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

Quentin Carbonneaux
Tahina Ramananandro
Jan Hoffmann
Zhong Shao

Yale University

April 14th, 2014
Does this program safely run?

```c
#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

in EDN magazine, October 25, 2013

Toyota's killer firmware isn't design gone wrong.

The Problem

Toyota's firmware 2.5.0.3 contains a critical vulnerability that can be exploited to gain full control of a Toyota's car's electronics. This firmware is installed in over 10 million vehicles worldwide, making it a significant security risk. The vulnerability allows an attacker to gain access to the vehicle's keyless entry and remote start systems, enabling them to control the vehicle remotely. This can lead to theft or hijacking of the vehicle, with possible consequences ranging from personal safety to financial losses.

The Solution

To address this issue, Toyota has issued a service bulletin to dealerships, advising them to update the firmware to version 2.5.0.4. This update fixes the vulnerability and prevents unauthorized access to the vehicle's systems. Toyota has also encouraged owners to update their firmware as soon as possible to protect their vehicles from potential security threats.

In conclusion, while the Toyota firmware vulnerability is a serious issue, it is not due to bad design. Instead, it is a result of a lack of proper testing and security measures during the development process. Toyota's quick response to the issue, by issuing a service bulletin and recommending an update, demonstrates their commitment to addressing the problem and protecting their customers.

Software

The EDN software library contains the source code of the technical solution. It follows a list of all the key features:

- goalkeeper: Toyota's firmware 2.5.0.3 contains a critical vulnerability that can be exploited to gain full control of a Toyota's car's electronics. This firmware is installed in over 10 million vehicles worldwide, making it a significant security risk. The vulnerability allows an attacker to gain access to the vehicle's keyless entry and remote start systems, enabling them to control the vehicle remotely. This can lead to theft or hijacking of the vehicle, with possible consequences ranging from personal safety to financial losses.

- Toyota's quick response to the issue, by issuing a service bulletin and recommending an update, demonstrates their commitment to addressing the problem and protecting their customers.
Toyota's killer firmware: Bad design and its consequences

Michael Dunn - October 28, 2013

109 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.

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.
[... ] 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- )
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

Source Program
\[ s \]

Verified Compiler
\[ \text{Verified } s \]

Verified Quantitative Hoare Logic
\[ \forall M : \forall W_M(s) \leq \beta(M) \]

Weight Bound
\[ \beta : (\mathcal{E} \rightarrow \mathbb{Z}) \rightarrow \mathbb{N} \]

Target Program
\[ C(s) \]

Event Metric
\[ M_s : \mathcal{E} \rightarrow \mathbb{Z} \]

Stack-Usage Bound
\[ \text{stack}(C(s)) \leq \beta(M_s) \]
Overview

Source Program $s$

Verified Quantitative Hoare Logic

Weight Bound
$\beta : (E \rightarrow \mathbb{Z}) \rightarrow \mathbb{N}$

Verified Compiler

Target Program $C(s)$

Event Metric $M_s : E \rightarrow \mathbb{Z}$

Safety Proof
$C(s) \sqsubseteq s$

Stack-Usage Bound
$\text{stack}(C(s)) \leq \beta(M_s)$
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

```c
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 $e ::= \ldots \mid \text{call}(f) \mid \text{return}(f)$

- Event and trace valuation:
  
  $V_M(\text{call}(f)) = M(f); V_M(\text{return}(f)) = -M(f)$;
  
  $V_M(e) = 0$ otherwise

  $V_M(\text{nil}) = 0; V_M(e::t) = V_M(e) + V_M(t)$

- 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:
  \[ V'_M(e) = V_M(e) \text{ for call/return} \]
  \[ V'_M(\text{nil}) = 0; V'_M(t++,e::\text{nil}) = \max( V'_M(t), V_M(t)+V'_M(e) ) \]

- 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 $\beta$ 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
int g(int y);

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

- 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) , %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)
  \[ r := \text{EAX} \mid \text{EBX} \mid \text{ECX} \mid \text{EDX} \mid \text{FP0} \]

- Statements (r* registers, ofs constant integer)
  \[ S ::= \text{Mload(chunk, raddr, rres)} \]
  \[ | \quad \text{Mstore(chunk, raddr, rval)} \]
  \[ | \quad \text{Mgetstack(chunk, ofs, rres)} \]
  \[ | \quad \text{Msetstack(chunk, ofs, rres)} \]
  \[ | \quad \text{Mgetparam(chunk, ofs, rres)} \]
  \[ | \quad \text{Mcall \ func} \quad | \quad \text{Mret} \]
  \[ | \quad \text{Mgoto label} \quad | \quad \text{Mlabel label:} \quad | \quad \ldots \]
Mach vs. Mach2

```c
int g(int y);

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

```c
int g {...}

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

Source Program
\[ s \]

Verified Compiler
\[ C(s) \]

Target Program
\[ C(s) \]

Event Metric
\[ M_s : \mathcal{E} \to \mathbb{Z} \]

Verified Quantitative Hoare Logic
\[ \forall M : W_M(s) \leq \beta(M) \]

Weight Bound
\[ \beta : (\mathcal{E} \to \mathbb{Z}) \to \mathbb{N} \]

Safety Proof
\[ C(s) \models s \]

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

\[ \frac{\{P\} S_1 \{R\} \quad \{R\} S_2 \{Q\}}{\{P\} S_1; S_2 \{Q\}} (Q:SEQ) \]

\[ \frac{c \geq 0 \quad \{P\} S \{Q\}}{\{P + c\} S \{Q + c\}} (Q:FRAME) \]

\[ \frac{P \geq P' \quad \{P'\} S \{Q'\} \quad Q' \geq Q}{\{P\} S \{Q\}} (Q:CONSEQ) \]
Selected rules

\[ \Gamma(f) = (P, Q) \]
\[ \Gamma \vdash \{P + M(f)\} f() \{Q + M(f)\} \quad (Q: \text{CALL}) \]

\[ \Gamma' = \Gamma, f : (P_f, Q_f) \quad \Sigma(f) = S_f \quad \Gamma' \vdash \{P\} S \{Q\} \quad \Gamma' \vdash \{P_f\} S_f \{Q_f\} \quad (Q: \text{ABSTRACT}) \]

\[ \Gamma \vdash \{P\} S \{Q\} \]
Selected rules

With:

- Global variable addresses $\Delta$
- Loop break
- Mutable state ($\theta$, $H$)
- Return value
- One argument

those rules become:

\[
\Gamma(f) = (P_f, Q_f) \quad P = \lambda(\theta, H) \cdot P_f(\llbracket E \rrbracket^\Delta_{\theta,H}, H) \quad Q = \lambda(\theta, H) \cdot Q_f(\llbracket x \rrbracket^\Delta_{\theta,H}, H) \\
\Gamma \vdash \{P + M(f)\} x = f(E) \{(Q + M(f), \bot, \bot)\} \quad (Q:\text{CALL})
\]

\[
\Gamma' = \Gamma, f : (P_f, Q_f) \quad \Sigma(f) = (x, S_f) \quad \Gamma' \vdash \{P\} S \{Q\} \quad \Gamma' \vdash \{P'\} S_f \{\bot, \bot, Q'\} \quad P' = \lambda(\theta, H) \cdot P_f(\theta(x), H) \quad Q' = \lambda(\theta, H) \cdot \lambda r . Q_f(r, H) \\
\Gamma \vdash \{P\} S \{Q\} \quad (Q:\text{ABSTRACT})
\]

But we also support:

- Several function arguments
- Auxiliary state
- Stack framing

See paper for more details.
Example with auxiliary state

\{ Z = \log_2(h_\sigma - l_\sigma) \Rightarrow M_b \cdot Z \}\n
bsearch(x, l, h) \{
  \text{if } (h-l <= 1) \text{ return } l;
  \{(Z > 0 \land Z = \log_2(h_\sigma - l_\sigma)) \Rightarrow M_b \cdot Z\}
  m = l + (h-l)/2;
  \{(Z > 0 \land Z = \log_2(h_\sigma - l_\sigma) \land m_\sigma = \frac{h_\sigma + l_\sigma}{2}) \Rightarrow M_b \cdot Z\}
  \text{if } (a[m] > x) \text{ h=m } \text{ else } l=m;
  \{[Z-1 = \log_2(h_\sigma - l_\sigma) \Rightarrow M_b \cdot (Z-1)] + M_b\}
  \text{return } \text{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 →* C', and for any metric M, \( W_M(t) \leq P(C, M) \)

- If \{P\} S \{Q\} is derivable, then for any \( \sigma \), (S, Kstop, \( \sigma \)) consumes at most P stack space
  - Stronger soundness: for any K, \( \sigma \)
    if (skip, K, \( \sigma \)) consumes at most Q stack space, then (S, K, \( \sigma \)) consumes at most P stack space

- Logic and soundness: 700 LOC
  Instantiation to Clight: 950 LOC
Accuracy

- **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

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Verified Stack Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>recid()</td>
<td>8a bytes</td>
</tr>
<tr>
<td>bsearch(x, lo, hi)</td>
<td>40(1 + log₂(hi - lo)) bytes</td>
</tr>
<tr>
<td>fib(n)</td>
<td>24n bytes</td>
</tr>
<tr>
<td>qsort(a, lo, hi)</td>
<td>48(hi - lo) bytes</td>
</tr>
<tr>
<td>filter_pos(a, sz, lo, hi)</td>
<td>48(hi - lo) bytes</td>
</tr>
<tr>
<td>sum(a, lo, hi)</td>
<td>32(hi - lo) bytes</td>
</tr>
<tr>
<td>fact_sq(n)</td>
<td>40 + 24n² bytes</td>
</tr>
<tr>
<td>filter_find(a, sz, lo, hi)</td>
<td>128 + 48(hi - lo) + 40 log₂(BL) bytes</td>
</tr>
</tbody>
</table>

**Table 1.** Manually verified stack bounds for C functions.

---

![Graph of fact_sq(x)](image1)

![Graph of bsearch(v, lo, hi), x = hi - lo](image2)
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 \( \text{liftO} \{A B C : \text{Type}\} (f: A \to B \to C) (\text{ox}: \text{option A}) (\text{oy}: \text{option B}): \text{option C} := \ldots \)

- Fixpoint \( B M \Gamma (s: \text{stm}): \text{option nat} := \)
  
  \[
  \begin{align*}
  \text{match } s \text{ with} & \\
  | \text{scall } _{f \_} &= \text{liftO plus (Some (M f)) (}\Gamma f) \\
  | \text{sseq } s1 s2 &= \text{liftO max (B M }\Gamma s1) (B M \Gamma s2) \\
  | \text{sif } _{st sf} &= \text{liftO max (B M }\Gamma \Phi st) (B M \Gamma sf) \\
  | \text{sloop } s &= \text{B M }\Gamma s \\
  | _{=} &= \text{Some 0} 
  \end{align*}
  \]

- Lemma \( \text{sound_B} : \)
  
  \[
  \begin{align*}
  \forall M \Gamma (CVALID: \text{valid_bctx }M \Gamma) s n \\
  (BS: B M \Gamma s = \text{Some n}), \\
  \text{valid_bound }M s n.
  \end{align*}
  \]

Proof.
- induction \( s \); intros; ...
- apply \( \text{sound\_skip} \).
- apply \( \text{sound\_ret} \) with \( Q := \text{fun } _{=} \Rightarrow \text{mkassn} 0 \).
- apply \( \text{sound\_break} \).
- + ... apply \( \text{sound\_seq} \) with \( Q := \text{fun } _{=} \Rightarrow \text{mkassn} (\text{max } x y) \) ...
- apply valid_max_l ...
- apply valid_max_r ...
- + case_eq \( (\Gamma f) \) ... eapply valid_le; [ apply Le.le_n_Sn ].
- eapply \( \text{sound\_consequence} \);
  
  [] apply \( \text{sound\_call2} \)
  
  with \( (C := \Gamma) 
  
  (Pg := \text{fun \_ pre phi}) 
  
  (Qg := \text{fun \_ post phi}) 
  
  (L := \text{fun } _{=} \Rightarrow \text{True}) ].
- ... eapply \( \text{CVALID} \); eauto.
- + eapply \( \text{sound\_consequence} \);
  [] eapply \( \text{sound\_loop} \)
  
  with \( (I := \text{fun } _{=} \Rightarrow \text{mkassn} n) 
  
  (Q := \text{fun } _{=} \Rightarrow \text{mkassn} n) \)
  
  ]; unfold \text{mkassn}; intuition. ...
- eapply IHs; eauto.
- Qed.
Automatic stack analyzer: soundness

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

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

- Lemma sound_B:
  \forall M \Gamma (CVALID: valid_bctx M \Gamma) s n
  \exists BS: B M \Gamma 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

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

**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.ramanananandro@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 \textit{in matching pairs}
Function inlining

\[(\text{call}(f) \cdot T, \theta) \leadsto (T, \text{call}(f) \cdot \theta)\]
\[(\text{ret}(f) \cdot T, \text{call}(f) \cdot \theta) \leadsto (T, \theta)\]

With \(\theta\) finite and only containing call events

Coinductively:

\[
\begin{align*}
\epsilon &\subseteq_{\theta} \epsilon \\
\epsilon \cdot T' &\subseteq_{\theta} t \cdot e \cdot T \\
\text{if } T' &\subseteq_{\theta} T \\
\text{and } (t \cdot e \cdot T, \theta) &\leadsto^* (e \cdot T, \theta') \\
\epsilon &\subseteq_{\theta} T \\
\text{if } \epsilon &\subseteq_{\theta} T' \\
\text{and } (T, \theta) &\leadsto^+ (T', \theta')
\end{align*}
\]

- If \(T' \subseteq_{\theta} T\), 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 \(T' \subseteq_{\varepsilon} T\)
Tailcall recognition

```c
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 $\theta$ finite and only containing return events

Coinductively:

$$
\begin{align*}
\epsilon & \equiv_\theta \epsilon \\
e \cdot T' & \equiv_\theta e \cdot T \\
\text{ret}(f) \cdot \epsilon & \equiv_\theta \epsilon \\
\text{ret}(f) \cdot e \cdot T' & \equiv_\theta e \cdot T \\
T' & \equiv_{\text{ret}(f) \cdot \theta} \text{ret}(f) \cdot T
\end{align*}
$$

• If $T' \subseteq_{\theta} T$, 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 $T' \subseteq_{\varepsilon} T$
Function inlining and tailcall recognition

• Need to modify simulation diagrams to take special refinement relations into account

• Proof in progress
Mach configuration

- Continuations
  \[ K ::= \text{Kn}il \mid \text{K}cons(\text{SP}, f, \text{code}, \text{RA}, K) \]

- Configurations
  \[ C ::= \text{State}(\text{mem}, \text{rset}, \text{SP}, f, \text{code}, K) \]
  \[ \quad \mid \text{Callstate}(\text{mem}, \text{rset}, f, K) \]
  \[ \quad \mid \text{Returnstate}(\text{mem}, \text{rset}, K) \]

- Callstate/Returnstate correspond to CompCert assembly Pallocframe/Pfreeframe pseudos
Mach2 configuration

• Continuations
  \( K ::= \text{Knil} | \text{Kcons}(f, \text{code}, \text{RA}, K) \)

• Configurations
  \( C ::= \text{State}(\text{mem}, \text{rset}, \text{SP}, f, \text{code}, K) \)
  \ | \text{Callstate}(\text{mem}, \text{rset}, f, K)
  \ | \text{Returnstate}(\text{mem}, \text{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