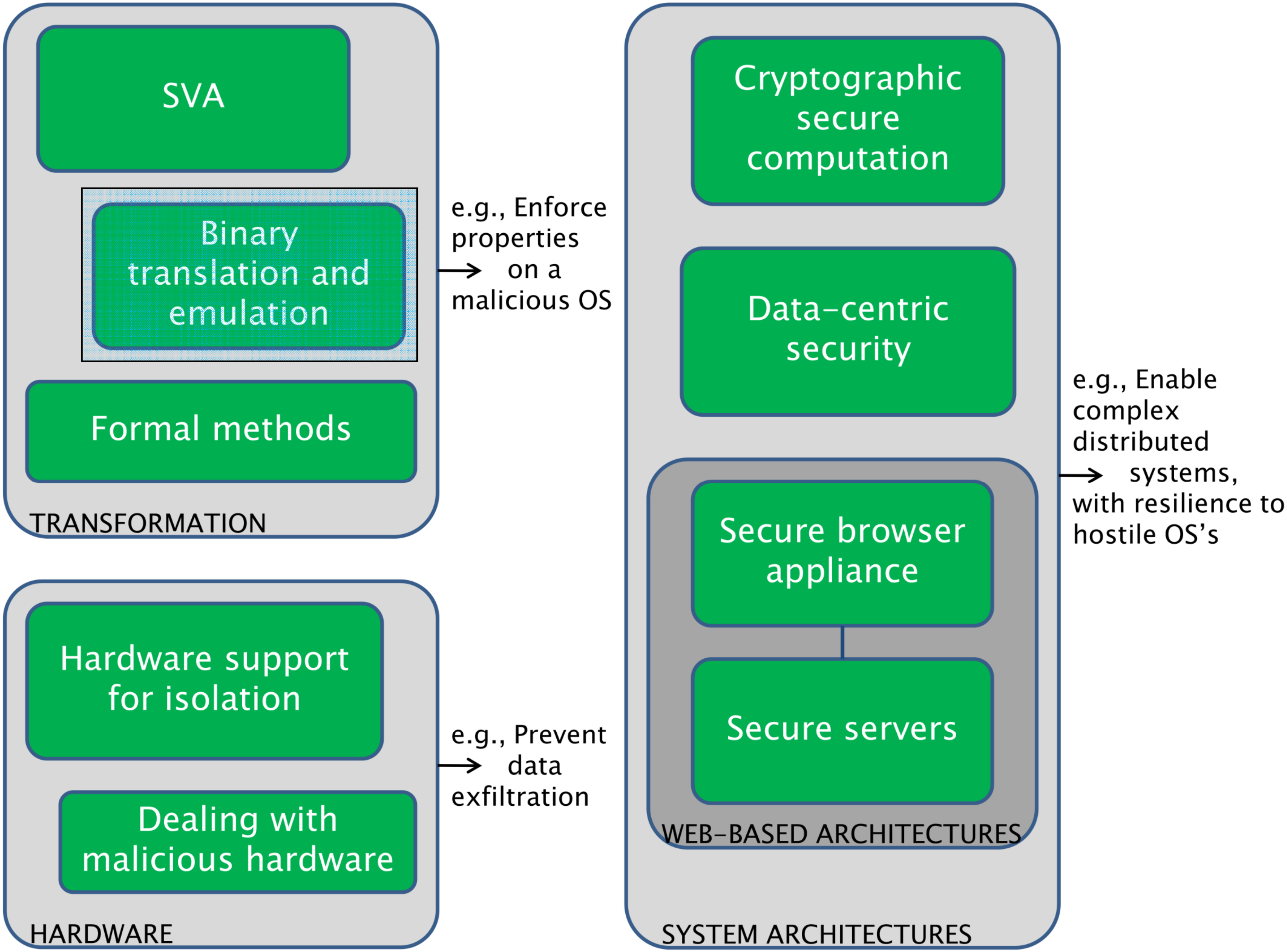
Binary Analysis and Rewriting

Arvind Ayyangar
Niranjan Hasabnis
Rui Qiao
Alireza Saberi
Mingwei Zhang
R. Sekar
Stony Brook University


SVA
Binary translation and emulation
Formal methods
TRANSFORMATION
e.g., Enforce properties on a malicious OS
Hardware support for isolation
Dealing with malicious hardware
HARDWARE
e.g., Prevent data exfiltration
Cryptographic secure computation
Data-centric security
Secure browser appliance
Secure servers
WEB-BASED ARCHITECTURES
SYSTEM ARCHITECTURES
e.g., Enable complex distributed systems, with resilience to hostile OS’s

Binary Translation

- A popular approach for implementing virtual machine monitors (VMMs)
  - Examples: QEMU, VMWare, …
  - Provide foundation to secure applications from hostile OS
- Maximizes applicability
  - Can work with arbitrary OSes and applications available only in binary form.

Motivation (Why are we doing this?)

- Existing binary translators not well-suited for enforcing many important security properties
  - Information flow, control-flow integrity, object-granularity memory safety, XFI, …
    - Some properties ill-suited to pure dynamic enforcement
      - May require maintenance of some global invariants
    - Most of them incur very high overheads (4x to 10x slowdown)
      - Start-up times are even worse
- Suboptimal register and memory use
  - Excessive register spills and memory accesses surrounding instrumentation

Research Problems (What are the hard problems?)

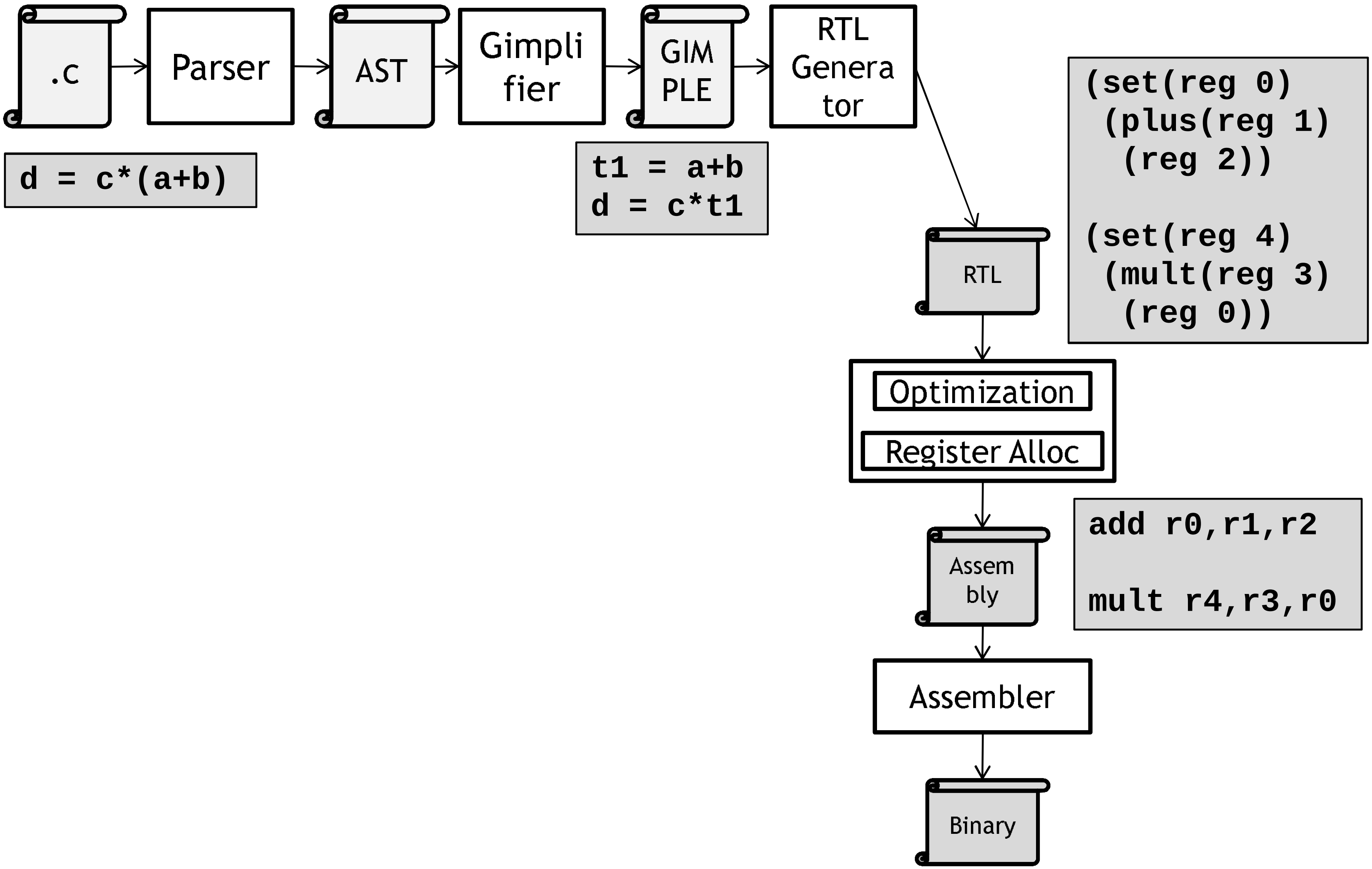
- Static analysis and optimization techniques for efficient enforcement of security properties
  - Optimization of original + instrumentation code
- Decoupling analysis and instrumentation from specifics of an instruction-set architecture (ISA)
  - Ideally, a single implementation across X86, ARM, MIPS, SPARC, PowerPC, etc.
  - Each ISA can be quite complex
    - X86: 1100+ instructions described by a 1500+ page manual

Our Approach (How do we proceed to solve them?)

- Develop novel compiler-based methods for overcome the drawbacks of today’s techniques
  - Leverage compiler infrastructure for efficient instrumentation and retargetability
    - Architecture-independent binary instrumentation using compiler machine descriptions
  - Robust, scalable static analysis of low-level code
    - Provides crucial missing pieces to complete the loop in compiler-based instrumentation

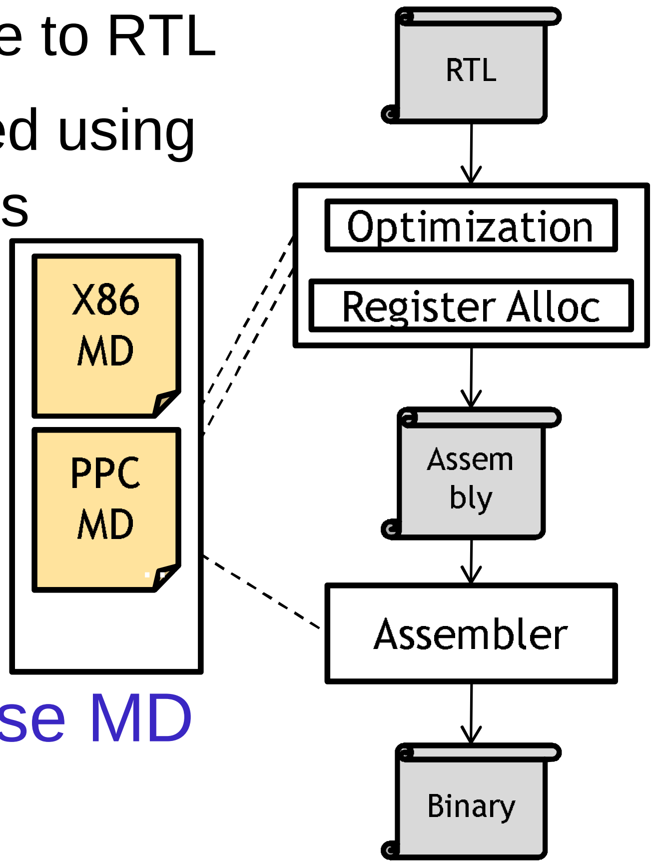
Architecture-Neutral Binary Instrumentation from Machine Descriptions

Architecture of GCC

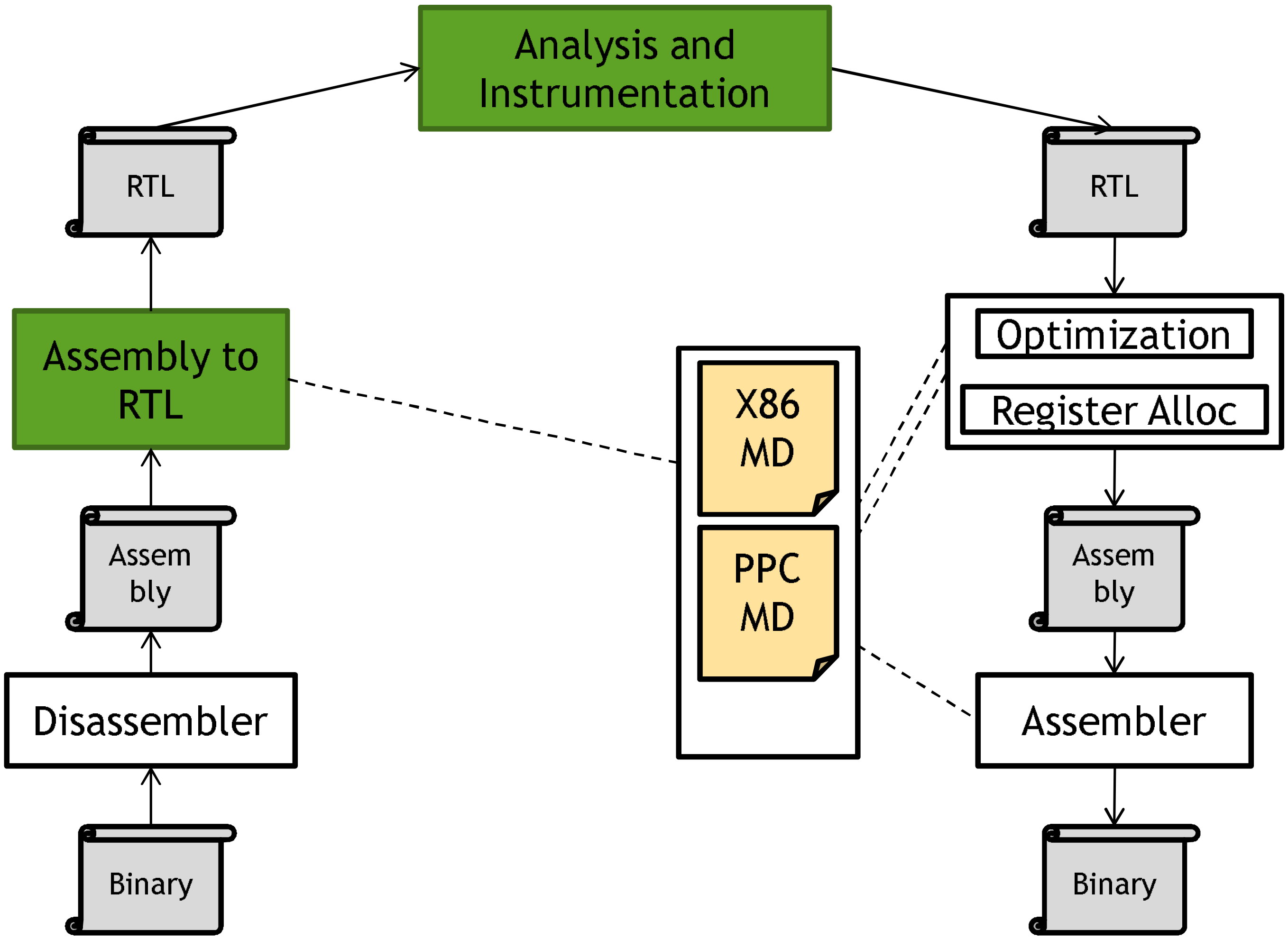


Leveraging Compiler Infrastructure

- Leverage architecture-independent code generators [Davidson and Fraser 1984] (and many others)
  - Translate intermediate code to RTL
  - Quality of final code ensured using extensive RTL optimizations
  - Final step uses machine descriptions (MD)
    - rules that map RTL snippets to assembly
    - Pattern-driven process
- Key Question: Can we use MD rules in reverse?



Compiler-based binary instrumentation



Benefits of using compiler infrastructure

- Eliminate error-prone step of developing instruction semantics
  - Specifications used in compilers extensively tested
    - Or else the compiler would generate incorrect code!
- Reuse back-end optimization phases for optimizing instrumented code
  - GCC runs about 40 different optimizations on RTL …
- Retargetability!

Example

```
(define_insn "*strsetqi_rex_1"
  [(set (mem:QI (match_operand:DI 1 "register_operand" "0"))
        (match_operand:QI 2 "register_operand" "a"))
   (set (match_operand:DI 0 "register_operand" "=D")
        (plus:DI (match_dup 1) (const_int 1)))]
  "!TARGET_64BIT"
  "stosb"
  )
```

- A few other MDs produce stosb
  - In this case, it moves EAX to ES:EDI, increments EDI

- General form: RTL/Cond → ASM
- RTL uses `match_operand` to specify *predicates* and **constraints** on operands, and `size annotations` to specify operand sizes

Example

```
(define_insn "*pushsi2"
  [(set (match_operand:SI 0 “push_operand” “=<”)
     (match_operand:SI 1 “general_no_elim_operand” “ri*m”))]
  "!TARGET_64BIT"
  “push %1"
  )
```

- General form: RTL/Cond → ASM
- RTL uses `match_operand` to specify *predicates* and **constraints** on operands, and `size annotations` to specify operand sizes

So, what is the catch?

- MD rules are meant to be used to translate RTL to ASM. Using them in reverse can be difficult:
  - Is the mapping invertible?
    - Predicates, constraints and conditions are ultimately checked by C-code that can be used only in forward direction
      - Map ASM to RTL --- this requires just the operands to be mapped, a simple process
      - With this mapping, use MD in forward direction to check if same ASM is generated. Otherwise, rule is not applicable
    - What if RTL operands are missing in ASM?
      - Use operand constraints to infer extra operands, or try all possibilities
    - ASM can be a piece of C-code that generates assembly
      - Solution: symbolic execution of problem code

So, what is the catch? (Continued)

- MD rules are meant to be used to translate RTL to ASM. Using them in reverse can be difficult:
  - Is it sound?
    - What if ASM depends on conditions unspecified in RTL?
    - Or, ASM does “more” than the RTL?
  - We can formally establish this
    - for instructions covered by the compiler’s MD

Status

- GCC's x86 MD is 34K lines
  - A few instructions are not covered by GCC
    - Just 3 of these used in complex apps (LLVM-compiled)
      - Firefox (5M instructions), GIMP (1.4M instructions)
  - We can so far handle about 50% of the x86 MD
- Implementation status:
  - Non-trivial applications can be handled
    - Some SPEC INT executables (e.g., gzip) can be disassembled, reassembled and run
  - Simple instrumentations
    - Null-pointer check (bzip2)

Related Work

- Efforts to simplify instruction semantics specification
  - CTL (UQBT), TSL (CodeSurfer), VEX (Valgrind), BIL (BAP), Harvard team's DSLs, …
  - We pursue a complementary approach: avoid the need for new specifications
    - Makes full treatment of large, complex ISAs approachable
    - Works well if your goals are similar to those of compilers
  - These specifications can be used to develop MDs for missing instructions, or improve precision
- Efforts to discover errors in emulators (Berkeley team)
  - Our focus is on property enforcement
    - Compiler MDs use a lot of over-approximations, and don't provide a good basis for high-fidelity emulation

Robust and scalable Static analysis of low-level code

Challenges in low-level binary code

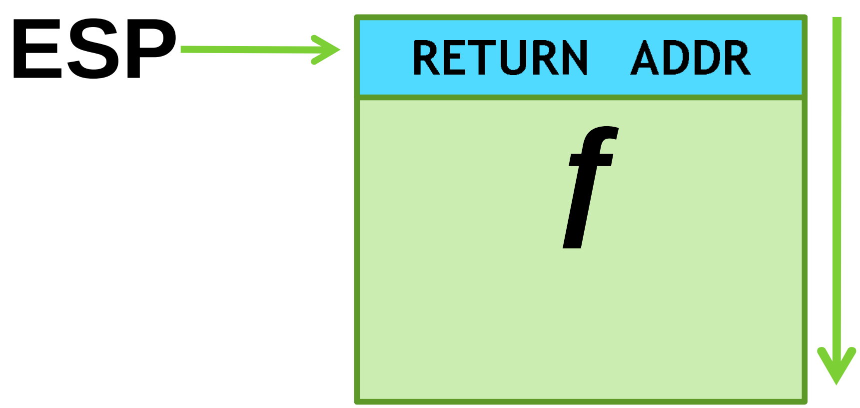
- Variable boundaries or types
- Function boundaries
- Parameter passing in optimized code
  - Missing pushes, parameter passing via registers,…
- Distinguishing local variables from other accesses
- Position-independent code (PIC), non-standard use of stack, functions with side-effects, ABI-compliance, …
- Hand-written assembly, exceptions, multi-threading, …

Static analysis of low-level code

- To solve these challenges, previous approaches
  - make optimistic assumptions, or rely on compiler idioms
    - often fail on optimized code and/or large programs
    - don’t work for other compilers, or hand-written assembly
- Our solution:
  - Use systematic analysis to reduce assumptions/heuristics
  - Accurately tracks local variables by analyzing values held in registers and on the stack
  - Trades off ability to reason about global memory to obtain scalability and modularity

Stack Analysis

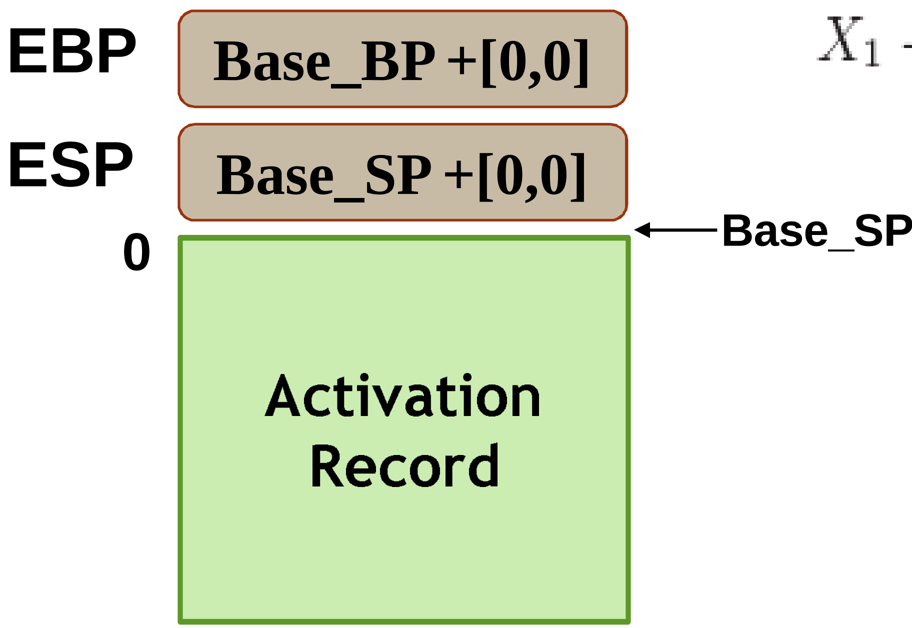
- Analyzes one function at a time
- Examines the use of stack to
  - Determine parameters
    - Number of them, whether in registers or on stack
  - Caller- and callee-saved registers
  - Summarize effect on parameters
    - Preservation of SP, return to caller, changes in parameter or register contents,…

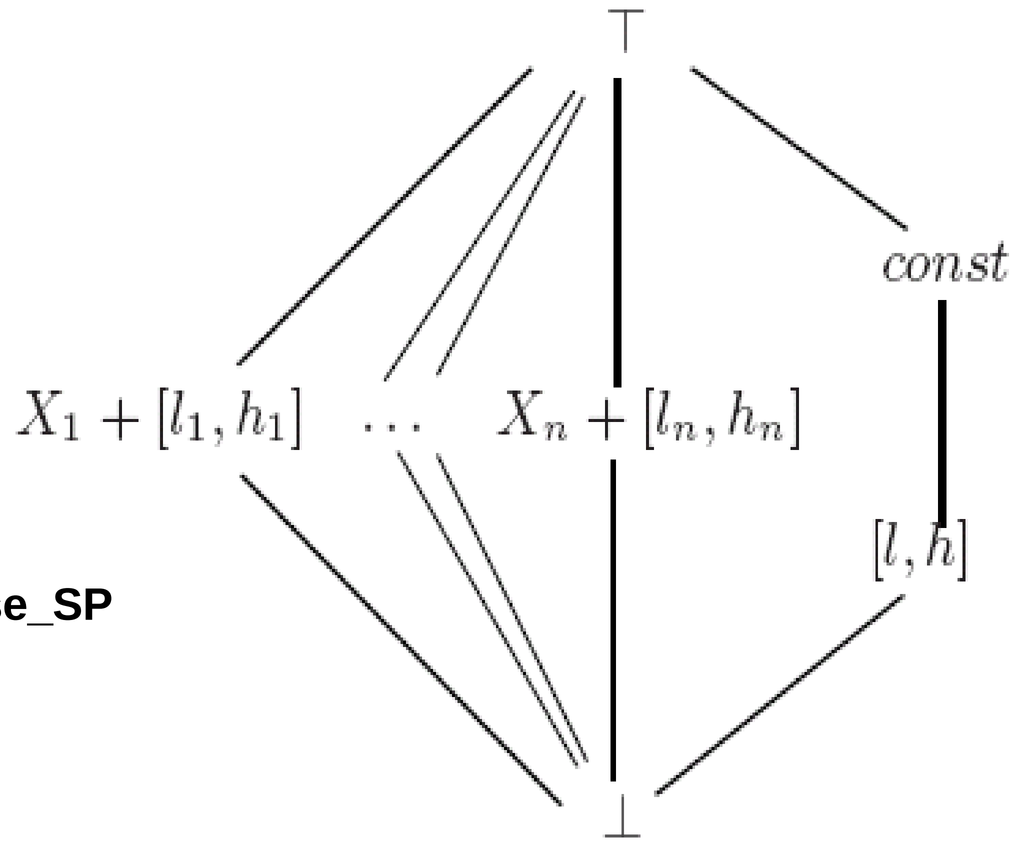


Abstract Interpretation for Stack Analysis

<*f*> :

```
push  %ebp
mov %esp, %ebp
sub $16, %esp
```



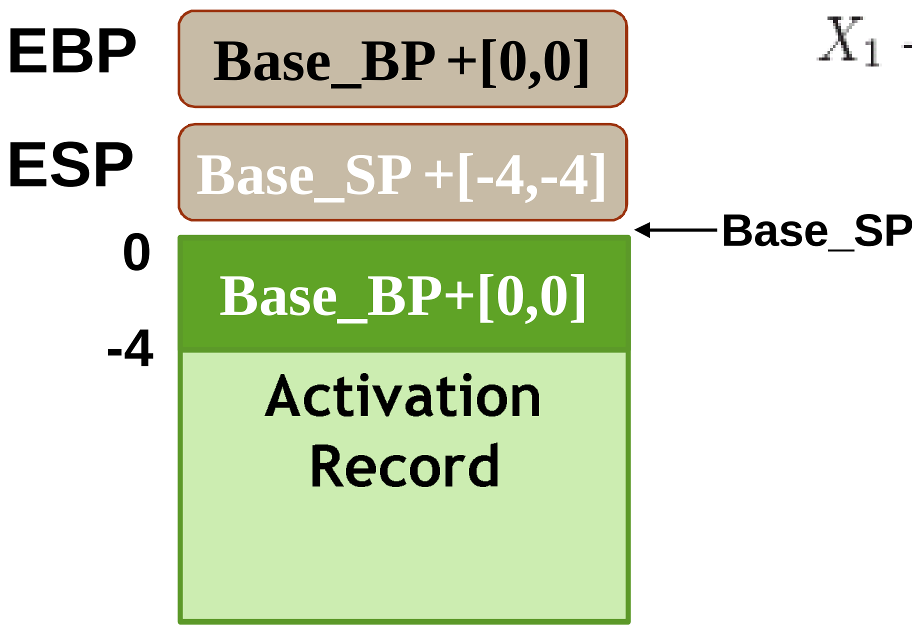


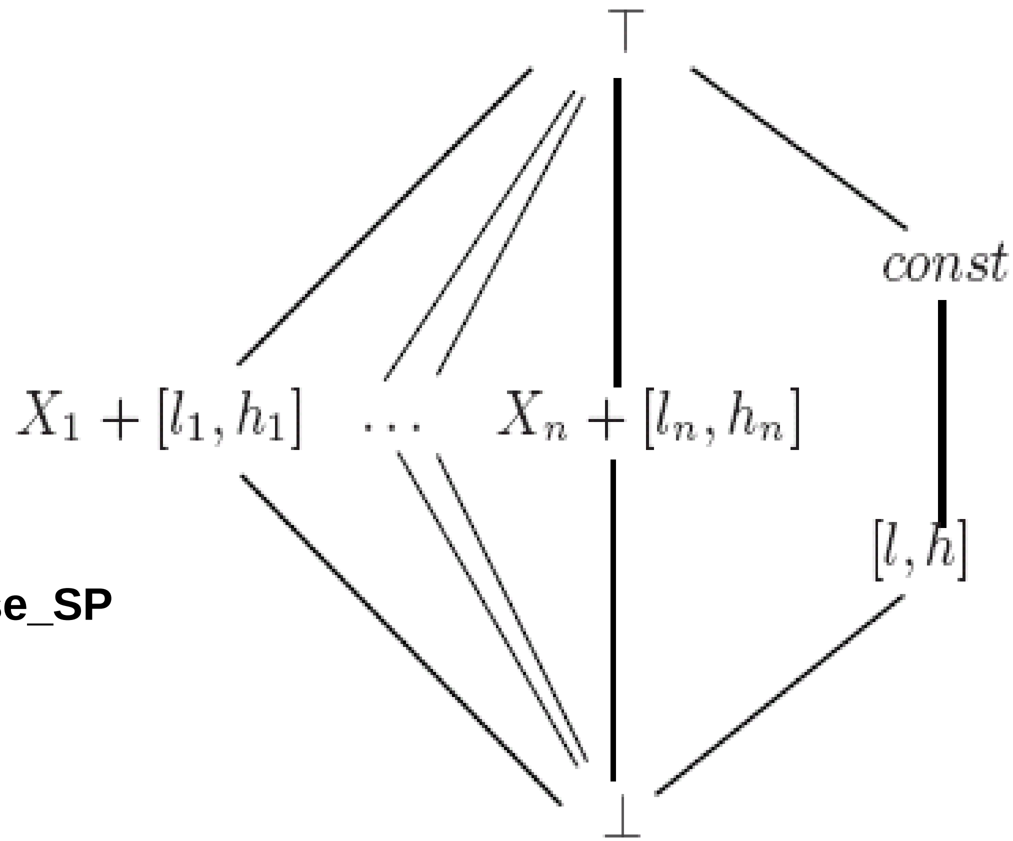
LATTICE

Abstract Interpretation for Stack Analysis

<*f*> :

```
push  %ebp
mov %esp, %ebp
sub $16, %esp
```

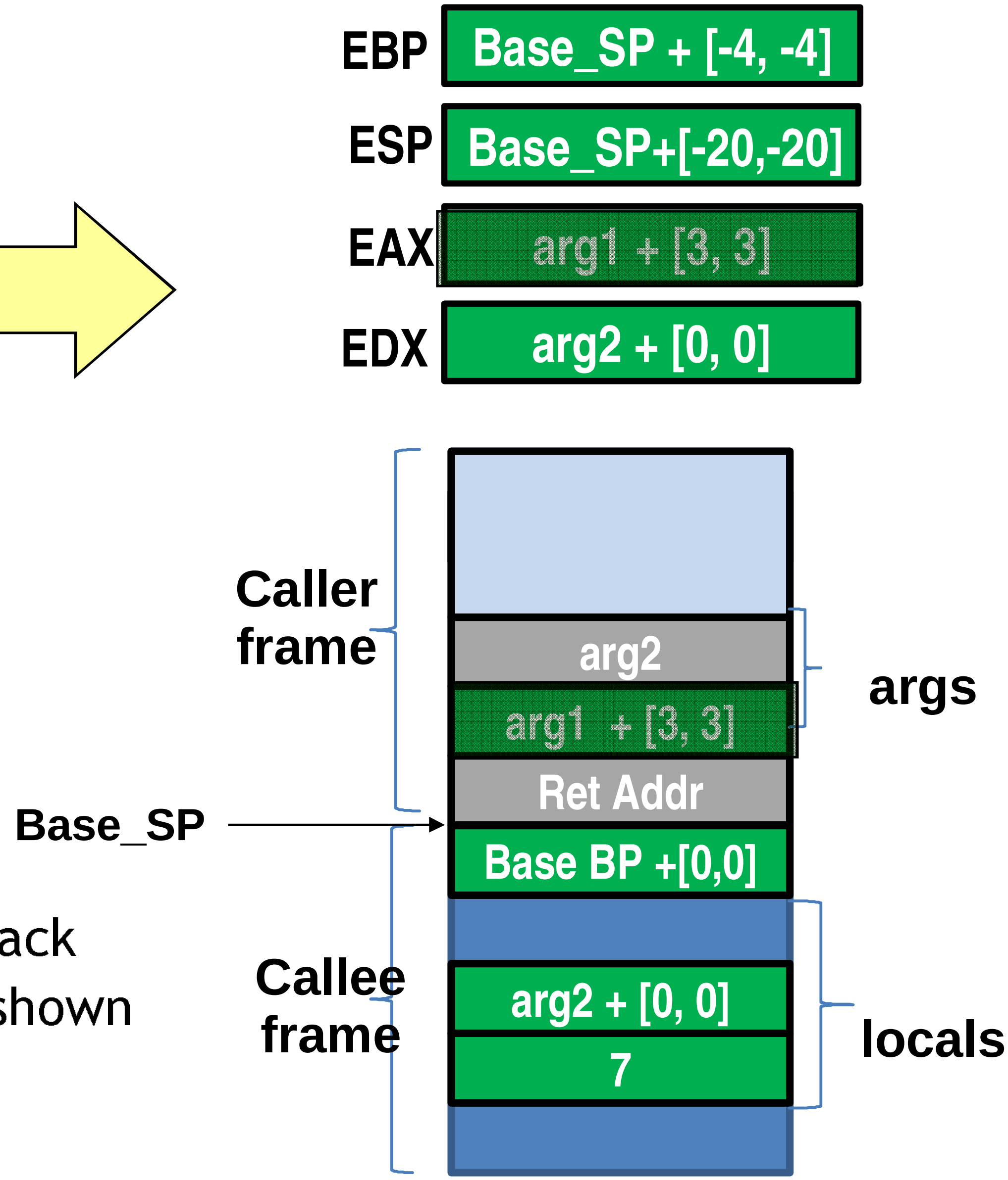




LATTICE

Stack Analysis (contd)

```
<f>:
push  %ebp
mov %esp, %ebp
sub $16, %esp
mov 8(%ebp), %eax
add $3, %eax
mov %eax, 8(%ebp)
mov $7, -12(%ebp)
mov 12(%ebp), %edx
mov %edx, -8(%ebp)
leave
ret
```

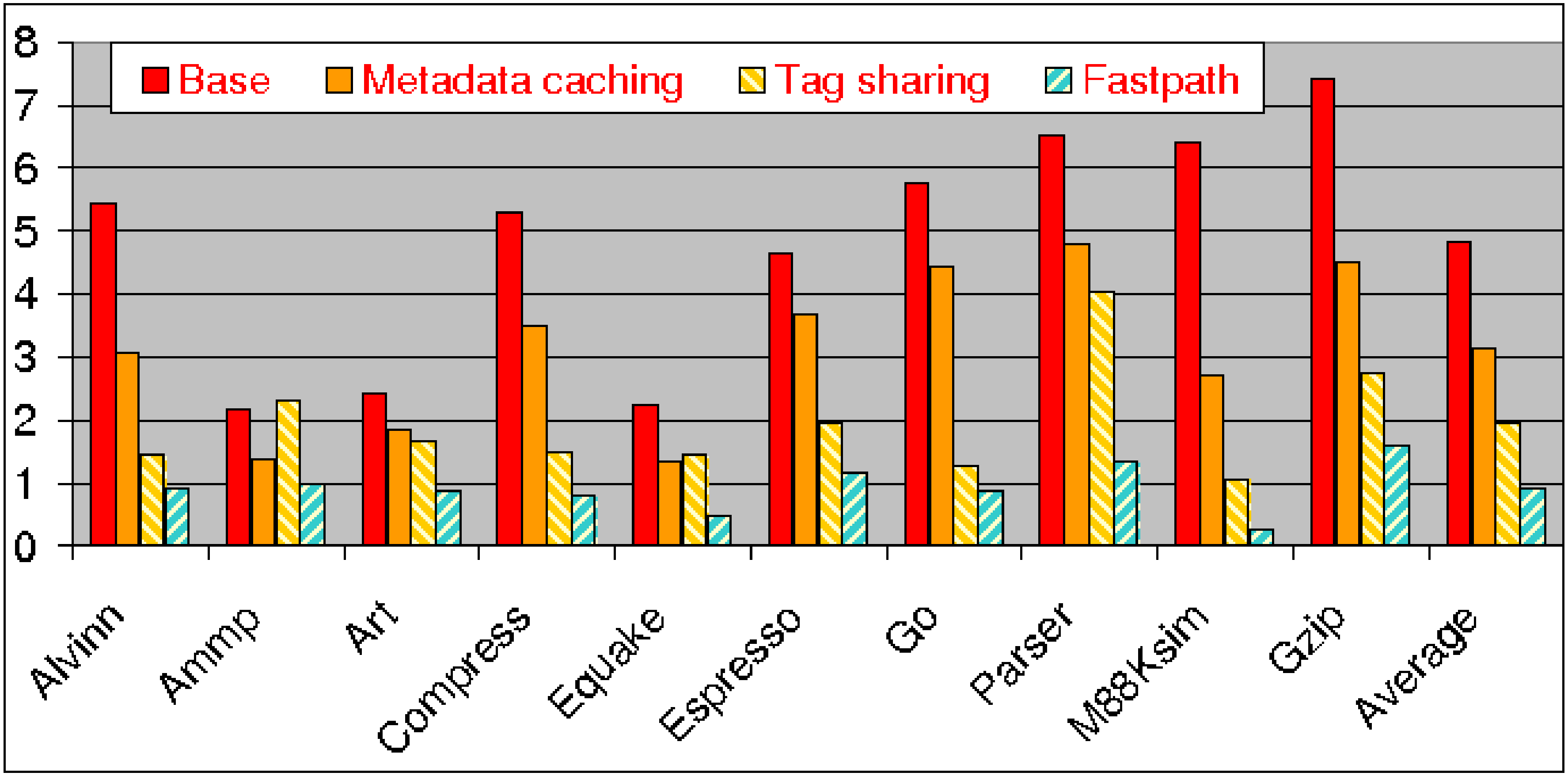


- Summary for *f*:
  - No change to ESP
  - Two input parameters on stack
  - EAX, EDX, arg1 changed as shown
  - Others unchanged

Static analysis benefits: Reducing taint-tracking overhead

- Analysis+Optimizations lead to a *6 times* performance improvement!

- 4x performance improvement over purely dynamic techniques



Summary and Future Work

- Develop novel compiler-based methods for efficient and robust binary instrumentation
  - Dramatically reduce efforts for modeling instruction sets
  - Robust, scalable static analysis of low-level code
    - Provides crucial missing pieces to complete the loop in compiler-based instrumentation
      - Abstract interpretation for accurate analysis of register, stack use
      - Type inference for discovery of code pointers
- Future work
    - Experimentation and evaluation
    - Robust and efficient binary instrumentation for information flow and related properties
    - Application to hostile OS defense