High System-Code Security with Low Overhead

Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder
High System-Code Security?

Today’s software is dangerous.

Example: OpenSSL
Overflow in ssl/t1_lib.c:3997 → ❤️

OpenSSL contains 53,073 memory accesses.
How to protect them all?
Protect *all* dangerous operations using sanity checks
✓ Checks are automatically added at compile time
✓ No source code modification is needed

```c
*p = 42;
if (!isValidAddress(p)) {
    reportError(p);
    abort();
}
*p = 42;
```
Problem: Sanity checks cause high performance overhead

<table>
<thead>
<tr>
<th>Tool</th>
<th>Avg. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressSanitizer (memory errors)</td>
<td>73%</td>
</tr>
<tr>
<td>SoftBound/CETS (full memory safety)</td>
<td>116%</td>
</tr>
<tr>
<td>UndefinedBehaviorSanitizer (integer overflows, type errors…)</td>
<td>71%</td>
</tr>
<tr>
<td>Assertions, code contracts, …</td>
<td>depends</td>
</tr>
</tbody>
</table>
Problem: Sanity checks cause high performance overhead

People use checks heavily for testing, but disable them in production

Goal: checks in production
Insight: Checks are not all equal

Most of the overhead comes from a few expensive checks

Checks in hot code, each executed many times

Most of the protection comes from many cheap checks

Checks in cold code
Our Approach: ASAP

As Safe As Possible

Lets users choose their overhead budget (e.g., 5%)
Automatically identifies sanity checks in software
Analyzes the cost of every check
Selects as many checks as fit in the user’s budget

Most of the overhead comes from a few expensive checks
Most of the protection comes from many cheap checks
ASAP Insight & Results

A few checks are very expensive.

Most protection comes from cheap checks.

A user on a 5% budget can get 87% of the protection.
Outline

Introduction: What is ASAP?

Design

Key Algorithms

Results

Conclusion
Design

ASAP is built into the compiler

✓ *Easy to use (set CC and CFLAGS)*
✓ *Compatible (parallel compilation, …)*

![Diagram]

- Compiler (LLVM)
  - Identify sanity checks
  - Profiler: measure check costs
  - Optimizer: select maximum set of checks
Users use ASAP like a regular compiler that adds checks.

ASAP stores intermediate compiler output.
Users use ASAP like a regular compiler that adds checks.

ASAP generates a program variant with profiling instrumentation. Users run this to measure check costs.
Users use ASAP like a regular compiler that adds checks

ASAP generates a program variant with profiling instrumentation. Users run this to measure check costs.

ASAP uses costs & budget to generate an optimized program
Outline

Introduction: What is ASAP?

Design

Key Algorithms

Results

Conclusion
Can users trust ASAP to select checks that use the least CPU cycles?

If ASAP says you’re 87% protected, what does this mean?
Measure Check Cost

...  
if (!isValidAddress(p)) {
    reportError(p);
    abort();
}
*p = 42;
...
...
Measure Check Cost

... 1. Add *profiling counters*

prof1++;
if (!isValidAddress(p)) {
    prof2++;
    reportError(p);
    abort();
}
prof3++;
*p = 42;
...
 prof1++;  
if (!isValidAddress(p)) {
    prof2++;  
    reportError(p);  
    abort();
}
prof3++;  
*p = 42;
...

1. Add **profiling counters**
2. Identify **check instructions**
Measure Check Cost

1. Add **profiling counters**
2. Identify **check instructions**
3. Use static model of **cycles per instruction**

```plaintext
... prof1++;
if (!isValidAddress(p)) {
    prof2++;
    reportError(p);
    abort();
}
prof3++;
*p = 42;
... 
```
Measure Check Cost

1. Add **profiling counters**
2. Identify **check instructions**
3. Use static model of cycles per instruction
4. Compute cost for check in CPU cycles:

$$\sum_{i \in \text{check}} \text{prof}(i) \cdot \text{cycles}(i)$$

Precise cost in CPU cycles

... prof1++;
if (!isValidAddress(p)) {
    prof2++;
    reportError(p);
    abort();
}
prof3++;
*p = 42;
...
If ASAP says you’re 87% protected, what does this mean?
ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented
ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented

---

**Experiment 1**

Source code of Python 2.7

Bug: line that has received a patch between version 2.7.1 and 2.7.8
Sanity level 87% ≈ 91% protection

ASAP quantifies protection using the *sanity level*

We would like to know the *effective protection level*

Methodology: measure how many bugs/vulnerabilities are effectively prevented

**Effective Protection ≥ sanity level**
# Experiment 2

## Known bugs

<table>
<thead>
<tr>
<th>Project</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 3.4</td>
<td>3</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>1</td>
</tr>
<tr>
<td>RIPE Benchmarks</td>
<td>10</td>
</tr>
</tbody>
</table>

All of these are in cold code
Experiment 2

**Known bugs**

<table>
<thead>
<tr>
<th>Project</th>
<th>Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 3.4</td>
<td>3</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>1</td>
</tr>
<tr>
<td>RIPE Benchmarks</td>
<td>10</td>
</tr>
</tbody>
</table>

All of these are in cold code

---

Experiment 3

**Vulnerabilities from CVE DB**

Analyze 145 vulnerabilities from 2014

- Memory errors
- Open source
- Patch available
- Error location known

83% of these are in cold code

**Checks in cold code provide real value**
Outline

Introduction: What is ASAP?
Design
Key Algorithms
Results
Conclusion
Results

Overhead for:
SPEC benchmarks
AddressSanitizer

100%
sanity level
Results

Overhead for:
SPEC benchmarks
AddressSanitizer

95% sanary level

CPU cycles saved
Results

Overhead for:
SPEC benchmarks
AddressSanitizer

90% sanity level
Results

Overhead for:
SPEC benchmarks
AddressSanitizer

80% sanity level

Residual overhead (not due to checks)
Conclusion

Run-time checks deliver strong protection at high cost.

Most of the *overhead* comes from a few expensive checks.
Most of the *protection* comes from many cheap checks.
Backup slides

CHECK ALL THE ACCESSES

pay all the overhead?
Sanity Check in LLVM

; <label>:0
%1 = load i32* %fmap_i_ptr, align 4
%2 = zext i32 %1 to i64
%3 = getelementptr inbounds i32* %eclass, i64 %2
%4 = ptrtoint i32* %3 to i64
%5 = lshr i64 %4, 3
%6 = add i64 %5, 17592186044416
%7 = inttoptr i64 %6 to i8
%8 = load i8* %7, align 1
%9 = icmp eq i8 %8, 0
br i1 %9, label %18, label %10

; <label>:10
%11 = ptrtoint i32* %3 to i64
%12 = and i64 %11, 7
%13 = add i64 %12, 3
%14 = trunc i64 %13 to i8
%15 = icmp slt i8 %14, %8
br i1 %15, label %18, label %16

; <label>:16
%17 = ptrtoint i32* %3 to i64
call void @__asan_report_load4(i64 %17) #3
call void asm sideeffect "", ""() #3
unreachable

; <label>:18
%19 = load i32* %3, align 4