End-to-End Verification of Stack-Space Bounds for C Programs

Quentin Carbonneaux *Jan Hoffmann*

Tahina Ramananandro *Zhong Shao*

Yale University

April 14th, 2014

Does this program safely run?

```
#include <stdint.h>
typedef uint64_t t;

void f (t* pa, t* pb) {
  if (*pa == 0) return;
  *pa--;
  f (pa, pb);
  *pb++;
}

int main (int argc, char*
argv[]) {
  t a = UINT64_MAX, b = 0;
  f (&a, &b);
  return a;
}
```

- gcc -O0 && ./a.out
  - Segfault (**stack overflow**)
- gcc -O1 && ./a.out
  - OK (function inlining)

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013
111 Comments

On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

Embedded software used to be low-level code we'd bang together using C or assembler. These days, even a relatively straightforward, albeit critical, task like throttle control is likely to use a sophisticated RTOS and tens of thousands of lines of code.

With all this sophistication, standards and practices for design, coding, and testing become paramount – especially when the function involved is safety-critical. Failure is not an option. It is something to be contained and benign.

So what happens when an automaker decides to wing it and play by their own rules? To disregard the rigorous standards, best practices, and checks and balances required of such software (and hardware) design? People are killed, reputations ruined, and billions of dollars are paid out. That's what happens. Here's the story of some software that arguably never should have been.

For the bulk of this research, EDN consulted Michael Barr, CTO and co-founder of Barr Group, an embedded systems consulting firm, last week. As a primary expert witness for the plaintiffs, the in-depth analysis conducted by Barr and his colleagues illuminates a shameful example of software design and development, and provides a cautionary tale to all involved in safety-critical development, whether that be for automotive, medical, aerospace, or anywhere else where failure is not tolerable. Barr is an experienced developer, consultant, former professor, editor, blogger, and author.

Check out this related EE Live! 2014 session: Killer Apps: Embedded Software's Greatest Hit Jobs

Barr's ultimate conclusions were that:

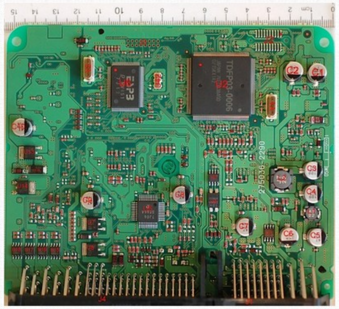
- Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality.
- Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
- Code-quality metrics predict presence of additional bugs.
- Toyota's fail safes are defective and inadequate (referring to them as a *"house of cards" safety architecture*).
- Misbehaviors of Toyota's ETCS are a cause of UA.

A damning summary to say the least. Let's look at what lead him to these conclusions:

Hardware

Although the investigation focused almost entirely on software, there is at least one HW factor: Toyota claimed the 2005 Camry's main CPU had error detecting and correcting (EDAC) RAM. It didn't. EDAC, or at least parity RAM, is relatively easy and low-cost insurance for safety-critical systems.

Other cases of throttle malfunction have been linked to tin whiskers in the accelerator pedal sensor. This does not seem to have been the case here.



The Camry ECM board. U2 is a NEC (now Renesas) V850 microcontroller.

Software

The ECM software formed the core of the technical investigation. What follows is a list of the key findings.

Mirroring (where key data is written to redundant variables) was not always done. This gains extra significance in light of …

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, MISRA-C rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

Two key items were not mirrored: The RTOS' critical internal data structures; and—the most important bytes of all, the final result of all this firmware—the TargetThrottleAngle global variable.

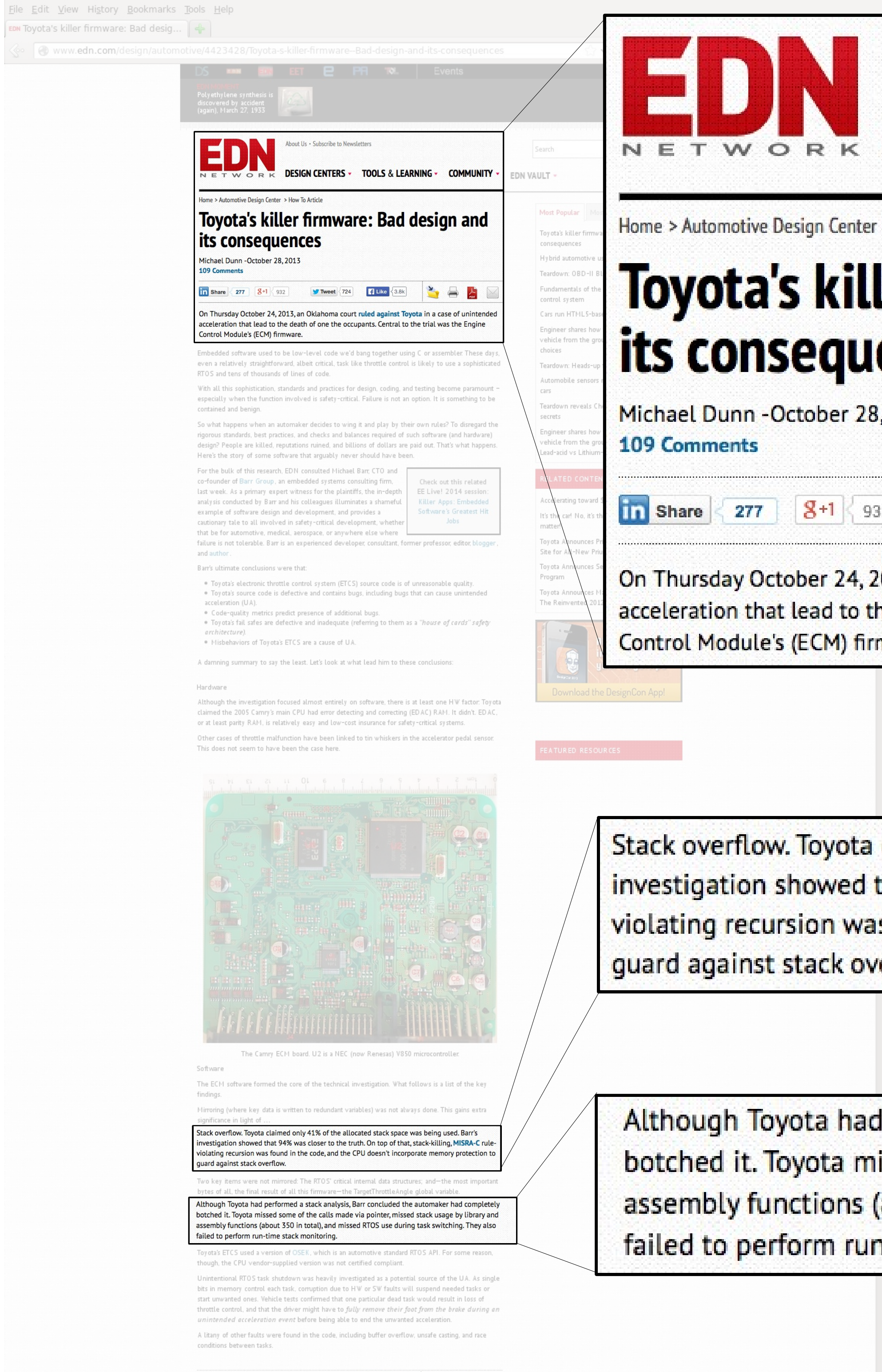
Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Toyota's ETCS used a version of OSEK, which is an automotive standard RTOS API. For some reason, though, the CPU vendor-supplied version was not certified compliant.

Unintentional RTOS task shutdown was heavily investigated as a potential source of the UA. As single bits in memory control each task, corruption due to HW or SW faults will suspend needed tasks or start unwanted ones. Vehicle tests confirmed that one particular dead task would result in loss of throttle control, and that the driver might have to *fully remove their foot from the brake during an unintended acceleration event* before being able to end the unwanted acceleration.

A litany of other faults were found in the code, including buffer overflow, unsafe casting, and race conditions between tasks.

Most Popular Most Commented
Toyota's killer firmware: Bad design and its consequences
Hybrid automotive use of ultracapacitors
Teardown: OBD-II Bluetooth adapter
Fundamentals of the automotive cabin climate control system
Cars run HTML5-based applications
Engineer shares how to build an electric vehicle from the ground up -- Part 1: Design choices
Teardown: Heads-up thermal imaging camera
Automobile sensors may usher in self-driving cars
Teardown reveals Chevy Volt's electronic secrets
Engineer shares how to build an electric vehicle from the ground up -- Part 1: Lead-acid vs Lithium-ion batteries
RELATED CONTENT
Accelerating toward $1 million prize
It's the car! No, it's the driver! Wait, it doesn't matter!
Toyota Announces Priority Registration Web Site for All-New Prius Plug-in Hybrid
Toyota Announces Second Annual Shareathon Program
Toyota Announces Marketing Campaign For The Reinvented 2012 Camry



EDN NETWORK

About Us · Subscribe to Newsletters

DESIGN CENTERS · TOOLS & LEARNING · COMMUNITY

Home > Automotive Design Center > How To Article

Toyota's killer firmware: Bad design and its consequences

Michael Dunn -October 28, 2013

109 Comments

Share 277 · +1 932 · Tweet 724 · Like 3.8k

On Thursday October 24, 2013, an Oklahoma court **ruled against Toyota** in a case of unintended acceleration that lead to the death of one the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, **MISRA-C** rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.

Does this program stack-overflow?

- Important in embedded software
  - led to deadly software bugs in Toyota cars
- Most stack analysis tools available for *compiled* code only
  - Harder to analyze
  - User interaction is troublesome
- How to prove, *at the source level*, that the compiled code does not stack-overflow?
  - How to model stack overflow at the source level?
  - How to prove stack-aware compiler correctness?

CompCert

- Formal C and assembly semantics
- Verified semantics-preserving compiler
  - Safety is preserved
  - For safe programs, I/O events and termination/divergence are preserved

CompCert and stack overflow

- Stack frame allocation always succeeds
  - Stack-overflow not modeled in either C or assembly
  - How to guarantee that, if source program does not crash, then neither does compiled code **not even by stack overflow?**

[...] it is **hopeless** to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code: stack consumption, like execution time, is a program property that is not preserved by compilation.

Xavier Leroy
(1968-)

Formal Certification of a Compiler Back-end

or: Programming a Compiler with a Proof Assistant

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr

Abstract

This paper reports on the development and formal certification (proof of semantic preservation) of a compiler from Cminor (a C-like imperative language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a certified compiler is useful in the context of formal methods applied to the certification of critical software: the certification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and verifying and reasoning about programs—Mechanical verification.; D.2.4 [*Software engineering*]: Software/program verification—Correctness proofs, formal methods, reliability; D.3.4 [*Programming languages*]: Processors—Compilers, optimization

General Terms Languages, Reliability, Security, Verification.

Keywords Certified compilation, semantic preservation, program proof, compiler transformations and optimizations, the Coq theorem prover.

1. Introduction

Can you trust your compiler? Compilers are assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers – and especially optimizing compilers – are complex programs that perform complicated symbolic transformations. We all know horror stories of bugs in compilers silently turning a correct program into an incorrect executable.

For low-assurance software, validated only by testing, the impact of compiler bugs is negligible: what is tested is the executable code produced by the compiler; rigorous testing will expose errors in the compiler along with errors in the source program. The picture changes dramatically for critical, high-assurance software whose certification at the highest levels requires the use of formal methods (model checking, program proof, etc). What is formally verified using formal methods is almost universally the source code; bugs in the compiler used to turn this verified source into an executable

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

can potentially invalidate
using formal methods. In
spective, the compiler is
that has been formally verified
and more often, has also been fo
software industry is aware of this i
niques to alleviate it, such as conduc
the generated assembly code after having turned all compiler optimizations off. These techniques do not fully address the issue, and are costly in terms of development time and program performance.

An obviously better approach is to apply formal methods to the compiler itself in order to gain assurance that it preserves the semantics of the source programs. Many different approaches have been proposed and investigated, including on-paper and on-machine proofs of semantic preservation, proof-carrying code, credible compilation, translation validation, and type-preserving compilers. (These approaches are compared in section 2.) For the last two years, we have been working on the development of a *realistic*, *certified* compiler. By *certified*, we mean a compiler that is accompanied by a machine-checked proof of semantic preservation. By *realistic*, we mean a compiler that compiles a language commonly used for critical embedded software (a subset of C) down to assembly code for a processor commonly used in embedded systems (the PowerPC), and that generates reasonably efficient code.

This paper reports on the completion of one half of this program: the certification, using the Coq proof assistant [2], of a lightly-optimizing back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. A front-end translating a subset of C to Cminor is being developed and certified, and will be described in a forthcoming paper.

While there exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see section 7 for a review), our work is novel in two ways. First, recent work tends to focus on a few parts of a compiler, mostly optimizations and the underlying static analyses [18, 6]. In contrast, our work is modest on the optimization side, but emphasizes the certification of a complete compilation chain from a structured imperative language down to assembly code through 4 intermediate languages. We found that many of the non-optimizing translations performed, while often considered obvious in compiler literature, are surprisingly tricky to formally prove correct. The other novelty of our work is that most of the compiler is written directly in the Coq specification language, in a purely functional style. The executable compiler is obtained by automatic extraction of Caml code from this specification. This approach has never been applied before to a program of the size and complexity of an optimizing compiler.

[...] it is **hopeless** to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code: stack consumption, like execution time, is a program property that is not preserved by compilation.

Xavier Leroy
(1968-)

Really?

Formal Certification of a Compiler Back-end

or: Programming a Compiler with a Proof Assistant

Xavier Leroy

INRIA Rocquencourt

Xavier.Leroy@inria.fr

Abstract

This paper reports on the development and formal certification (proof of semantic preservation) of a compiler from Cminor (a C-like imperative language) to PowerPC assembly code, using the Coq proof assistant both for programming the compiler and for proving its correctness. Such a certified compiler is useful in the context of formal methods applied to the certification of critical software: the certification of the compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and verifying and reasoning about programs—Mechanical verification.; D.2.4 [*Software engineering*]: Software/program verification—Correctness proofs, formal methods, reliability; D.3.4 [*Programming languages*]: Processors—Compilers, optimization

General Terms Languages, Reliability, Security, Verification.

Keywords Certified compilation, semantic preservation, program proof, compiler transformations and optimizations, the Coq theorem prover.

1. Introduction

Can you trust your compiler? Compilers are assumed to be semantically transparent: the compiled code should behave as prescribed by the semantics of the source program. Yet, compilers – and especially optimizing compilers – are complex programs that perform complicated symbolic transformations. We all know horror stories of bugs in compilers silently turning a correct program into an incorrect executable.

For low-assurance software, validated only by testing, the impact of compiler bugs is negligible: what is tested is the executable code produced by the compiler; rigorous testing will expose errors in the compiler along with errors in the source program. The picture changes dramatically for critical, high-assurance software whose certification at the highest levels requires the use of formal methods (model checking, program proof, etc). What is formally verified using formal methods is almost universally the source code; bugs in the compiler used to turn this verified source into an executable

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

can potentially invalidate
using formal methods. In
spective, the compiler is a
that has been formally verified
and more often, has also been fo ... ical
software industry is aware of this i ... tech-
niques to alleviate it, such as conduct ... views of
the generated assembly code after having turned all compiler optimizations off. These techniques do not fully address the issue, and are costly in terms of development time and program performance.

An obviously better approach is to apply formal methods to the compiler itself in order to gain assurance that it preserves the semantics of the source programs. Many different approaches have been proposed and investigated, including on-paper and on-machine proofs of semantic preservation, proof-carrying code, credible compilation, translation validation, and type-preserving compilers. (These approaches are compared in section 2.) For the last two years, we have been working on the development of a *realistic*, *certified* compiler. By *certified*, we mean a compiler that is accompanied by a machine-checked proof of semantic preservation. By *realistic*, we mean a compiler that compiles a language commonly used for critical embedded software (a subset of C) down to assembly code for a processor commonly used in embedded systems (the PowerPC), and that generates reasonably efficient code.

This paper reports on the completion of one half of this program: the certification, using the Coq proof assistant [2], of a lightly-optimizing back-end that generates PowerPC assembly code from a simple imperative intermediate language called Cminor. A front-end translating a subset of C to Cminor is being developed and certified, and will be described in a forthcoming paper.

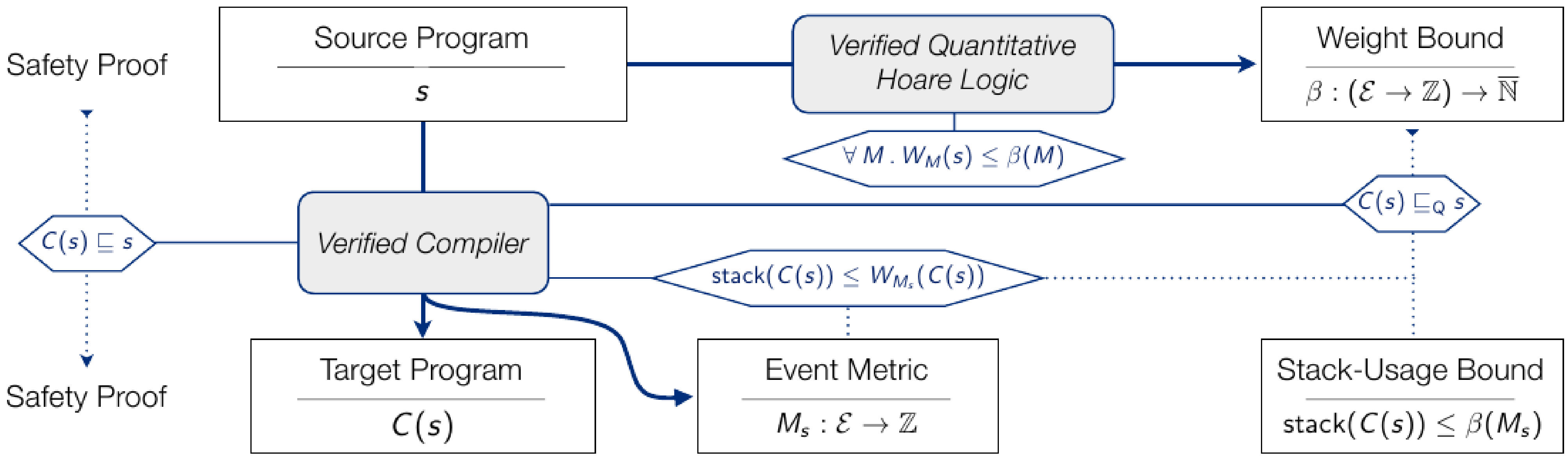
While there exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see section 7 for a review), our work is novel in two ways. First, recent work tends to focus on a few parts of a compiler, mostly optimizations and the underlying static analyses [18, 6]. In contrast, our work is modest on the optimization side, but emphasizes the certification of a complete compilation chain from a structured imperative language down to assembly code through 4 intermediate languages. We found that many of the non-optimizing translations performed, while often considered obvious in compiler literature, are surprisingly tricky to formally prove correct. The other novelty of our work is that most of the compiler is written directly in the Coq specification language, in a purely functional style. The executable compiler is obtained by automatic extraction of Caml code from this specification. This approach has never been applied before to a program of the size and complexity of an optimizing compiler.

POPL
2006

Our solution: Quantitative CompCert

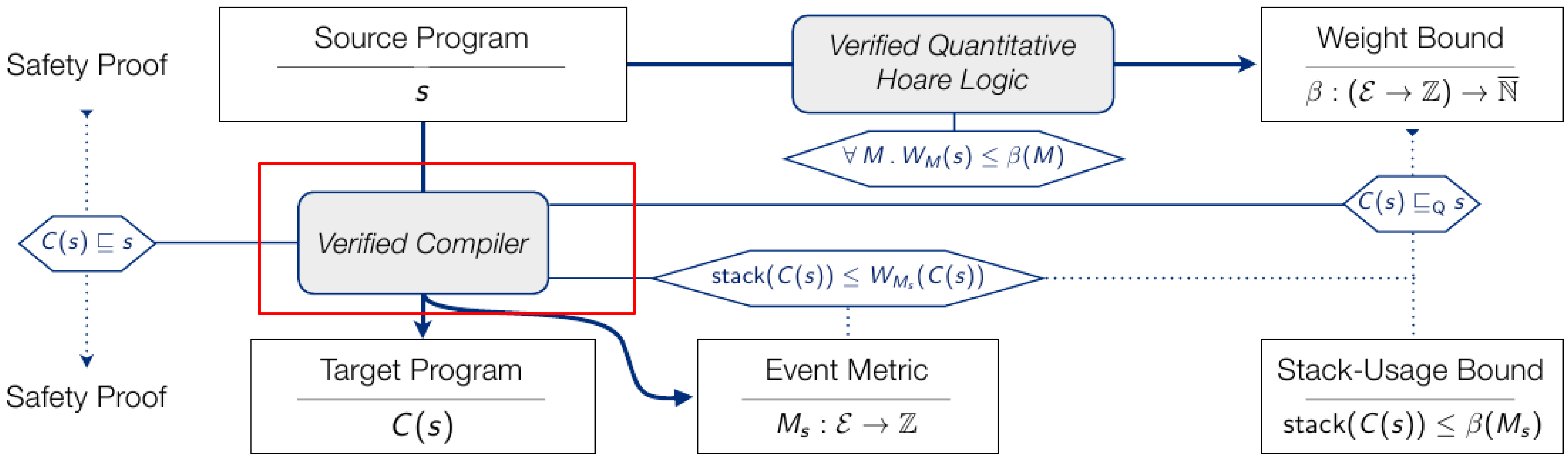
- Introduce stack consumption in C semantics
- Preserve stack consumption by compilation passes: quantitative refinement
- Refine assembly semantics with finite stack
- Make compiler correctness depend on *source-level* stack bound
  - Introduce a program logic on Clight to derive stack consumption bound
  - Introduce automatic stack analyzer to automatically use program logic on programs without recursion

Overview



Safety Proof
Source Program
s
Verified Quantitative Hoare Logic
Weight Bound
$\beta : (\mathcal{E} \to \mathbb{Z}) \to \overline{\mathbb{N}}$
$\forall M \,.\, W_M(s) \leq \beta(M)$
$C(s) \sqsubseteq_Q s$
$C(s) \sqsubseteq s$
Verified Compiler
$\mathsf{stack}(C(s)) \leq W_{M_s}(C(s))$
Safety Proof
Target Program
$C(s)$
Event Metric
$M_s : \mathcal{E} \to \mathbb{Z}$
Stack-Usage Bound
$\mathsf{stack}(C(s)) \leq \beta(M_s)$

Overview



Safety Proof
Source Program
s
Verified Quantitative Hoare Logic
Weight Bound
$\beta : (\mathcal{E} \to \mathbb{Z}) \to \overline{\mathbb{N}}$
$\forall M . W_M(s) \leq \beta(M)$
$C(s) \sqsubseteq_Q s$
$C(s) \sqsubseteq s$
Verified Compiler
$\mathrm{stack}(C(s)) \leq W_{M_s}(C(s))$
Safety Proof
Target Program
$C(s)$
Event Metric
$M_s : \mathcal{E} \to \mathbb{Z}$
Stack-Usage Bound
$\mathrm{stack}(C(s)) \leq \beta(M_s)$

Stack consumption in C semantics

- CompCert C produces an I/O *event trace*
  - Preserved by compilation
- Add function **call/return events**

- Model the stack consumption as *trace weight* parameterized by an *event metric* for call/return events
  - Preserve the *weights*
  - Stack consumption of a function is parameterized by the **stack frame sizes** of its callees

- Operational semantics does not go wrong on stack overflow
  - Does not know the event metric, only generates events

Example

```
int f (int x) {
  return x+1;
}

main () {
  f(0);
}
```

- main() generates trace:

  call(main) ::
  call(f) :: return(f) ::
  return(main) :: nil

- Stack consumption:

  M(*main*) + M(*f*)

  where M is an *event metric* (giving non-negative stack frame size for each function)

Stack consumption

- Events e ::= … | call(f) | return(f)
- Event and trace valuation:

  $V_M(call(f)) = M(f)$; $V_M(return(f)) = -M(f)$;
  $V_M(e) = 0$ otherwise

  $V_M(nil) = 0$; $V_M(e::t) = V_M(e) + V_M(t)$

- Trace weight:

  $W_M(T) = \sup \{V_M(t) \mid T = t \,.\, T'\}$

Stack consumption

Coq implementation: I/O events have constant (maybe non-null) stack consumption

- Event and trace valuation:

  $V'_M(e) = V_M(e)$ for call/return

  $V'_M(nil) = 0$; $V'_M(t{+}{+}e::nil) = \max(V'_M(t), V_M(t)+V'_M(e))$

- Trace weight:

  $W'_M(T) = \sup \{V'_M(t) \mid T = t . T'\}$

Quantitative refinement

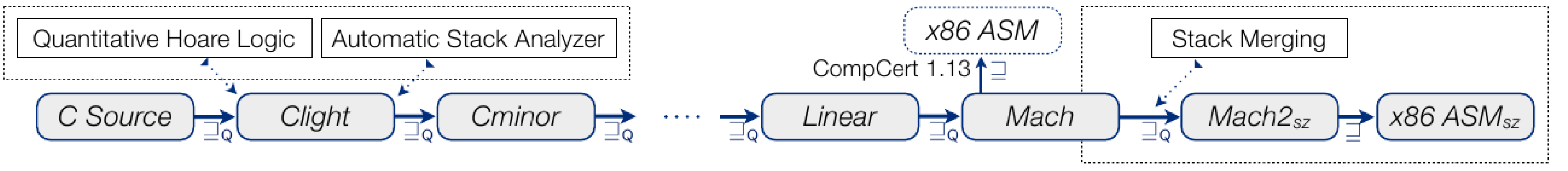
For any target behavior T', there exists a source behavior T such that:

- Pruned traces (call/return events removed) are preserved
- Termination/divergence is preserved
- For all metrics M, $W_M(T') \leq W_M(T)$
  - Equality holds for most passes (all events preserved)
  - Do not change the metric during a pass (use the assembly metric)

Quantitative compiler correctness

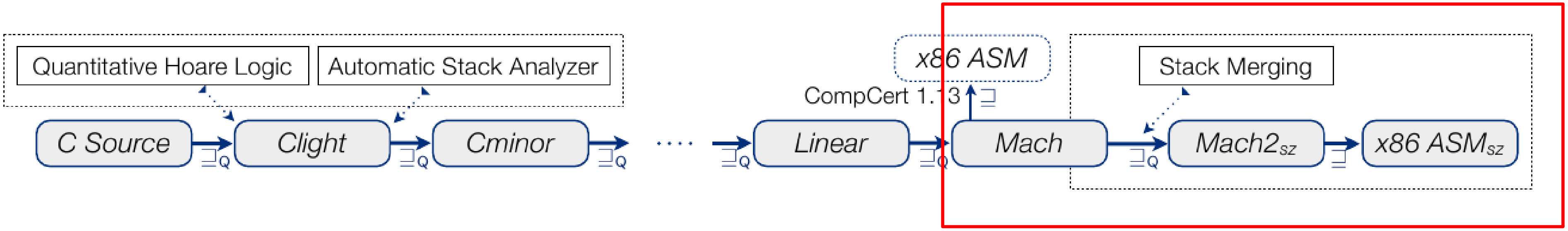
- Given stack size $\beta < 2^{31}$, for all source code *s*, if all the following hold:
  - The compiler produces assembly code C(s) *and event metric* M
  - *s* does not go wrong in infinite stack space
  - **All traces T of *s* have weight $W_M(T) \leq \beta$**
  - Assembly C(s) is run with β stack size
- Then:
  - C(s) refines s (I/O events and termination/divergence are preserved)
  - C(s) does not go wrong
  - In particular, C(s) is guaranteed to not stack overflow

Quantitative CompCert



- Function inlining and tailcall recognition underway
- All other passes supported

Quantitative CompCert



CompCert stack management

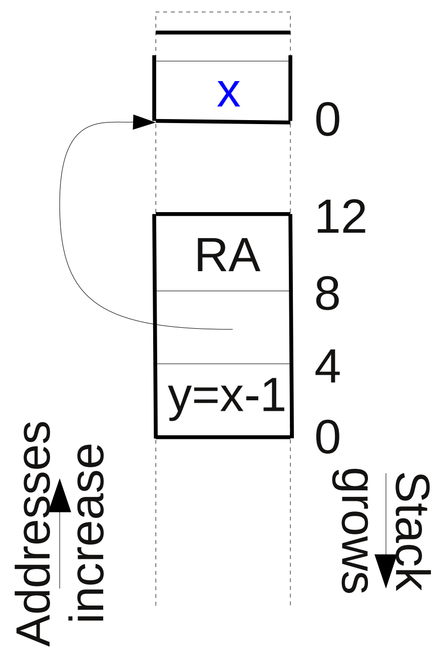
- CompCert memory model: allocate a fresh stack frame memory block upon function entry
  - No pointer arithmetics across different memory blocks
  - Always succeeds
- Still used for assembly language semantics
  - Requires Pallocframe/Pfreeframe pseudo-instructions to manage stack frame blocks
  - Turned into pointer arithmetics by unverified “pretty-printing” phase

CompCert-generated assembly...

```
int g(int y);

int f(int x) {
  return g(x-1)-2;
}
```

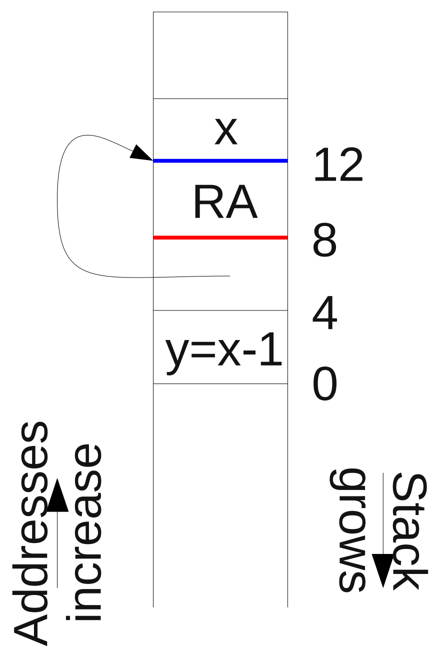
```
f:
  Pallocframe  12, 4
  mov  $4(%esp)   , %edx
  movl (%edx)     , %eax
  subl $1         , %eax
  movl %eax       , (%esp)
  call g
  subl $2         , %eax
  Pfreeframe   12, 4
  ret
```

- Formal semantics of Pallocframe/Pfreeframe also:
  - stores/loads return address in/from callee's stack frame
    - Uses RA pseudo-register to model caller's return address slot
  - stores/loads back link to caller's stack frame



… after unverified "pretty-printing"

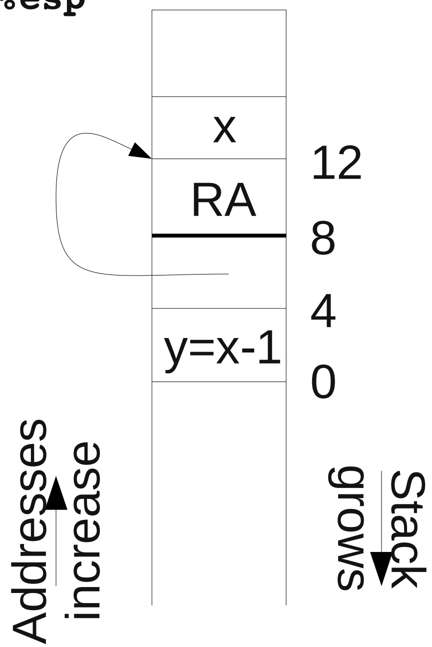
```
f:
  Pallocframe   12, 4
  mov   $4(%esp)   , %edx
  movl  (%edx)     , %eax
  subl  $1         , %eax
  movl  %eax       , (%esp)
  call  g
  subl  $2         , %eax
  Pfreeframe    12, 4
  ret
```



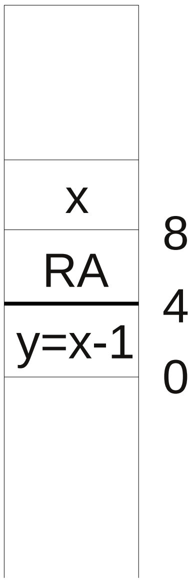

```
f:
  subl $8          , %esp
  leal $12(%esp)   , %edx
  movl %edx        , 4(esp)
  mov  $4(%esp)    , %edx
  movl (%edx)      , %eax
  subl $1          , %eax
  movl %eax        , (%esp)
  call g
  subl $2          , %eax

  addl $8          , %esp
  ret
```

But we can do better and prove it!

```
f:
  subl $8          , %esp
  leal $12(%esp)   , %edx
  movl %edx        , 4(esp)
  mov  $4(%esp)    , %edx
  movl (%edx)      , %eax
  subl $1          , %eax
  movl %eax        , (%esp)
  call g
  subl $2          , %eax
  addl $8          , %esp
  ret
```



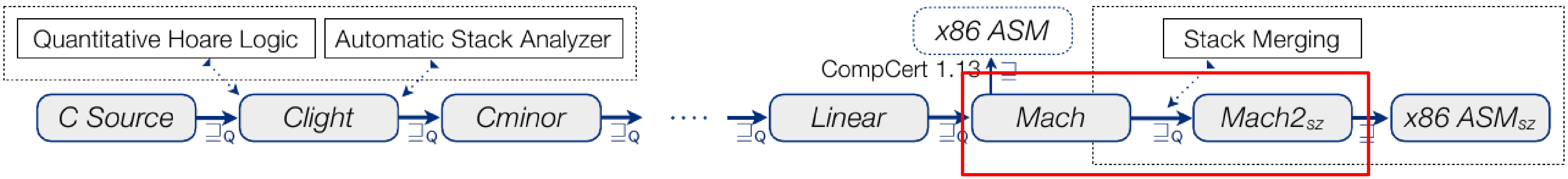

```
f:
  subl $4          , %esp
  mov  $8(%esp)    , %eax
  subl $1          , %eax
  movl %eax        , (%esp)
  call g
  subl $2          , %eax
  addl $4          , %esp
  ret
```



Assembly with finite stack

- Allocate one single stack block at program start
  - Program goes wrong on stack overflow
  - No need for pseudo-instructions
- Merge all stack frames together into the single stack block
  - Requires *memory injection* proof

Quantitative CompCert

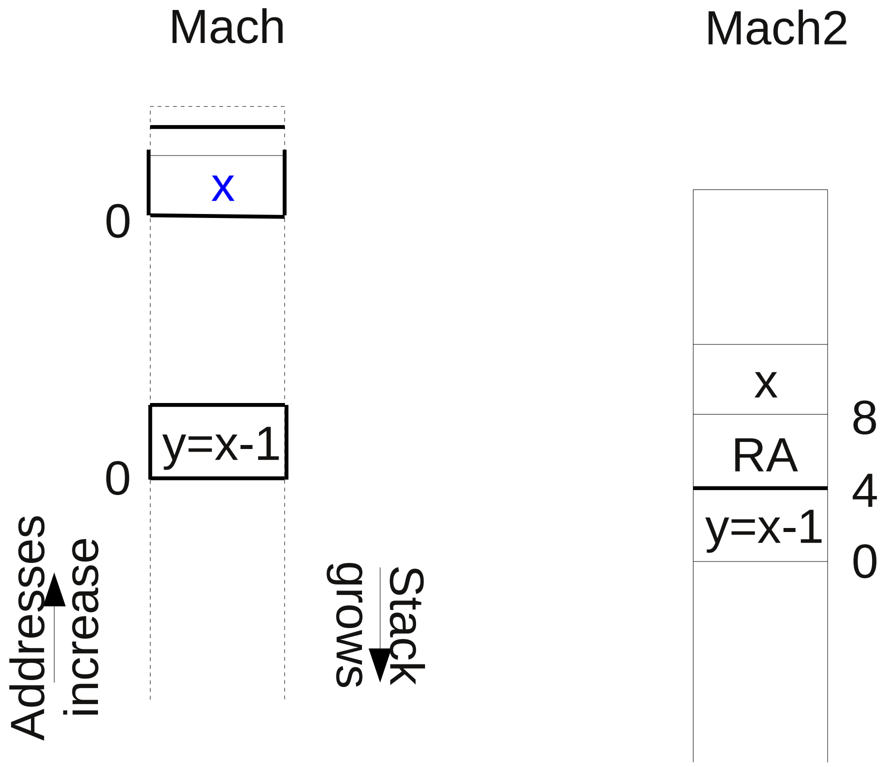


Stack merging

- CompCert Mach to single-stack Mach2 phase
  - Mach already puts arguments into stack
  - Mach no longer stores RA into stack, Mach2 does
  - Mach and Mach2 have same syntax
  - No code transformation: reinterpretation of semantics with single stack
- Mach2 to assembly
  - Implement function entry/exit with stack pointer arithmetics
  - No significant memory changes
- Total changes: 5k LOC (out of CompCert's 90k)

Mach vs. Mach2

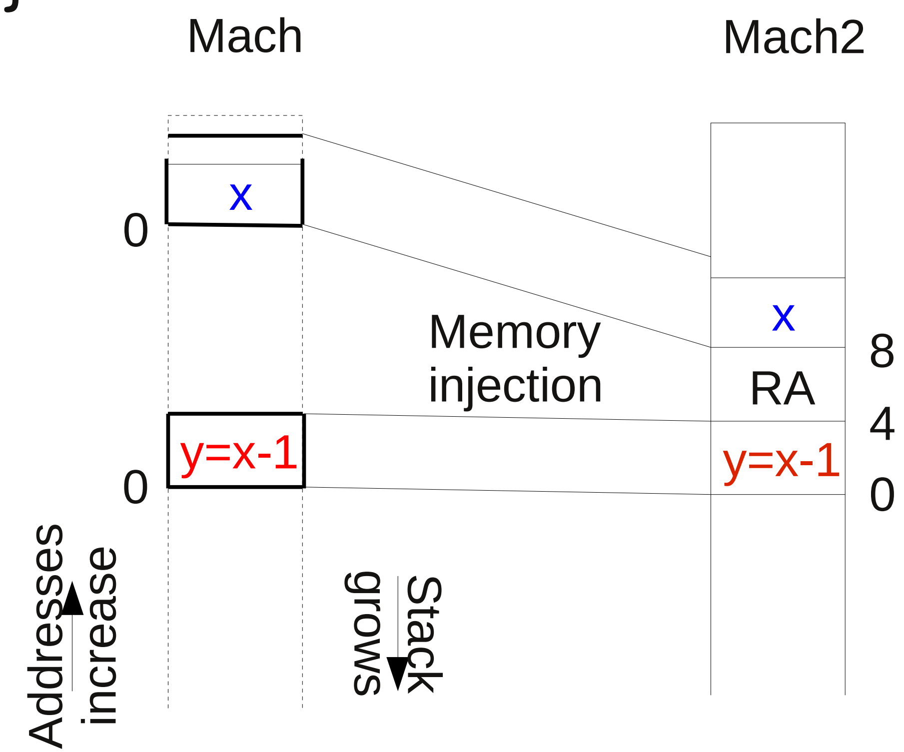
- Registers (x86)
  r := EAX | EBX | ECX | EDX | FP0
- Statements (r* registers, ofs constant integer)
  S ::=Mload(chunk, raddr, rres)
  | Mstore(chunk, raddr, rval)
  | Mgetstack(chunk, ofs, rres)
  | Msetstack(chunk, ofs, rres)
  | Mgetparam(chunk, ofs, rres)
  | Mcall func | Mret
  | Mgoto label | Mlabel label: | ...



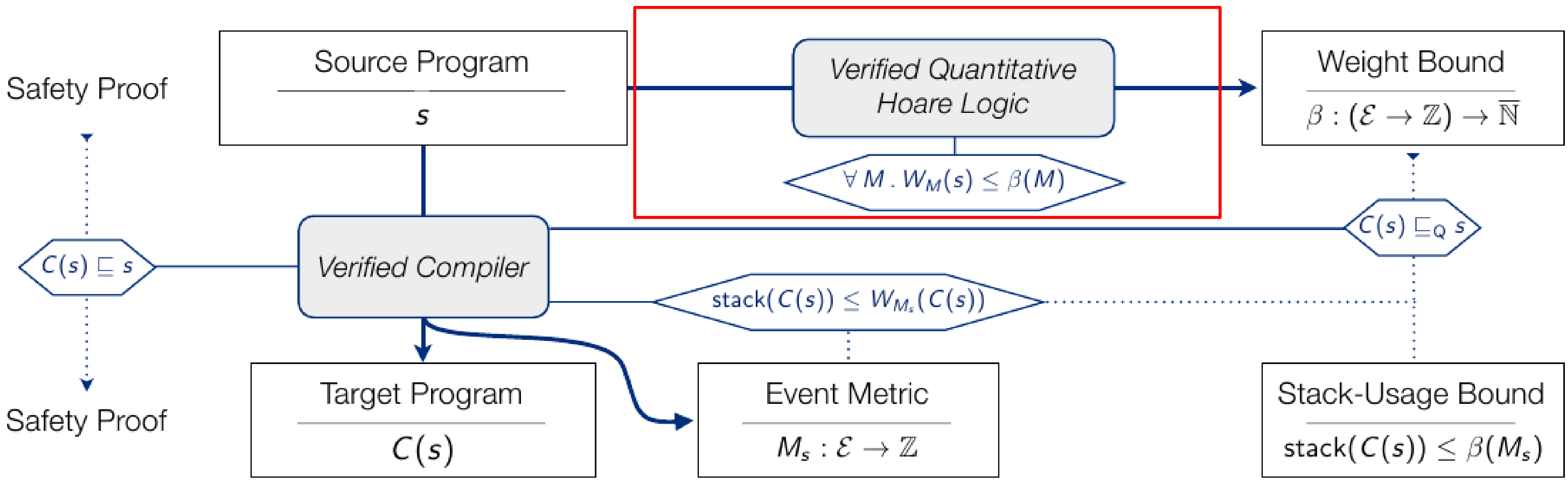
Mach vs. Mach2

```
int g(int y);

int f(int x) {
  return g(x-1)-2;
}
```

```
int g {...}

int f {
  Mgetparam(Mint32, 0, EAX);
  Mop(Osubimm 1, EAX);
  Msetstack(Mint32, 0, EAX);
  Mcall(g);
  Mop(Osubimm 2, EAX);
  Mret
}
```



Overview



Safety Proof
Source Program
s
Verified Quantitative Hoare Logic
$\forall M . W_M(s) \leq \beta(M)$
Weight Bound
$\beta : (\mathcal{E} \to \mathbb{Z}) \to \overline{\mathbb{N}}$
$C(s) \sqsubseteq_Q s$
$C(s) \sqsubseteq s$
Verified Compiler
$\text{stack}(C(s)) \leq W_{M_s}(C(s))$
Safety Proof
Target Program
$C(s)$
Event Metric
$M_s : \mathcal{E} \to \mathbb{Z}$
Stack-Usage Bound
$\text{stack}(C(s)) \leq \beta(M_s)$

Quantitative program logic

- Hoare-like logic
- Assertions have values in $\{0, 1, 2, \ldots, \infty\}$
  - Represent available stack space
- {P} S {Q} roughly: if P stack space is available before S, then:
  - S does not stack overflow (unless $P=\infty$), and
  - for all possible terminating executions of S, Q stack space is available after S

Assertions

- Clight statements S, continuations K, local state θ
- Global state (“heap” = CompCert memory state) H
- Mutable state σ = (θ, H)
- Configuration C = (S, K, σ)
- Assertion P: C → {0, 1, 2, …, ∞}
  - Coq implementation: C→**N**→**Prop**, represents sets of valid bounds

Selected rules

$$\frac{\{P\}\,S_1\,\{R\} \qquad \{R\}\,S_2\,\{Q\}}{\{P\}\,S_1; S_2\,\{Q\}}\ (\text{Q:}\textsc{Seq})$$

$$\frac{c \geqslant 0 \qquad \{P\}\,S\,\{Q\}}{\{P + c\}\,S\,\{Q + c\}}\ (\text{Q:}\textsc{Frame})$$

$$\frac{P \geqslant P' \qquad \{P'\}\,S\,\{Q'\} \qquad Q' \geqslant Q}{\{P\}\,S\,\{Q\}}\ (\text{Q:}\textsc{Conseq})$$

Selected rules

$$\frac{\Gamma(f) = (P, Q)}{\Gamma \vdash \{P + M(f)\}\, f()\, \{Q + M(f)\}} \text{ (Q:CALL)}$$

$$\frac{\Gamma' = \Gamma, f : (P_f, Q_f) \qquad \Sigma(f) = S_f \qquad \Gamma' \vdash \{P\}\, S\, \{Q\} \qquad \Gamma' \vdash \{P_f\}\, S_f\, \{Q_f\}}{\Gamma \vdash \{P\}\, S\, \{Q\}} \text{ (Q:ABSTRACT)}$$

Selected rules

With:

- Global variable addresses Δ
- Loop break
- Mutable state (θ, H)
- Return value
- One argument

those rules become:

$$\frac{\Gamma(f) = (P_f, Q_f) \qquad P = \lambda(\theta, H) \,.\, P_f([\![E]\!]^{\Delta}_{(\theta,H)}, H) \qquad Q = \lambda(\theta, H) \,.\, Q_f([\![x]\!]^{\Delta}_{(\theta,H)}, H)}{\Gamma \vdash \{P + M(f)\}\, x = f(E)\, \{(Q + M(f), \bot, \bot)\}} \text{ (Q:CALL)}$$

$$\frac{\Gamma' \vdash \{P\}\, S\, \{Q\} \qquad \Gamma' \vdash \{P'\}\, S_f\, \{\bot, \bot, Q'\} \qquad \begin{array}{c}\Gamma' = \Gamma, f : (P_f, Q_f) \qquad \Sigma(f) = (x, S_f) \\ P' = \lambda(\theta, H) \,.\, P_f(\theta(x), H) \qquad Q' = \lambda(\theta, H) \,.\, \lambda r \,.\, Q_f(r, H)\end{array}}{\Gamma \vdash \{P\}\, S\, \{Q\}} \text{ (Q:ABSTRACT)}$$

But we also support:

- Several function arguments
- Auxiliary state
- Stack framing

See paper for more details.

Example with auxiliary state

$\{Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot Z)\}$

```
bsearch(x,l,h) {
  if (h-l <= 1) return l;
```

$\{(Z>0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma)) \Rightarrow M_b \cdot Z\}$

```
  m = l+(h-l)/2;
```

$\{(Z>0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \wedge \mathsf{m}_\sigma = \frac{\mathsf{h}_\sigma + \mathsf{l}_\sigma}{2}) \Rightarrow M_b \cdot Z\}$

```
  if (a[m]>x) h=m else l=m;
```

$\{[Z-1 = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot (Z-1)] + M_b\}$

```
  return bsearch(x,l,h);
```

$\{[M_b \cdot (Z-1)] + M_b\}$

```
}
```

$\{M_b \cdot Z\}$

Soundness

- “**C** consumes at most P stack space” iff for any t, **C**' such that **C** –t→* **C**', and for any metric M, $W_M(t) \leq P(\mathbf{C}, M)$
- If {P} S {Q} is derivable, then for any σ, (S, Kstop, σ) consumes at most P stack space
  - Stronger soundness: for any K, σ if (skip, K, σ) consumes at most Q stack space, then (S, K, σ) consumes at most P stack space
- Logic and soundness: 700 LOC
  Instantiation to Clight: 950 LOC

Accuracy

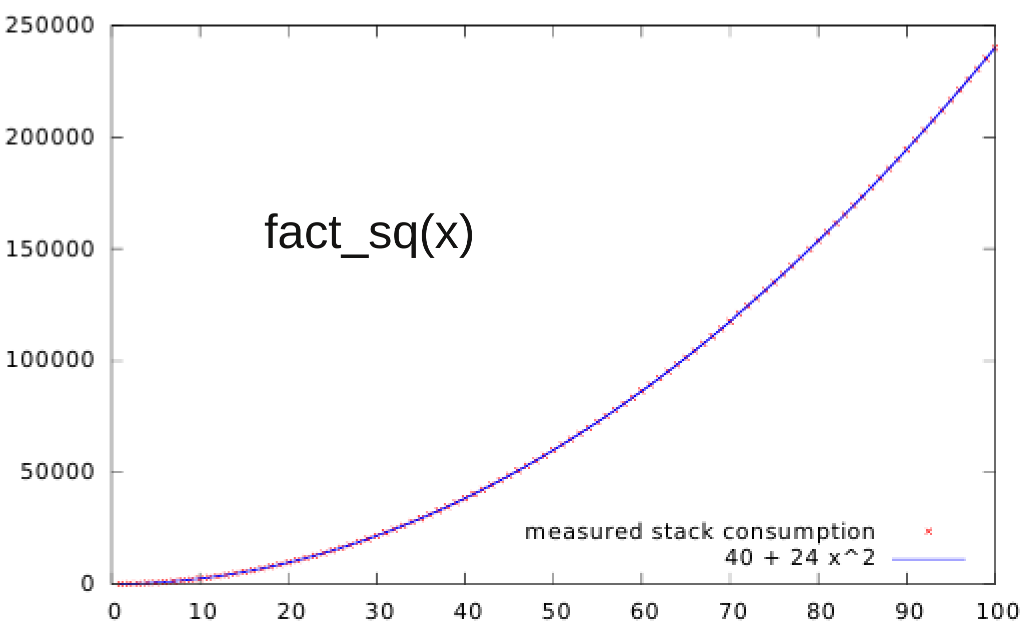
Function Name	Verified Stack Bound
recid()	8a bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathrm{hi} - \mathrm{lo}))$ bytes
fib(n)	24n bytes
qsort(a, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathrm{hi} - \mathrm{lo})$ bytes
sum(a, lo, hi)	$32(\mathrm{hi} - \mathrm{lo})$ bytes
fact_sq(n)	$40 + 24\mathrm{n}^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathrm{hi} - \mathrm{lo}) + 40\log_2(\mathrm{BL})$ bytes

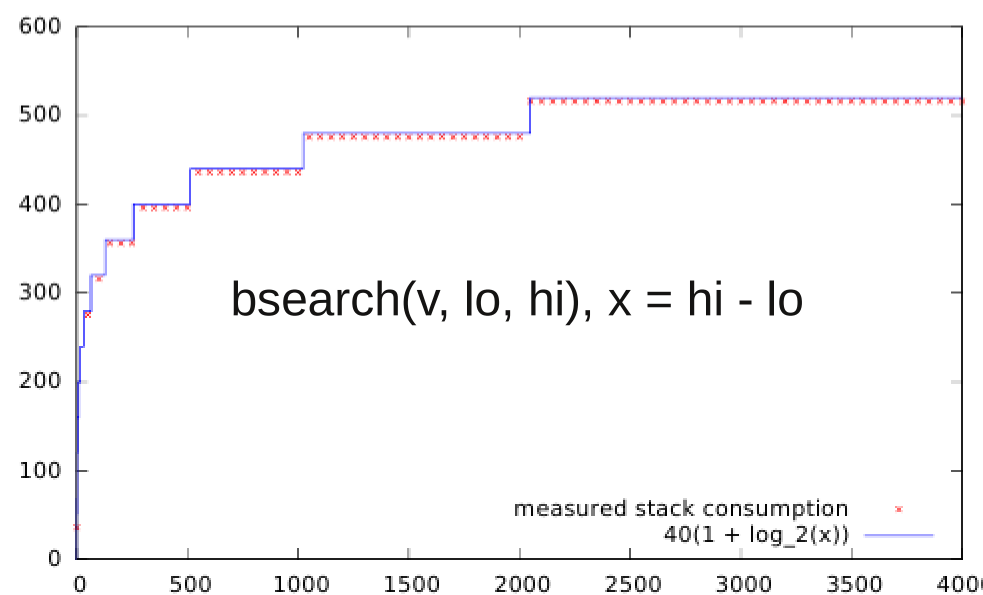
Table 1. Manually verified stack bounds for C functions.

- Bound verified manually
  using our program logic, then
  instantiated by CompCert-generated
  stack frame sizes

- Actual stack consumption measured
  at run-time thanks to a *stack monitor*
  using `ptrace` (200 lines of C+Perl)

- 4 bytes difference
  due to space reserved for RA
  in the last callee's stack frame





Automatic stack analyzer

- For C code without recursion (e.g. MISRA C), program logic can be automatically applied to derive stack bound
  - 500 lines of Coq
- Instrumented compiler to generate both compiled code and stack bound
  - 400 lines of Coq + 500 Ocaml

Automatic stack analyzer

- ```
  Let liftO {A B C: Type} (f: A -> B -> C)
  (ox: option A) (oy: option B): option C :=
  ...
  ```

- ```
  Fixpoint B M Γ (s: stm): option nat :=
    match s with
    | scall _ f _ =>
      liftO plus (Some (M f)) (Γ f)
    | sseq s1 s2 =>
      liftO max (B M Γ s1) (B M Γ s2)
    | sif _ st sf =>
      liftO max (B M Γ Phi st) (B M Γ sf)
    | sloop s => B M Γ s
    | _ => Some 0
    end.
  ```

- ```
  Lemma sound_B:
    forall M Γ (CVALID: valid_bctx M Γ) s n
           (BS: B M Γ s = Some n),
    valid_bound M s n.
  ```

```
Proof.
  induction s; intros; ...
  + apply sound_skip.
  + apply sound_ret with (Q := fun _ => mkassn 0).
  + apply sound_break.
  + … apply sound_seq with (Q := fun _ => mkassn (max x y)) …
apply valid_max_l … apply valid_max_r ...
  + case_eq (Γ f) ... eapply valid_le; [ apply Le.le_n_Sn |].
    eapply sound_consequence;
    [| apply sound_call2
      with (C := Γ)
           (Pg := fun_pre phif)
           (Qg := fun_post phif)
           (L := fun _ _ => True) ].
  …  eapply CVALID; eauto.
  + eapply sound_consequence;
   [| apply sound_loop
      with (I := fun _ => mkassn n)
           (Q := fun _ => mkassn n)
   ]; unfold mkassn; intuition. …
  eapply IHs; eauto.
Qed.
```
```

Automatic stack analyzer: soundness

- ```
  Let liftO {A B C: Type} (f: A -> B -> C)
  (ox: option A) (oy: option B): option C :=
  ...
  ```
- ```
  Fixpoint B M Γ (s: stm): option nat :=
    match s with
    | scall _ f _ =>
       liftO plus (Some (M f)) (Γ f)
    | sseq s1 s2 =>
       liftO max (B M Γ s1) (B M Γ s2)
    | sif _ st sf =>
       liftO max (B M Γ Phi st) (B M Γ sf)
    | sloop s => B M Γ s
    | _ => Some 0
    end.
  ```
- ```
  Lemma sound_B:
    forall M Γ (CVALID: valid_bctx M Γ) s n
           (BS: B M Γ s = Some n),
    valid_bound M s n.
  ```
- ```
  Fixpoint bound_of_lvl  ge M
    (lvl: nat) f :=
    match lvl with
    | 0 => None
    | S lvl' =>
        match find_func_ ge f with
        | Some bdy =>
          B M (bound_of_lvl ge M lvl') bdy
        | None => None
        end
    end.
  ```
- ```
  Theorem bound_lvl_sound:
    forall ge M l,
     valid_bctx M (bound_of_lvl ge M l).
  Proof.
    induction l.
    …  apply sound_B … apply IHl …
  Qed.
  ```
  ```

Automatic stack analyzer: “completeness”

- ```
  Fixpoint bound_of_lvl  ge M
    (lvl: nat) f :=
    match lvl with

    | 0 => None
    | S lvl' =>
      match find_func_ ge f with
      | Some bdy =>
        B M (bound_of_lvl ge M lvl') bdy
      | None => None
      end
    end.
  ```

- ```
  Theorem bound_of_lvl_complete:
    forall M p
    (CLOSED: … p …)
    (CG_WELLFOUNDED:
      forall id fi,
      In (id, Gfun fi) p.(prog_defs) →
      forall id',
      in_stm id' fi.(fi_body) →
      id' < id)
    lvl f
    (LVL: f < lvl)
    fi
    (FDEF: In (f, Gfun fi) p.(prog_defs)),
    exists n,
      bound_of_lvl
       (Genv.globalenv p) M
       lvl f = Some n.
  ```

Automatic stack bounds

File Name / Line Count	Function Name	Verified Stack Bound
mibench/net/dijkstra.c *(174 LOC)*	enqueue	40 bytes
	dequeue	40 bytes
	dijkstra	88 bytes
mibench/auto/bitcount.c *(110 LOC)*	bitcount	16 bytes
	bitstring	32 bytes
mibench/sec/blowfish.c *(233 LOC)*	BF_encrypt	40 bytes
	BF_options	8 bytes
	BF_ecb_encrypt	80 bytes
mibench/sec/pgp/md5.c *(335 LOC)*	MD5Init	16 bytes
	MD5Update	168 bytes
	MD5Final	168 bytes
	MD5Transform	128 bytes
mibench/tele/fft.c *(195 LOC)*	IsPowerOfTwo	16 bytes
	NumberOfBitsNeeded	24 bytes
	ReverseBits	24 bytes
	fft_float	160 bytes
certikos/vmm.c *(608 LOC)*	palloc	48 bytes
	pfree	40 bytes
	mem_init	72 bytes
	pmap_init	176 bytes
	pt_free	80 bytes
	pt_init	152 bytes
	pt_init_kern	136 bytes
	pt_insert	80 bytes
	pt_read	56 bytes
	pt_resv	120 bytes
certikos/proc.c *(819 LOC)*	enqueue	48 bytes
	dequeue	48 bytes
	kctxt_new	72 bytes
	sched_init	232 bytes
	tdqueue_init	208 bytes
	thread_init	192 bytes
	thread_spawn	96 bytes
compcert/mandelbrot.c *(92 LOC)*	main	56 bytes
compcert/nbody.c *(174 LOC)*	advance	80 bytes
	energy	56 bytes
	offset_momentum	24 bytes
	setup_bodies	16 bytes
	main	112 bytes

Table 2. Automatically verified stack bounds for C functions.

Conclusion

- Stack overflow need not be enforced by source semantics
  - Stack consumption as add-on to existing operational semantics
- Yet, stack consumption can be verified at the source level and preserved by compilation
- Paves the way for other quantitative properties:
  - Malloc/free heap memory consumption
  - clock cycles, energy...

Thank you!

- Paper (accepted to PLDI 2014, to appear), TR, Coq development and artifact VM: http://cs.yale.edu/~tahina/certikos/stack
- For any questions: tahina.ramananandro@yale.edu

Function inlining

```
void h();
g() { h(); return 1;}
f() { int i=g(); return i+1; }
```

- Call(f) ::
  call(g) ::
  call(h) :: return(h) ::
  return(g) ::
  return(f) :: nil

```
void h();
f() { int i=(h(), 1);
      return i+1; }
```

- Call(f) ::
  call(h) :: return(h) ::
  return(f) :: nil

- Events are removed *in matching pairs*

Function inlining

$$(\mathsf{call}(f) \cdot T, \theta) \rightsquigarrow (T, \mathsf{call}(f) \cdot \theta)$$
$$(\mathsf{ret}(f) \cdot T, \mathsf{call}(f) \cdot \theta) \rightsquigarrow (T, \theta)$$

With θ finite and only containing call events

Coinductively:

$$\epsilon \sqsubseteq_\theta \epsilon$$
$$e \cdot T' \sqsubseteq_\theta t \cdot e \cdot T \quad \text{if } T' \sqsubseteq_{\theta'} T \text{ and } (t \cdot e \cdot T, \theta) \rightsquigarrow^* (e \cdot T, \theta')$$
$$\epsilon \sqsubseteq_\theta T \quad \text{if } \epsilon \sqsubseteq_{\theta'} T' \text{ and } (T, \theta) \rightsquigarrow^+ (T', \theta')$$

- If T' $\sqsubseteq_\theta$ T, then
  - for *t'* finite prefix of *T'* there is *t* finite prefix of *T* such that $V_M(t') - V_M(\theta) \le V_M(t)$
  - So, $W_M(T') - W_M(\theta) \le W_M(T)$
- Thus, it suffices to prove that for any T' of the target, there is T of the source such that T' $\sqsubseteq_\varepsilon$ T

Tailcall recognition

```
int h();
int g(x)
 { return h(x+1); }
int f(x)
 { return g(x+2); }
```

- Call(f) ::
  call(g) ::
  call(h) :: return(h) ::
  return(g) ::
  return(f) :: nil

- Caller produces return event *before* transferring to tail-callee
- call(f) :: return(f) :: call(g) :: return(g) :: call(h) :: return(h)

Tailcall recognition

With θ finite and only containing return events

Coinductively:

$$\epsilon \sqsubseteq_\theta \epsilon$$

$$e \cdot T' \sqsubseteq_\theta e \cdot T \quad \text{if } T' \sqsubseteq_\theta T$$

$$\mathsf{ret}(f) \cdot \epsilon \sqsubseteq_\theta \epsilon$$

$$\mathsf{ret}(f) \cdot e \cdot T' \sqsubseteq_\theta e \cdot T \quad \text{if } T' \sqsubseteq_{\mathsf{ret}(f)\cdot\theta} T$$

$$T' \sqsubseteq_{\mathsf{ret}(f)\cdot\theta} \mathsf{ret}(f) \cdot T \quad \text{if } T' \sqsubseteq_\theta T$$

- If T' $\sqsubseteq_\theta$ T, then
  - for *t'* finite prefix of *T'* there is *t* finite prefix of *T* such that $V_M(t') + V_M(\theta) \leq V_M(t)$
  - So, $W_M(T') + W_M(\theta) \leq W_M(T)$
- Thus, it suffices to prove that for any T' of the target, there is T of the source such that T' $\sqsubseteq_\varepsilon$ T

Function inlining and tailcall recognition

- Need to modify simulation diagrams to take special refinement relations into account
- Proof in progress

Mach configuration

- Continuations
  K ::= Knil | Kcons(SP, f, code, ~~RA~~, K)
- Configurations
  C ::= State(mem, rset, SP, f, code, K)
      | Callstate(mem, rset, f, K)
      | Returnstate(mem, rset, K)
- Callstate/Returnstate correspond to CompCert assembly Pallocframe/Pfreeframe pseudos

Mach2 configuration

- Continuations
  K ::= Knil | Kcons(f, code, RA, K)
- Configurations
  C ::= State(mem, rset, SP, f, code, K)
      | Callstate(mem, rset, f, K)
      | Returnstate(mem, rset, K)
- Callstate/Returnstate do not modify the stack

Thank you!

- Paper (accepted to PLDI 2014, to appear), TR, Coq development and artifact VM: http://cs.yale.edu/~tahina/certikos/stack
- For any questions: tahina.ramananandro@yale.edu