Cool Systems Research @ UCI: Making a Billion Users Happier and Safer

Michael Franz
University of California, Irvine
Let's Get Started...

✦ compiler construction
  ✦ is a very well understood field

✦ all existing compilers follow essentially same principle
  ✦ first, build a control-flow graph (CFG)
  ✦ then, traverse the control flow graph while generating code for its hierarchical structures

✦ even embedded and just-in-time compilers follow this model, except that the unit of compilation may be smaller
  ✦ typically, one method at a time
A;
while (i < 1000) do
    if (x < y)
        then D;
    else E;
    F;
    G;
    H;
A;
while (i < 1000) do
    if (x < y)
        then D;
    else E;
    F;
    G;
H;
Traditional Compilers

- several problems with this approach:
  - not all code in a method gets executed equally often, and some code may never get executed at all
  - unfortunately, traditional compilers need to create and traverse whole CFG anyway
  - many CFG-oriented algorithms are very complex, and hence not well suited for just-in-time compilation
  - witness Sun’s longtime distinction between “server compiler” and “client compiler” for Java, which differ in the intermediate data structures and algorithms used internally
  - a compiler can only optimize code that it “sees” simultaneously
  - so for “global” optimization, we need to compile it all together, i.e., build CFG for *everything*
  - cannot look “inside” dynamic link libraries and independent compilation units
Most Code Is “Cold”

- each line represents one basic block
- not a lot is “hot”
A;
while (i < 1000) do
  if (x < y)
    then D;
  else E;
F;
G;
H;
A;
while (i < 1000) do
    if (x < y)
        then D;
    else E;
F;
G;
H;
Idea

- compile only the loops in a program

- leave the rest to an interpreter or a quick and dirty “base line” compiler

- **discover loops** while interpreting the program or while running “base line” program containing instrumentation code

- make a bet that execution will stay on the path that was observed when the loop was discovered

- if we lose the bet, we “bail out” back to the interpreter (but we can make a new bet at that point)
A;
while (i < 1000) do
  if (x < y)
    then D;
  else E;
F;
G;
H;
A;
while (i < 1000) do
  if (x < y)
    then D;
  else E;
F;
G;
H;
Compile Only Loop

A

i ≥ 1000

x < y

D

E

F

G

H

i ≥ 1000

x < y

bailout

E

F

G
Optimize Loop

A

i ≥ 1000

x < y

D

E

F

G

H

EGF

bailout

bailout

i ≥ 1000

x < y
Or Maybe Even...

A

i ≥ 1000

D

x < y

E

F

G

H

loop invariant computation

F

i ≥ 1000

bailout

bailout

x < y

EG
Multiple Loop Paths

- if we bail out too often at the same spot, that means that there is another path through the loop that we should probably compile as well

- record it, compile it, hook it up to the former bail-out point
Too Many Bailouts

A

\[ i \geq 1000 \]

\[ x < y \]

D
E

F
G

H

bailout

i \geq 1000

x < y

bailout

EGF
Add Another Path

i ≥ 1000

x < y

D

E

F

G

H

i ≥ 1000

x < y

bailout

EGF

DFG
Our Contributions

- original idea of tracing invented by Bala, Düsterwald and Banerjia in 1999 ("Dynamo")
  - same-architecture: optimized from PA-RISC to PA-RISC
  - modest performance improvement, mainly by inlining (but without optimizing away stack frames!)
- my group’s contributions:
  - VML to native, oo-dispatch inlining with deep bailouts
  - a tree representation for alternative paths through a loop using a special form of SSA
  - a new way of “stitching together” previously compiled traces
  - an optimal register allocator on trace trees
- actual deployments:
  - Mozilla Firefox [JavaScript], Sun’s Maxine [Java], Microsoft SPUR [DotNet CLR]; related: Android's Dalvik VM
Loop Containing Call

A

i ≥ 1000

D

x < y

F

G

H

o.m() -> C.M() -> a < b

R

S

T

return
Method Call Is Hot Too

A

i ≥ 1000

x < y

D

o.m()

F

G

H

C.M()
a < b

R

S

T

return
Call Is Inlined Into Loop

\[ i \geq 1000 \]

\[ x < y \]

\[ D \]

\[ o.m() \]

\[ F \]

\[ G \]

\[ C.M() \]

\[ a < b \]

\[ return \]

\[ H \]

\[ i \geq 1000 \]

\[ x < y \]

\[ o.cls=C \]

\[ a < b \]

\[ return \]

\[ RFGT \]
Call Is Inlined Into Loop

“deep” bailout, requires construction of C.M()’s stack frame on the fly
Multiple Receivers

A

\[ i \geq 1000 \]

\[ x < y \]

D

\[ \text{om()} \]

F

\[ a < b \]

G

R

S

C.M()

T

U

V

C1.M()

return

return
Dynamic Specialization

this is key to efficient compilation of dynamically typed languages such as JavaScript - often can eliminate boxing and unboxing of primitive data types and perform arithmetic directly in-line
JavaScript

- dynamically typed, type inference is difficult in the general case
- the semantics of "add" depend on its operands, this results in lots of runtime checks

```javascript
var sum = 0
for (var i = 0; i < 1000; i++) {
    if (i == 990) {
        sum += " Hello World "
    }
    sum += 1
}
print(sum)

"990 Hello World 1111111111"
JavaScript

- dynamic compiler optimizes for the case where “sum” is an integer
- after “sum” becomes a string, execution resumes unoptimized in the interpreter

```javascript
var sum = 0
for (var i = 0; i < 1000; i++) {
  if (i == 990) {
    sum += " Hello World "
  }
  sum += 1
}
print(sum)

"990 Hello World 1111111111"
```
So How Did We Get There?

✧ (after the fact: conversation with Düsterwald)
✧ Dynamo team did not optimize away stack frames of inlined calls — needed to respect calling conventions of binary code
✧ mainly as result of this constraint, they never attempted to compile from a high-level language but only implemented same-architecture binary rewriting optimizations
✧ our key idea: a special form of SSA that models traces
✧ insight: if we keep the VM context completely separate from that of compiled code
✧ and if we create explicit read-in instructions from the VM to the SSA trace (these become SSA definitions)
✧ and if we create explicit write-backs from the modified SSA values to the VM
✧ then bailouts and compensation code become simple
Making People Happy
Originally: Java Tracing JIT

- compared to HotSpot, average over a large sample
  - code generation rate is about 350 times faster
  - total code generated is about 1/30th (hot methods contain lots of cold code)
  - total compiler size is about 1/100th
  - total memory footprint is about 1/7th
- these results got our foot in the door with Mozilla
- chicken and egg problem - who pays for the step-up between an academic research prototype and an industrial-strength one?
  - Mozilla invested significant resources to build TraceMonkey
  - after the fact, looks like a no brainer
  - technology would probably still lie dormant if we had tried to charge for it
JavaScript: Browser Wars

- prior to August 2008, no major web browser had a just-in-time compiler for JavaScript
- strange really, considering that they all had JIT compilers for Java
- and considering that the amount of JavaScript processed by a typical browser in a single day was “orders of magnitude larger” than the largest Java “applet” ever seen by such a browser [Brendan Eich quote]
- August 23, 2008: Mozilla launches Firefox 3.1 alpha with our trace compiler (“TraceMonkey”)
- September 1, 2008: Google launches Chrome/V8
- September 18, 2008: Apple releases SquirrelFish Extreme
Inside The Box
Trace Stitching (Dynamo)

Trace Linking
Trace optimizations are possible across trace link edges.

Trace Stitching
No trace optimizations are possible across trace stitch edges.
Trace Trees (Gal & Franz)
Optimizing Trace Trees

- trace trees are re-optimized every time an edge is added
- optimizations cannot be destructive
- even if (2) is dead in the initial trace 1-2-3-5 it must not be removed from the tree
- otherwise 4 could not be fused to it later
- instead, optimizations only flag instructions (i.e. this instruction is dead)
- at re-compilation, flags are cleared
Nested Loops

- inner loop is more likely to be hotter
- leads to “outerlining” effect
Pathological Cases

- many equally luke-warm paths

```c
for (int i = 0; i < 256; i++) {
    if ((i & 1) != 0) { ... }
    if ((i & 2) != 0) { ... }
    if ((i & 4) != 0) { ... }
    if ((i & 8) != 0) { ... }
    if ((i & 16) != 0) { ... }
    if ((i & 32) != 0) { ... }
    if ((i & 64) != 0) { ... }
    if ((i & 128) != 0) { ... }
}
```

- leads to code explosion with trace trees (because of tail duplication)
Pathological Cases

- nested loops ("hot pair")

```java
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 1000; j++) {
        ...
    }
    for (int j = 0; j < 1000; j++) {
        ...
    }
}
```

- never get back to the outer loop when tracing
Nested Trace Trees (Bebenita & Franz)

- capture significantly more complicated control flow and thereby expose more optimization potential
- preserves the SSA form of trace trees and adds little overhead to the compiler pipeline
- each trace tree can be compiled individually, or can be specialized based on nesting location
- optimization results can be propagated up and down the nesting hierarchy
- eliminate most switches between the interpreter and compiled trace trees
- improve register allocation: hottest regions are in leaf nested trees, and are allocated separately
Hot Pairs Revisited

- a possible nesting of tree A and B into C
- guards are inserted into the nesting tree to ensure that the nested tree exits at the same place that was originally recorded (!)

```java
public static int bar() {
    int sum = 0;
    for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
            sum ++;
            for (int i = 0; i < 1000; i++)
                sum ++;
        }
    }
    return sum;
}
```
Hot Pairs Revisited

- A much more likely nesting would be tree A into B
- Tree A is compiled before B
- The trace recorded after a bailout in B nests the previously compiled tree A into B

```java
public static int bar() {
    int sum = 0;
    for (int j = 0; j < 2000; j++) {
        for (int i = 0; i < 1000; i++) {
            sum ++;
        }
    }
    return sum;
}
```
for (var n = 0; n < 1000; n++) {
    for (var n2 = 0; n2 < 1000; n2++) {
        for (var i = 0; i < a.length - 1; i++) {
            var tmp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = tmp;
        }
    }
}
Code Paths Traced (SPUR)
optimized array-double-swap.js

- trace has 10 loop instructions, 2 loop guards!
- speedup from 224ms to 10ms, performance is comparable the C# version of the loop, 7x faster than the CLR operating on JScript code, and slightly faster than V8
Summary

- our approach leads to small, fast compilers
- complex optimizations are significantly simpler on traces, saving code size and execution time
- only really hot code is ever compiled
- still only scraping the surface of what is possible

- the Browser Wars will provide better benchmarking data than any academic could ever produce
  - we expect that Trace Tree compilation will win and that eventually most just-in-time compilers will be based on our scheme
- see discussion “Has Tracing Won?” on the web
Making People Safer
Software Bugs

✦ most software contains errors
✦ some of these errors are security hazards
✦ writing high-assurance (certified error-free) software is extremely labor intensive
✦ over the past several decades, the relative cost of human time vs. computer time has shifted significantly (human time got more expensive)
✦ overall theme of our research: save (relatively expensive and scarce) human time by investing (relatively cheap and abundant) computer time
Problem: Critical Software Bugs

- many software vulnerabilities are “critical”
  - allow the instantaneous complete takeover of computers that have the bug
- bugs of this severity are discovered regularly in widely used software
- deeper question: are there “insiders” who know about the bugs?
Ex: Buffer Overflow Attack

- attacker provides long input that overwrites return address “further down” on stack
- eventual “return” transfers control to payload code

```
char buf[100];
...
gets(buf);
```
Ex: Buffer Overflow Attack

- attacker provides long input that overwrites return address "further down" on stack
- eventual "return" transfers control to payload code

```c
char buf[100];
...
gets(buf);
```
Ex: Buffer Overflow Attack

- attacker provides long input that overwrites return address “further down” on stack
- eventual “return” transfers control to payload code

```c
char buf[100];
...
gets(buf);
```
Isn’t This Trivial & Old?

- one would think so, but unfortunately...
- buffer overflows and related pointer vulnerabilities still account for extremely serious exploits
- sometimes, vulnerabilities exist for extended periods

- e.g., 19 August 2009
  - Researchers uncovered a vulnerability in the Linux kernel that puts most versions built in the past eight years at risk of complete takeover. Affects all 2.4 and 2.6 kernels since 2001 on all architectures.

The bug involves the way kernel-level routines such as sock_sendpage react when they are left unimplemented. Instead of linking to a corresponding placeholder, (for example, sock_no_accept), the function pointer is left uninitialized. Sock_sendpage doesn’t always validate the pointer before dereferencing it, leaving the OS open to local privilege escalation that can completely compromise the underlying machine.
Security Advisory for Adobe Flash Player, Adobe Reader and Acrobat

Release date: March 14, 2011
Last updated: March 21, 2011
Vulnerability identifier: APSA11-01
CVE number: CVE-2011-0609
Platform: All Platforms

Summary

A critical vulnerability exists in Adobe Flash Player 10.2.152.33 and earlier versions (Adobe Flash Player 10.2.154.18 and earlier for Chrome users) for Windows, Macintosh, Linux and Solaris operating systems, Adobe Flash Player 10.1.106.16 and earlier versions for Android, and the Authplay.dll component that ships with Adobe Reader and Acrobat X (10.0.1) and earlier 10.x and 9.x versions of Reader and Acrobat for Windows and Macintosh operating systems.

This vulnerability (CVE-2011-0609) could cause a crash and potentially allow an attacker to take control of the affected system. There are reports that this vulnerability is being exploited in the wild in targeted attacks via a Flash (.swf) file embedded in a Microsoft Excel (.xls) file delivered as an email attachment. At this time, Adobe is not aware of attacks targeting Adobe Reader and Acrobat. Adobe Reader X Protected Mode mitigations would prevent an exploit of this kind from executing.
Security Advisory for Adobe Flash Player, Adobe Reader and Acrobat

Release date: April 11, 2011

Vulnerability identifier: APSA11-02

CVE number: CVE-2011-0611

Platform: See "Affected software versions" section below for details

Summary

A critical vulnerability exists in Flash Player 10.2.153.1 and earlier versions (Adobe Flash Player 10.2.154.25 and earlier for Chrome users) for Windows, Macintosh, Linux and Solaris, Adobe Flash Player 10.2.156.12 and earlier versions for Android, and the Authplay.dll component that ships with Adobe Reader and Acrobat X (10.0.2) and earlier 10.x and 9.x versions for Windows and Macintosh operating systems.

This vulnerability (CVE-2011-0611) could cause a crash and potentially allow an attacker to take control of the affected system. There are reports that this vulnerability is being exploited in the wild in targeted attacks via a Flash (.swf) file embedded in a Microsoft Word (.doc) file delivered as an email attachment, targeting the Windows platform. At this time, Adobe is not aware of any attacks via PDF targeting Adobe Reader and Acrobat. Adobe Reader X Protected Mode mitigations would prevent an exploit of this kind from executing.
Evolution Of Solutions

✦ prevent programming errors in the first place
  ✦ static analysis tools
✦ accept that errors exist, prevent control transfer
  ✦ return address/code pointer integrity checking
✦ accept control transfer, prevent code execution
  ✦ non-executable stack
  ✦ instruction set randomization/encryption
✦ accept execution of enemy’s code, prevent damage
  ✦ multi-variant code execution
Deeper Problem: Insider Threats

✦ how many agents of the Antipodean intelligence service work at Microsoft?
✦ how many members of cyber-crime syndicates?
✦ isn’t it possible that there are organizations that have partial maps of exploitable vulnerabilities in major software systems?
✦ (since such vulnerabilities may also be exploited for defensive purposes, isn’t it even possible that the U.S. Intelligence Community is one of them?)
Insider Threats

- compared to launching a single spy satellite, placing 1,000 agents inside Microsoft is probably dirt cheap
- also, compare the salary of a programmer to the cost of cyber-based identity theft (in U.S. alone, it amounted to US $ 54B in 2009; some not insignificant part of that is the scammers' profit)
- malicious insiders need not actually be creating back-doors themselves — simply reporting vulnerabilities to an outsider rather than to own management is sufficient
Is Open Source Better?

✦ no, the situation may be even more severe
✦ open source teams usually have limited resources and typically cannot afford to employ techniques such as high-end static analysis
✦ but an adversary with substantial resources could build a testing lab that systematically finds vulnerabilities — without anyone knowing
✦ hence, open-source software may lead to **asymmetric threats** in which an attacker knows more about a system than the system’s own creator
Anatomy Of An Exploit

- successful exploit depends on two things:
  - the attacker knows about a place in which the program has an “unspecified” behavior
  - example: a particular buffer will actually accept input greater than its length
  - even though the behavior is “unspecified”, the attacker knows exactly what will happen in this case
- leads to defense strategy of randomization
Randomization

✦ make it more difficult for an attacker to guess behavior in an “unspecified” state
  ✦ for example, Windows OS now includes address-space layout randomization
✦ unfortunately, experiments have shown that due to alignment and kernel restrictions, maximum variability is really only 16 bits
  ✦ susceptible to brute-force attacks
Beyond Randomization

- combine randomization with parallelism and checkpointing:
  - generate several slightly different versions of the same program
  - run these versions in lockstep in parallel on different cores
  - look for discrepancies in behavior
- choose randomization parameters in a way that “symmetric” attacks become difficult
Example: Stack Addressing

- Downward Growing Stack:
  - Caller Data
  - Return Address
  - Frame Pointer
  - Buffer
  - Other Data
  - Free Space

- Upward Growing Stack:
  - Free Space
  - Other Data
  - Buffer
  - Frame Pointer
  - Return Address
  - Caller Data
Example: Stack Addressing

**Downward Growing Stack**
- Caller Data
- Return Address
- Frame Pointer
- Buffer
- Other Data
- Free Space

**Upward Growing Stack**
- Free Space
- Other Data
- Buffer
- Frame Pointer
- Return Address
- Caller Data
Example: Stack Addressing

**Downward Growing Stack**
- Caller Data
- Return Address
- Frame Pointer
- Buffer
- Other Data
- Free Space

**Upward Growing Stack**
- Free Space
- Other Data
- Buffer
- Frame Pointer
- Return Address
- Caller Data
Beyond Randomization

✦ most programs have deterministic behavior even when they operate outside of their specification; this is exploited by attackers
✦ randomization makes it harder for an attacker to guess “out of specification” behavior, but randomization parameters can still be discovered
✦ running several different randomized versions in parallel sets the threshold much higher: each version must be individually subverted without collateral damage affecting the other versions
Technology Context

- multi-core processors are rapidly becoming the norm, even in laptop computers
- but: most day-to-day computer programs are inherently sequential and cannot (yet) be parallelized to make full use of this hardware
- use the extra processing cores for enhancing security, rather than performance
The Hardware Is Here!

- Intel has already announced an 80-core processor
- Expect number of cores to double every 18 months from now on
- 1,000 cores within a decade
Checkpointing

- different program versions running in lockstep are expected to perform semantically equivalent operations at the same time
- possible checkpointing granularities:
  - system calls: all versions must be seeing same call with same parameters
  - note that a program can cause harm only through system calls
  - instruction-level: must be graduating instructions with same opcode across all participating cores
Hardware Support

Core 1
- fetch
- decode
- ...
- execute
- ...
- retire

Core 2
- fetch
- decode
- ...
- execute
- ...
- retire
Hardware Support

Core 1
- fetch
- decode
- execute
- retire

Core 2
- fetch
- decode
- execute
- retire

LD -8(R6), R2
LD -12(R6), R3
ADD R2, R3
ST R3, -20(R6)
MULI #10, R3
ST R3, -24(R6)

LD 12(R2), R1
LD 18(R2), R5
ADD R1, R5
ST R5, 32(R2)
MULI #10, R5
ST R5, 40(R2)
Hardware Support

Core 1

- LD -8(R6), R2
- LD -12(R6), R3
- ADD R2, R3
- ST R3, -20(R6)
- MULI #10, R3
- ST R3, -24(R6)

Core 2

- LD 12(R2), R1
- LD 18(R2), R5
- ADD R1, R5
- ST R5, 32(R2)
- MULI #10, R5
- ST R5, 40(R2)

= ?

opcode = ? opcode
Hardware Support

LD -8(R6), R2
LD -12(R6), R3
ADD R2, R3
ST R3, -20(R6)
MULI #10, R3
ST R3, -24(R6)

Core 1

fetch
decode
...
execute
...
retire

Core 2

fetch
decode
...
execute
...
retire

LD 12(R2), R1
LD 18(R2), R5
ADD R1, R5
ST R5, 32(R2)
MULI #10, R5
ST R5, 40(R2)

LD LD
LD LD
ADD ADD
ST ST
MULI MULI
ST ST
Hardware Support

Core 1
- fetch
- decode
- execute
- retire

Core 2
- fetch
- decode
- execute
- retire

LD -8(R6), R2
LD -12(R6), R3
ADD R2, R3
ST R3, -20(R6)
MULI #10, R3
ST R3, -24(R6)

LD 12(R2), R1
LD 18(R2), R5
ADD R1, R5
ST R5, 32(R2)
MULI #10, R5
ST R5, 40(R2)

(patent pending)
More Hardware Support

✦ add transactional memory features
  ✦ execute program in “chunks”
  ✦ checkpoint a chunk across cores, commit to memory only when state identical
  ✦ roll back if no agreement
✦ combine with majority voting when more than 2 cores are available (“RAID for computations”)
  ✦ roll back corrupted versions
  ✦ synthesize correct state from surviving versions
✦ (patent pending)
Long-Term Benefit

✦ may have discovered the true “killer app” for multi-core processors that would otherwise likely be sitting idle
✦ permanently eradicate several categories of threats once and for all
✦ shift viewpoint from finding program errors to eliminating their adverse effects
   ✦ but in the process, discover such errors, enabling their correction
✦ might become a “fuzzing” software testing tool
An Even Bolder Approach

- "create a unique version of every program for every person in the universe"
Overall Idea In One Slide

Software Developer creates Software

Software delivers to Diversity Engine within App Store

Diversity Engine within App Store creates Variants

App Store

Subsequent downloaders receive functionally identical but internally different versions of the same software

No change in process for developers

All the magic happens “in the cloud”

No change in process for users
Solves Patch Exposure Problem

critical factor: time interval between when a patch is available and when it is applied

the highest-value users often turn out to be the most vulnerable (international business travellers who cannot download patches while roaming, military deployed in areas with low connectivity)
Safe Patching

- releasing a patch reveals no information to attackers
- Option 1: errors are corrected via replacement software

replacement software

original software

vulnerability cannot be extracted simply by comparing
Safe Patching

- releasing a patch reveals no information to attackers
- Option 2: AppStore creates custom patch for each variant

(custom patch for this version only)

(custom patch is worthless unless you have the exact variant it relates to)
What’s The Cost?

- run-time cost
  - current unicompilers focus on finding the “best” of several alternative code paths, using heuristics
  - a multicompiler enumerates all the alternative paths that implement the same semantics and then gives a different alternative to each successive end-user
  - the difference between the “best” path and a semantically equivalent alternative path chosen for the sake of implementation diversity represents a potential performance loss
    - there are often many alternative paths that all have the “best” performance
    - hardware evolution keeps diminishing the performance differential between “best” path and “sub-optimal” paths

- up-front cost
  - cloud computing resources are cheap: $131 to rent a capable server/storage/bandwidth for one month on EC2 = $0.18 per hour
  - $0.09 to create a unique version of Firefox (30Mloc)
Software Updates & Patches

* interesting research problem of how to update diversified software efficiently
  * needs to be client-driven, so that no “outsider” can know which specific version is running on the client
  * possible solution with a random seed that drives the diversification detailed on next slides
* possibly combine with a “tiling” mechanism that reduces update costs further
**Possible Update Mechanism**

*Initial App Generation*

- v1.0
- v1.1
- v1.3

Software evolves over time

**Diversification Engine**

- Random seeds drive diversification
- Different variants of same software version

Client receives three pieces of information:
- Original version number,
- Random seed, and
- Diversified binary code
Possible Update Mechanism

Initial App Generation

software evolves over time

random seeds drive diversification

different variants of same software version

Client Computers

Diversification Engine

client requests update stating its current version number and random seed

Diversification Engine

client receives three pieces of information:
- original version number,
- random seed, and
- diversified binary code

computed patch v1.1 → v1.3 for client using random seed α

old version on client

newest version

variants are generated using client's random seed

On-Demand Patch Generation
The Multicompiler

✦ at the core, this is a new randomization technique
✦ if a program has an error, the error doesn’t go away, but its effects become unpredictable to the attacker
✦ previous randomization techniques have changed the starting addresses of the stack and heap, but alignment and addressing constraints limit effectiveness (still below brute force threshold, relative distances stay the same)
✦ because we generate a whole new program from scratch for each target, we can apply any transformation available to a compiler
✦ code refactoring on a large scale
Advanced Persistent Threat

✦ targeted for a specific task, going after high-value assets
✦ sophisticated, possibly one-of-a-kind attacks are viable
✦ low and slow
  ✦ infiltrate quietly
  ✦ if designed for extremely long infiltration period, may even contain redundant attack vectors for each stage of attack (Stuxnet)
✦ sophisticated (human) control rather than a mindless automated piece of code
✦ operators with a specific objective are skilled, motivated, organized, and well funded
Advanced Persistent Threat

- the ultimate target of the attack is often completely unrelated to the attacker’s entry point into the organization
  - “low and slow” type of attack might first establish a bridgehead and then incrementally scout out internal networks and credentials; escalate privileges over an extended period of time
  - “custom botnet”
- initial breach (“jailbreak”) typically exploits a vulnerability in software already installed on the host
  - browser component attacks, esp. JavaScript and Flash
  - booby-trapped documents
  - back-doored physical media (USB keys in parking lot…)
  - “rogue-ware” with additional bonus functionality
our goal is not to defend specific applications or services, but to raise the cost for attackers in general

as a result, we raise the cost of jailbreaking a system

so instead of merely providing a direct defense for a specific application or service, our technology also provides an indirect defense against bridgehead-establishing attacks

two indirect mechanisms

- first order: randomize the **immediate** effect of errors
- second order: randomize installed code base on the device, thwarting **indirect** attacks (return-oriented code)
Jailbreak Attack (simplified example)

code with vulnerability
Read(buffer)
RETURN
Jailbreak Attack (simplified example)

Stack

code with vulnerability

Read(buffer)
RETURN
Jailbreak Attack (simplified)

EAX = "whitehouse"
JSR open_connection
...

Read(buffer)
RETURN

code with vulnerability

Stack

100000
100004
100008
10000C
100010
100014
100018
10001C
100020
100024
100028
10002C
100030
100034
100038
Jailbreak Attack (simplified example)

EAX = “whitehouse”
JSR open_connection
...

Stack

100000
100004
100008
10000C
100010
100014
100018
10001C
100020
100024
100028
10002C
100030
100034
100038
10003C
100040
100044
100048
10004C

Read(buffer)
RETURN

Code with vulnerability
Jailbreak Attack (simplified example)

```plaintext
EAX = "whitehouse"
JSR open_connection
...

100000
100004
100008
10000C
100010
100014
100018
10001C
100020
100024
100028
10002C
100030
100034
100038
10003C
100040
100044
100048
10004C
100050
100054
100058
10005C
```

Stack

---

code with vulnerability
Read(buffer)
RETURN

---
Existing Defenses

- non-executable memory protections such as Microsoft's Data Execution Prevention (DEP), CPU-supported non-executable memory (NX/XD), and mandatory code-signing such as on iPhone OS
- attacker cannot inject new code
- as it turns out, this does not stop the modern attacker, who can devise an attack that *indirectly* uses perfectly legitimate code already present on the target computer
  - “return oriented” attack (aka “borrowed code”)
- it turns out to be extremely difficult to defend against this indirect type of attack using conventional means
  - “Return-Less Kernel” and “Gadget-Free Binaries” approaches defeated within months, see Schacham’s “Return-Oriented Programming Without Returns”
ROP Attack (simplified)

code with vulnerability
Read(buffer)
RETURN

in Display_ColorText
...
555A00  EAX = “black”
555A04  else
555A08  EAX = “white”
555A0C  RETURN

in HelpTexts_IOSpecific
...
212500  EBX = “keyboard”
212504  else
212508  EBX = “mouse”
21250C  RETURN

in Audio_LowerVolume
...
ABCD00  SUBI #5000, @EBX
ABCD04  RETURN

in PrintManager_Prepare
...
919100  JSR open_connection
919104  RETURN

in FileSystem_QualifiedDirectoryName
...
777700  JSR strg_concat_EAX_EBX
777704  RETURN
ROP Attack (simplified)

Stack

buffer

EAX = "black"
EAX = "white"
RETURN

Number 202025

In Display_ColorText

EAX = "black"
EAX = "white"
RETURN

In HelpTexts_IOSpecific

EBX = "keyboard"
EBX = "mouse"
RETURN

In Audio_LowerVolume

SUBI #5000, @EBX
RETURN

In PrintManager_Prepare

JSR open_connection
RETURN

In FileSystem_QualifiedDirectoryName

JSR strg_concat_EAX_EBX
RETURN

Stac

code with vulnerability

Read(buffer)
RETURN

Legitimate code on target

EB

EA
ROP Attack (simplified)

**Stack**

```
0x100000: 555A08
0x100004: 212508
0x100008: ABCD00
0x10000C: 777700
0x100010: 919100
```

**Code with vulnerability**

```
555A00    EAX = "black"
555A04    else
555A08    EAX = "white"
555A0C    RETURN
```

**Legitimate code on target**

```
212500    EBX = "keyboard"
212504    else
212508    EBX = "mouse"
21250C    RETURN
```

**In Display_ColorText**

```
... 555A00    EAX = "black"
... 555A04    else
... 555A08    EAX = "white"
... 555A0C    RETURN
```

**In HelpTexts_IOSpecific**

```
... 212500    EBX = "keyboard"
... 212504    else
... 212508    EBX = "mouse"
... 21250C    RETURN
```

**In Audio_LowerVolume**

```
... ABCD00    SUBI #5000, @EBX
ABC0D04    RETURN
```

**In PrintManager_Prepare**

```
... 919100    JSR open_connection
... 919104    RETURN
```

**In FileSystem_QualifiedDirectoryName**

```
... 777700    JSR strg_concat_EAX_EBX
... 777704    RETURN
```
ROP Attack (simplified)

Stack

- 555A08
- 212508
- ABCD00
- 777700
- 919100

EAX = "black"

- 555A04
- 555A08
- 212504
- 212508

EAX = "white"

RETURN

EAX = "white"

RETURN

EBX = "keyboard"

- 212500
- 212504
- ABCD00

EBX = "mouse"

- 212508
- 21250C

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508
- 21250C

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN

EBX = "keyboard"

- 212500
- 212504
- 212508

RETURN
ROP Attack (simplified example)

Code with vulnerability

```
Read(buffer)
RETURN
```

in `Display_ColorText`

```
... 555A00 EAX = "black"
555A04 else
555A08 EAX = "white"
555A0C RETURN
```

in `HelpTexts_IOSpecific`

```
... 212500 EBX = "keyboard"
212504 else
212508 EBX = "mouse"
21250C RETURN
```

in `Audio_LowerVolume`

```
... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN
```

in `PrintManager_Prepare`

```
... 919100 JSR open_connection
919104 RETURN
```

in `FileSystem_QualifiedDirectoryName`

```
... 777700 JSR strg_concat_EAX_EBX
777704 RETURN
```
### ROP Attack (simplified)

<table>
<thead>
<tr>
<th>Stack (EA)</th>
<th>Code with vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td></td>
</tr>
<tr>
<td>100004</td>
<td></td>
</tr>
<tr>
<td>100008</td>
<td></td>
</tr>
<tr>
<td>100010</td>
<td></td>
</tr>
<tr>
<td>100014</td>
<td></td>
</tr>
<tr>
<td>100018</td>
<td></td>
</tr>
<tr>
<td>100020</td>
<td></td>
</tr>
<tr>
<td>100024</td>
<td></td>
</tr>
<tr>
<td>10002C</td>
<td></td>
</tr>
<tr>
<td>100030</td>
<td></td>
</tr>
<tr>
<td>100034</td>
<td></td>
</tr>
<tr>
<td>100038</td>
<td></td>
</tr>
</tbody>
</table>

**in Display_ColorText**

```
... 555A00 EAX = "black"
555A04 else
555A08 EAX = "white"
555A0C RETURN
```

**in HelpTexts_IOSpecific**

```
... 212500 EBX = "keyboard"
212504 else
212508 EBX = "mouse"
21250C RETURN
```

**in Audio_LowerVolume**

```
... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN
```

**in PrintManager_Prepare**

```
... 919100 JSR open_connection
919104 RETURN
```

**in FileSystem_QualifiedDirectoryName**

```
... 777700 JSR strg_concat_EAX_EBX
777704 RETURN
```
ROP Attack (simplified)

- **Display_ColorText**
  - Code with vulnerability
  - Read(buffer)
  - RETURN

- **HelpTexts_IOSpecific**
  - 555A00 EAX = “black”
  - 555A04 else
  - 555A08 EAX = “white”
  - 555A0C RETURN

- **Audio_LowerVolume**
  - ABCD00 SUBI #5000, @EBX
  - ABCD04 RETURN

- **PrintManager_Prepare**
  - 919100 JSR open_connection
  - 919104 RETURN

- **FileSystem_QualifiedDirectoryName**
  - 777700 JSR strg_concat_EAX_EBX
  - 777704 RETURN
ROP Attack (simplified)

- **Display_ColorText**
  - Code with vulnerability
  - `Read(buffer)`
  - `RETURN`

- **HelpTexts_IOSpecific**
  - `555A00 EAX = “black”`
  - `212508 EAX = “white”`
  - `RETURN`

- **Audio_LowerVolume**
  - `ABCD00 SUBI #5000, @EBX`
  - `ABCD04 RETURN`

- **FileSystem_QualifiedDirectoryName**
  - `777700 JSR strg_concat_EAX_EBX`
  - `777704 RETURN`

- **PrintManager_Prepare**
  - `919100 JSR open_connection`
  - `919104 RETURN`

- **FileSystem_QualifiedDirectoryName**
  - `777700 JSR strg_concat_EAX_EBX`
  - `777704 RETURN`
ROP Attack (simplified)

Stack

<table>
<thead>
<tr>
<th>EA</th>
<th>EB</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>mouse</td>
</tr>
</tbody>
</table>

Code with vulnerability

in Display_ColorText

... 555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific

... 212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume

... ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in PrintManager_Prepare

... 919100 JSR open_connection
919104 RETURN

in FileSystem_QualifiedDirectoryName

... 777700 JSR strg_concat_EAX_EBX
777704 RETURN
ROP Attack (simplified)

- **Display_ColorText**
  - Code with vulnerability
  - Read(buffer)
  - RETURN

- **HelpTexts_IOSpecific**
  - 555A00 EAX = “black”
  - 555A04 else
  - 555A08 EAX = “white”
  - 555A0C RETURN

- **Audio_LowerVolume**
  - 212500 EBX = “keyboard”
  - 212504 else
  - 212508 EBX = “mouse”
  - 21250C RETURN

- **PrintManager_Prepare**
  - 919100 JSR open_connection
  - 919104 RETURN

- **FileSystem_QualifiedDirectoryName**
  - 777700 JSR strg_concat_EAX_EBX
  - 777704 RETURN

- **Stack**
  - EA
    - white
  - EB
    - house
ROP Attack (simplified)

code with vulnerability

Read(buffer)
RETURN

in Display_ColorText

... 555A00  EAX = “black”
     555A04  else
     555A08  EAX = “white”
     555A0C  RETURN

in HelpTexts_IOSpecific

... 212500  EBX = “keyboard”
     212504  else
     212508  EBX = “mouse”
     21250C  RETURN

in Audio_LowerVolume

... ABCD00  SUBI #5000, @EBX
     ABCD04  RETURN

in PrintManager_Prepare

... 919100  JSR open_connection
     919104  RETURN

in FileSystem_QualifiedDirectoryName

... 777700  JSR strg_concat_EAX_EBX
     777704  RETURN
ROP Attack (simplified)

code with vulnerability

in Display_ColorText

... 555A00  EAX = "black"
555A04  else
555A08  EAX = "white"
555A0C  RETURN

in HelpTexts_IOSpecific

... 212500  EBX = "keyboard"
212504  else
212508  EBX = "mouse"
21250C  RETURN

in Audio_LowerVolume

... ABCD00  SUBI #5000, @EBX
ABCD04  RETURN

in PrintManager_Prepare

... 919100  JSR open_connection
919104  RETURN

in FileSystem_QualifiedDirectoryName

... 777700  JSR strg_concat_EAX_EBX
777704  RETURN
ROP Attack (simplified)

code with vulnerability

in Display_ColorText

... 555A00 EAX = “black”
      555A04 else
      555A08 EAX = “white”
      555A0C RETURN

in HelpTexts_IOSpecific

... 212500 EBX = “keyboard”
      212504 else
      212508 EBX = “mouse”
      21250C RETURN

in Audio_LowerVolume

... ABCD00 SUBI #5000, @EBX
      ABCD04 RETURN

in PrintManager_Prepare

... 919100 JSR open_connection
      919104 RETURN

in FileSystem_QualifiedDirectoryName

... 777700 JSR strg_concat_EAX_EBX
      777704 RETURN
ROP Attack (simplified)

Stack

code with vulnerability

Read(buffer)
RETURN

in Display_ColorText
...
555A00 EAX = “black”
555A04 else
555A08 EAX = “white”
555A0C RETURN

in HelpTexts_IOSpecific
...
212500 EBX = “keyboard”
212504 else
212508 EBX = “mouse”
21250C RETURN

in Audio_LowerVolume
...
ABCD00 SUBI #5000, @EBX
ABCD04 RETURN

in FileSystem_QualifiedDirectoryName
...
777700 JSR strg_concat_EAX_EBX
777704 RETURN

in PrintManager_Prepare
...
919100 JSR open_connection
919104 RETURN

whitehouse

house

stack

EA

EB
Return-Oriented Programming

- any reasonably large code base includes a Turing-complete set of “Gadgets” that can be used to splice together arbitrary programs (including branching and looping)
  - compounded on x86 by variable length instructions
  - massive-scale compiler-generated code diversity prevents the attacker from compiling a “Gadget Dictionary” that is identical across many machines
- in conjunction with the non-executable stack hardware feature, this is a second-order defense
  - even if the attacker were able to overwrite the stack, a return oriented attack would then require detailed knowledge (down to individual bytes!) of the installed binaries on the target
  - we can prevent such a knowledge from existing
Summary & Conclusion

✦ creating a unique binary of every program for every user has become feasible only very recently
✦ if this idea is adopted widely, it will change many of the assumptions and models underlying current threats to deployed software
✦ limits the effect of large-scale attacks
✦ makes it much more difficult to perform directed attacks against specific targets, because attackers can no longer examine their own copies of software to find exploitable vulnerabilities that will also apply to a specific target
✦ makes it much more difficult to generate attack vectors by reverse-engineering of software updates and patches
✦ especially in the mobile space, where some users (e.g., roaming travellers) may not be able to download updates or may not have the required bandwidth to do so (e.g., military in remote places)