Trustworthy Hardened Code

Greg Morrisett
Dartmouth -- April 2014
Much Help

- Gang Tan (Lehigh)
- Vikram Adve (Illinois)
- Andrew Myers (Cornell)
- Stephen Chong (Harvard)
- Jean-Baptiste Tristan (Oracle)
- Paul Govereau (PhD student)
- Dan Huang (PhD student)
- Joseph Tassarotti (Ugrad)
- Edward Gan (Ugrad)
- Uifar Elingsson (Google)
- Brad Chen (Google)
- Stephen McCamant (U.Min.)
- NSF
- Google
- AFOSR
- ONR
- IARPA
The Problem

- Our computing infrastructure is built with C/C++
- Yet, these languages are *extremely* error prone:
  - buffer overruns, integer overflows, double frees
  - space leaks, format strings, null pointer dereferences
  - deadlocks, race conditions, …
- There are enough issues at the application level that we shouldn’t still be having to worry about language-level problems.
When I was younger...

• Just re-code everything in a decent language!!!
  - e.g., something that’s type-safe such as Clu or Ada or Modula 3 or ML or Racket or Haskell or …

• Alas, too expensive.
  - RedHat circa 2000: ~30 Mloc
  - WinXP: ~50 Mloc

• Additionally, type-safe languages have lots of C/C++ code hidden in them!
  - > 1Mloc in JDK
A number of companies offer static analysis tools for identifying language-level bugs in C/C++.  
- Microsoft’s Prefix/Prefast, Coverity, HP’s Fortify, ...

My experience:
- good at finding some bugs
- but far too many false positives
- (and also false negatives)
An alternative:

Inlined Reference Monitors

- Formulate a safety policy.
- Insert run-time checks into the code to enforce the policy.
- Using static analysis, try to optimize away the run-time checks.
Some Example Policies

- Stackguard [Wagle & Cowan], MS /GS switch
- SFI: Software Fault Isolation [Wahbe et al.]
- CFI: Control-Flow Isolation [Abadi et al.]
- XFI: Extended Flow Isolation [Erlingsson et al.]
- SafeCode [Dhurjati et al.], Softbound [Nagarakatte et al.]
- Byte Granularity Isolation [Castro et al.]
Policy Tradeoffs

1. What vulnerabilities are mitigated?
2. How much code do we break?
3. How much overhead do we incur?
4. How hard is it to get the implementation right?
Policy Tradeoffs

1. What vulnerabilities are mitigated?
2. How much code do we break?
3. How much overhead do we incur?
4. How hard is it to get the implementation right?
   - Focus for this talk.
IRMIs are compilers.
- Translate untrusted code so that it (hopefully) respects the policy.
- For richer policies, we must optimize the code to avoid the overheads of run-time checks.

Compilers are not something you want to trust.
- see Yang et al’s 2011 PLDI paper.

But, there is hope!
- c.f., Xavier Leroy’s CompCert
Certified Inlined Reference Monitors

In practice, it’s very hard to prove the correctness of a production compiler.

But, we can fall back on *translation validation*:

- build a separate checker that the IRM rewriter produced code that respects the policy.
- prove the correctness of the checker.
- I’ll discuss a couple of certified IRMs we have developed for SFI and SafeCode.
A basic sandbox integrity policy.

- all jumps are constrained to a segment of memory
- all writes are constrained to a (separate) segment
- [optionally, all reads are constrained]

Example applications:

- stored procedures in DB
- native code plug-ins for browsers (e.g., Google’s NaCl)
- in-kernel device drivers (e.g., Nooks)
- isolating native code for run-times (e.g., Robusta)
Why SFI instead of OS?

Communication costs:

- We want to run some untrusted code in the context of a bigger application.
  - e.g., DB, browser, kernel, JVM, …

- Serializing and copying data, context switching, etc. too expensive for the number of cross-domain communications.

- Prevent unmediated access to both OS-level and application-level resources.
Wahbe et al. (1994)

Rewrite MIPS assembly code so that it respects sandbox policy when executed.

- mask high bits of all effective addresses so they are forced to be in the proper segment.

\[ \text{Mem}[A] := r \rightarrow t := \text{mask}(A); \text{Mem}[t] := r \]

Problem: code might jump over masking operation.

- we need the masking and deference to be “atomic”
Solution for the MIPS

- Dedicate two registers: 1 for data (D), 1 for control (C).

- Invariant: dedicated registers *always* point into the proper segment.
  - so all writes to D and C are mask instructions.
  - to store \( r \) at address \( A \): \( D := \text{dmask}(A); \text{Mem}[D] := r \)
  - to jump to address \( A \): \( C := \text{cmask}(A); \text{goto} \ C \)

- If an attacker jumps over masking operations, the code still stays in the sandbox.
  - it just uses the wrong value of D or C
  - but they are always pointing to “valid” addresses
SFI Pros and Cons

• Works on arbitrary assembly code.
  - much easier integration

• Doesn’t break existing C/C++ code.

• High performance:
  - Wahbe et al. claimed 4% overhead on the MIPS for jump/write protection.
  - order of magnitude cheaper RPC.

• But a very weak policy compared to type safety.
What about x86?

- For 32-bit x86 machines, can’t afford to burn 2 registers.

- But, we have segment registers!
  - just need to make sure code doesn’t change segment registers or use segment override prefix.

- More serious problem for both 32 and 64-bit x86:
  - Variable length instructions.
  - there are *multiple* parses we must consider.
```c
int main() {
    int x = 27;
    x += x * x;
    return x;
}
```

```
5589 e583 ec10 c745 fc1b 0000
008b 45fc 8d50 018b 45fc 0faf
c289 45fc 8b45 fcc9 c390 9090
```
SFI for Complex Instruction Sets

- McCamant & Morrisett (2006)
- Goal: force a single parse of the code.
- All direct jumps must be to the beginning of an instruction in our parse.
- For computed jumps:
  - insert no-ops until we are on a k-byte aligned boundary
  - mask the destination address so it is k-byte aligned
- Now just check single parse is okay.
  - Use segment registers on 32-bit machine: 5% overhead
  - Explicit masking on 64-bit machine: 20% overhead
Google’s Native Client (NaCl)

• Yee et al. (2009)

• New SFI service in Chrome browser.
  ▪ load and run x86 executable

• Modified GCC tool-chain
  ▪ inserts appropriate masking, alignment

• Pepper API
  ▪ access to the browser, DOM, 3D acceleration, etc.
One key issue

• When loading the binary, Google checks that it has been properly rewritten.
  ▪ Google’s checker is hand-written C code that includes a (partial) x86 decoder.
  ▪ Nice! Eliminates GCC toolchain from TCB.

• But, a bug in the checker could result in a security breach.
  ▪ Earlier implementations of SFI had bugs
We built a new checker for (32-bit) NaCl that we call RockSalt.

- based on an idea from Seaborn.
- smaller: 80 lines of C versus 600
  - basically a driver operating over automatically generated tables
- faster: on 200Kloc of compiled C code
  - Google’s original checker: 0.9s vs. RockSalt: 0.2s
- stronger: (mostly) proven correct
  - table generation proven correct
  - ML driver proven correct, but manually translated to C
How RockSalt works

• Specify regexps for parsing legal x86 instructions
  ▪ preclude instructions that might change or override the segment registers.
  ▪ preclude doing a computed jump without first masking the effective address of the destination.

• Compile regexps to a table-based DFA
  ▪ interpret DFA tables &
  ▪ record start positions of instructions &
  ▪ check jump and alignment constraints
What we proved...

• If we give the checker a string of bytes $B$, and the checker accepts, then if we load $B$ into an appropriate x86 context and begin execution, the code will respect the sandbox policy.

• The real challenge is building a formal model of x86 code execution.
  - And to gain some confidence that the model is correct!
The x86 Model

Use domain-specific languages to specify the semantics.

• Decoder:
  ▪ type-indexed parsing combinators for regular grammars
  ▪ easy denotational semantics
  ▪ operational semantics via derivatives
  ▪ proof of adequacy/soundness

• Execution:
  ▪ register transfer language (think GCC)
  ▪ translate x86 instructions into RTLs
  ▪ give operational semantics for RTLs
Our x86 Model in Coq

Machine States

Decoder

Instruction Abstract Syntax

Translator

RTL: RISC-based Core

RTL interpreter

Essentially, a big interpreter coded in a (purely) functional language.
Specifying the Decoder

- Instruction Abstract Syntax
- Decoder
- Machine States
- Translator
- RTL: RISC-based Core
- RTL interpreter

- x86 grammar
  - typed regular grammars
  - declarative grammar semantics
  - optimizations + proofs of correctness
  - execution semantics (derivative-based parser)
  - adequacy of execution semantics

Grammar DSL
Specifying the Decoder

- Instruction Abstract Syntax
- Decoder
- Machine States
- RTL: RISC-based Core
- RTL interpreter

- x86 semantics
- Semantic DSL
- Typed register transfers
- Operational semantics
- Optimizations + proofs of correctness
- Symbolic reasoning
Validating the Model

- We can extract an executable Ocaml program to run x86 binaries from the specification.

- We used Intel’s PIN to instrument binaries so we could dump out register states and compare.

- Used Csmith to generate random C programs, compile, test against implementations.
  - ~10M instructions in a few hours

- Used parser spec to generate fuzz tests.
The Proofs

• Need to show:
  ▪ when checker says “yes” on byte sequence B
  ▪ then when B is loaded into a suitable execution environment, it will respect the sandbox policy.

• Must relate the parsing that the checker does (i.e., DFA) to the parsing that the semantics does (i.e., denotational semantics).
  ▪ Need an inversion principle so that we can reason about the possible forms of instructions we might see.
  ▪ then argue that those instructions, when translated, and executed, respect invariants.
The usual proof in Coq is done largely by hand.

This just doesn’t scale when we have hundreds of instructions, which each translate into tens of RTL instructions.

So we used a combination of reflection and tactics to heavily automate the proof.
Reflection

• Example: prove that grammar is unambiguous.

• Construct a program in Coq:
  \[ \text{unambig} : \text{grammar } t \rightarrow \text{bool} \]

• Prove that \text{unambig} is correct:
  \[ \text{unambig } g = \text{true} \rightarrow (s,v1) \in [[g]] \rightarrow (s,v2) \in [[g]] \rightarrow v1 = v2. \]
Some Related Work

- PittSFeld (our earlier work)
  - formalized a small model (7 instructions) and proved that the alignment variants implied the sandbox policy
  - but Kroll & Dean found bugs in the implementation’s decoder!

- Armor (Zhao et al)
  - verified SFI tools for the ARM
  - verification condition generator & abstract interpretation, so much more semantic in character
  - but also much more expensive

- x86 modeling work at Cambridge (Myreen, Fox, et al.)
  - inspiration for many of the ideas here

- Also \( \lambda \)-RTL and SLED (Ramsey)
Summary thus far…

• RockSalt is a new checker for Google’s SFI
  - “declarative” formulation of policy
  - very fast checker
  - machine-checked proof of correctness

• This required an enormous amount of engineering for 80 lines of code!
  - 5K lines of definitions; 10K lines of proof
  - but almost all of that was defining an x86 model
  - and then validating the model
SFI is the simplest kind of IRM we can do.
- easy to apply
- doesn’t break code
- but it’s an extremely weak as a policy

Can we do something much stronger?
Secure Virtual Architecture [Adve]

Based on the SAFEcode compiler [PLDI’06]:

- Compiles C, C++, Java, Haskell, etc.
- Enforces a much stronger policy similar to type safety.
- Works by instrumenting the LLVM intermediate representation + some runtime support.
- Has been used to compile the Linux 2.4.22
  - required < 300 changed lines
  - prevented 4 of 5 known vulnerabilities
  - ~20% - 50% overhead
SAFEcode Policy

• Ensures a control-flow integrity guarantee as in type-safe languages (e.g., Java).

• Pointers are either typed or untyped.
  ▪ $p : T^*$ means $p$ is guaranteed to point to a valid $T$
  ▪ $p : ?^*$ means $p$ could be any value

• Loads/stores/casts for untyped pointers must be dynamically checked.
  ▪ memory broken into typed regions
  ▪ splay tree supports fast range checks
SAFEcode Pros and Cons

- Compared to SFI, will break some code.
- Compared to SFI, higher run-time costs.
- Compared to SFI, much harder to implement.
- A much stronger policy:
  - strictly subsumes SFI, CFI, XFI.
  - stops most of the common vulnerabilities.

Claim: about as strong as you can get without breaking too much code or incurring too much overhead.
Two Checkers

We reduce the TCB through the use of 2 checkers:

1. a type-checker that ensures we instrumented the code correctly.
   - We’ve proven this type-checker correct, along similar lines to what I described for SFI but for a *much* more complicated policy.
   - Again, a primary challenge was building and validating a model (an LLVM abstract machine) but this was *much* easier than the x86.

2. a general translation validator that checks whether the input to the optimizer is equivalent to the output.
   - A big chunk is formalized, but not yet proven correct.
Certification for SAFEcode

C/C++ → Clang → LLVM IR → SAFEcode analysis & transforms → LLVM IR → enhanced types

LLVM IR → LLVM optimizer

LLVM IR → type checker

equiv checker → LLVM code generator

Binary
• Translation validator for LLVM’s optimizer
  ▪ A tool that tries to prove that a function is equivalent to its optimized version.
  ▪ If we can’t prove the functions are equivalent, we use the unoptimized function.
  ▪ Based on ideas from Tate et al’s equatlity-based saturation (2009).

• Gives us a trivial proof that the optimizer is correct.
How do we do this?

- Convert LLVM’s intermediate language into a *categorical graph representation*.
  - integrates control- and data-flow into a uniform representation
  - think circuit representations of computations
  - but incorporates loops by lifting everything to the level of streams of values.

- Normalize graphs using a set of rewrite rules, and then compare for equivalence.
  - this gives us many equivalences for free, such as common sub-expressions and loop-invariant computations.
We considered a pipeline of the following intra-procedural optimizations (taken as a black box):

- ADCE (advanced dead code elimination)
- GVN (global value numbering)
- SCCP (sparse-condition constant propagation)
- LICM (loop invariant code motion),
- LD (loop deletion)
- LU (loop unswitching)
- DSE (dead store elimination)
% SPEC Functions Validated

% SPEC functions validated by LLVM-MD

<table>
<thead>
<tr>
<th>Function</th>
<th>alarm</th>
<th>fail</th>
<th>ok</th>
<th>boring</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqlite3</td>
<td>150</td>
<td>32</td>
<td>683</td>
<td>497</td>
</tr>
<tr>
<td>bzip2</td>
<td>20</td>
<td>11</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>h264ref</td>
<td>154</td>
<td>23</td>
<td>315</td>
<td>116</td>
</tr>
<tr>
<td>hmmer</td>
<td>99</td>
<td>19</td>
<td>321</td>
<td>207</td>
</tr>
<tr>
<td>lbm</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>libquantum</td>
<td>46</td>
<td>2</td>
<td>45</td>
<td>22</td>
</tr>
<tr>
<td>mcf</td>
<td>8</td>
<td>2</td>
<td>12</td>
<td>45</td>
</tr>
<tr>
<td>milc</td>
<td>45</td>
<td>9</td>
<td>109</td>
<td>74</td>
</tr>
<tr>
<td>perlbench</td>
<td>327</td>
<td>112</td>
<td>920</td>
<td>604</td>
</tr>
<tr>
<td>sjeng</td>
<td>23</td>
<td>13</td>
<td>79</td>
<td>51</td>
</tr>
<tr>
<td>sphinx</td>
<td>62</td>
<td>10</td>
<td>219</td>
<td>97</td>
</tr>
<tr>
<td>gcc</td>
<td>1108</td>
<td>230</td>
<td>2615</td>
<td>1732</td>
</tr>
</tbody>
</table>
Validation time relative to Compile

Takeaway: expensive to use on each compile, but not too expensive for a release or in debugging an optimization.
Some Related Work

• Nagarakatte et al’s Softbound [PLDI’09]
  - Similar object-level memory safety guarantee.
  - Proved correctness of the instrumentation.
  - Constructed a model of LLVM IR.

• UCSD’s Peggy [POPL’09]
  - Inspired our work.
  - They generate all equivalent programs and share representations.
  - More complete, rewrite orderings don’t matter.
  - But much slower (10-100x) than directed rewrites.
Summary

• SAFEcode provides a good policy for C/C++
  ▪ enforces an approximation to type-safety
  ▪ strong enough to stop most language-level attacks
  ▪ but weak enough to admit real C/C++ code

• Our tools help minimize the SAFEcode TCB
  ▪ verified type-checker for intermediate representation
  ▪ translation validator for the optimizer

• Again, a huge amount of modeling work:
  ▪ in this case, for LLVM’s IR, SAFEcode type system
Stepping Back

• I don’t think we’ll ever get rid of the need for low-level C/C++ code.
  ▪ performance, control
  ▪ compatibility (in all respects)

• But I do think we can get similar safety guarantees as with higher-level languages.
  ▪ industry will start with simple things (SFI, CFI).
  ▪ but attackers will force them to adopt stronger policies such as SAFEcode’s or Softbound’s.
On Formal Methods

• We’ve made tremendous progress!
  ▪ verifying real code, not just toy programs

• Major short-term challenges:
  ▪ producing models of real systems (e.g., x86, LLVM)
  ▪ validating those models
  ▪ sharing those models across environments

• Even so, it’s hard.
  ▪ but maybe that’s the pressure we need to get the right architectures in place.