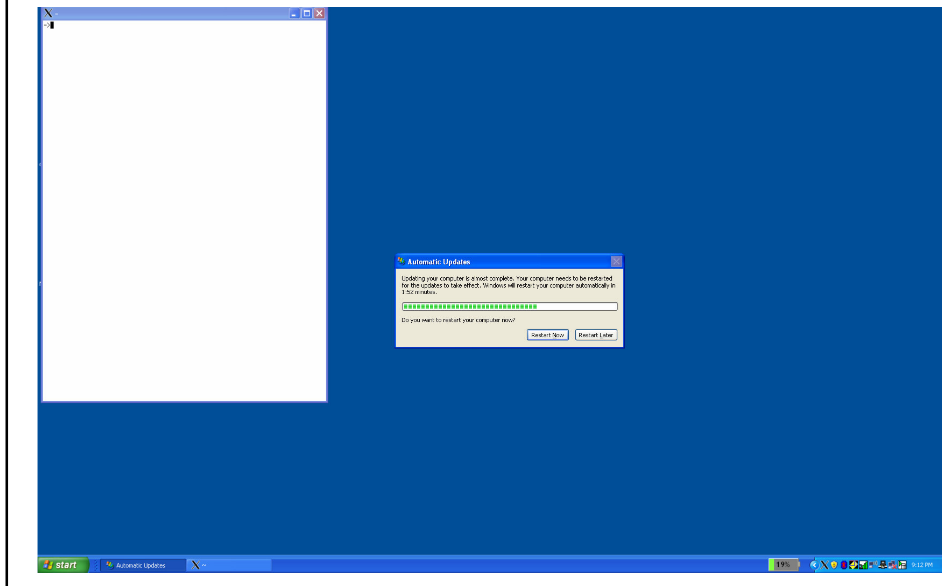
Dynamic Software Updating

Iulian Neamtiu
(sitting in for Dr. Hicks)

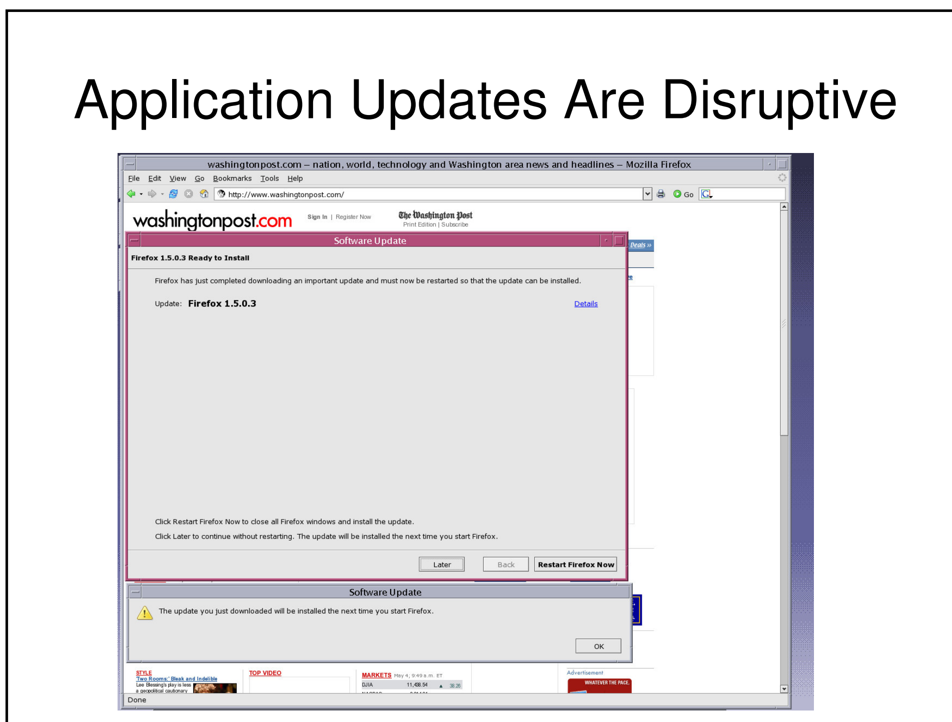
Outline

- The software updating problem
- What is *Dynamic Software Updating* ? Why is DSU useful ?
- How can we build a DSU system ? What are the challenges ?
- Ginseng – a DSU system for userspace apps, developed at UMD
- DSU for operating systems

OS Updates Are Disruptive

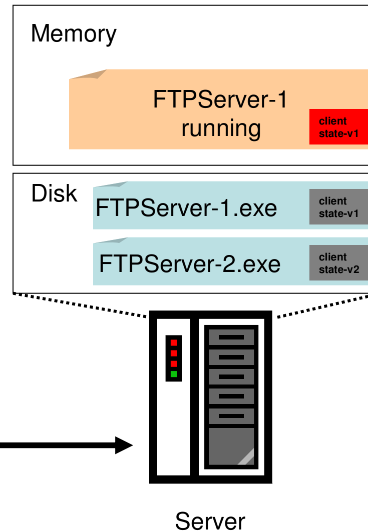


Application Updates Are Disruptive



Case Study: Updating an FTP Server

Goal: update the FTP server without disrupting service to client



Client

Server

Dynamic Software Updating

(DSU)

- Update a running program with new code and data
  - Preserves state and processing
- Critical for non-stop systems
  - Air-traffic control, financial transaction processing, network components, …
- Convenient for other systems
  - No need to reboot your OS after a patch!

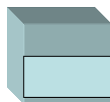
Advantages

- General-purpose
  - Preserves arbitrary application state between updates (requires no state externalization - DB, file system)
- No service disruption
- No redundant hardware
  - Application is updated in place

Developing for DSU

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
```

- Start: existing source

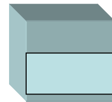


Running system

Developing for DSU

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

- Start: existing source
- Modify program as needed



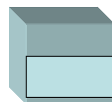
Running system

Developing for DSU

accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c

- Start: existing source
- Modify program as needed
- Compile it and test it

New version

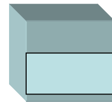


Running system

Developing for DSU

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

- Start: existing source
- Modify program as needed
- Compile it and test it
- Develop dynamic patches

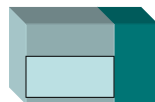


Running system

Developing for DSU

```
accept.c
cold.c
common.c
data.c
file.c
libhttpd.c
loop.c
main.c
maint.c
match.c
name.c
nameconvert.c
readreq.c
tdate_parse.c
timer.c
dir_slave.c
```

- Start: existing source
- Modify program as needed
- Compile it and test it
- Develop dynamic patches
- Apply patches to running system



Running system

How can we support/implement dynamic updating ?

DSU for a Hypothetical FTP Server

```
struct clientState {
  sock_t socket;
};


void sendFile(sock_t s){
  buffer b = readFile();
  send(s,b);
}

void server() {
  struct clientState *c = accept();

  while(1) {
    switch (getRequest(c)) {
      case GET_FILE:
        sendFile(c->socket);
      case …:
    }
    update(); // magic
  }
}
```

Server code, version 1

```
struct clientState {
  stat_s statistics;
  sock_t socket;
};

void sendFile(sock_t s){
  buffer b = readFile();
  if (! errorState(b))
    send(s,b);
}

. . .
```

Server code, version 2

What do we want to achieve ?
What are the challenges ?

Challenges

- Flexibility
  - The changes I make to the source code I want to make on-line
- Safety
  - My program shouldn't fail when I do it!
- Efficiency
  - Ease of Use
    - No need for unusual app restructuring.
    - Minimize per-update programmer work.
  - Performance

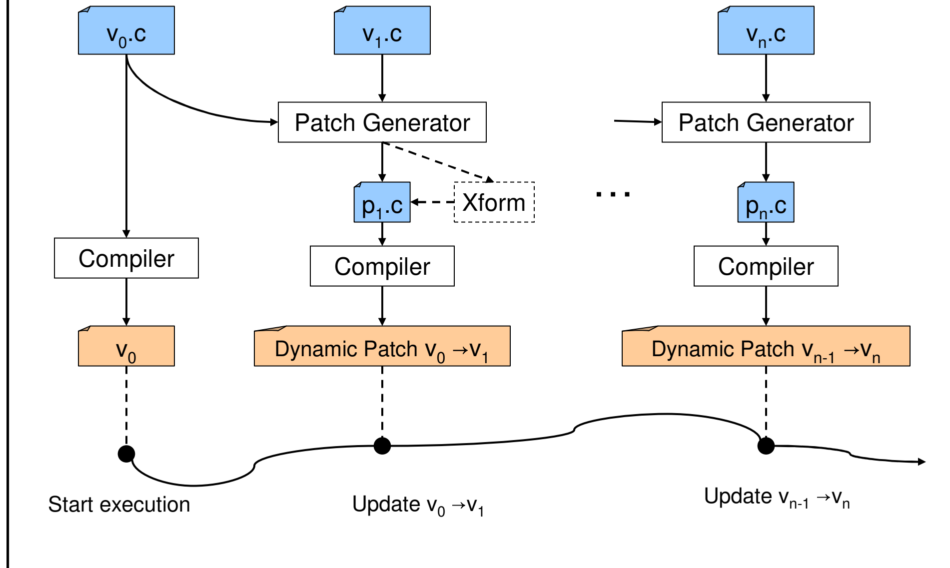
Can We Use Existing Mechanisms for DSU ?

- Kernel modules
  - Kernel extensions
  - Load/unload at runtime
- Dynamically linked libraries (.dll/.so)
  - Implement common services (e.g., `printf`, GUI)
  - Linked into process address space at loadtime

Ginseng

- System for building DSU software
  www.cs.umd.edu/projects/dsu
- Approach is practical
  - Broad range of updates to C programs
  - Minimal changes to applications
  - Updates easy to write, mostly automatically
  - Strong safety guarantees
- Validated on realistic applications
  - OpenSSH, VsFTPd, Zebra
    - Three years worth of updates
  - Low performance overhead

Ginseng Architecture



Compilation Techniques

- Type wrapping: compiler makes accesses to values of named type to be through special functions
  - Must run type transformers on the accessed value if its type has been updated
- Function indirection: compiler adds an indirection between each caller and called function
  - Each function call will always be to the most recent version

Example

```
struct T { int x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

Example: Type wrapping

```
struct __T0 { int x; int y; };

struct T {
  unsigned int version;
  union { struct __T0 data;
          char slop[…]; } u;
};

struct __T0* __con_T(struct T* abs){
  __DSU_transform(abs);
  return &abs->u.data;
}
```

Example: Accessing Types

```
void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

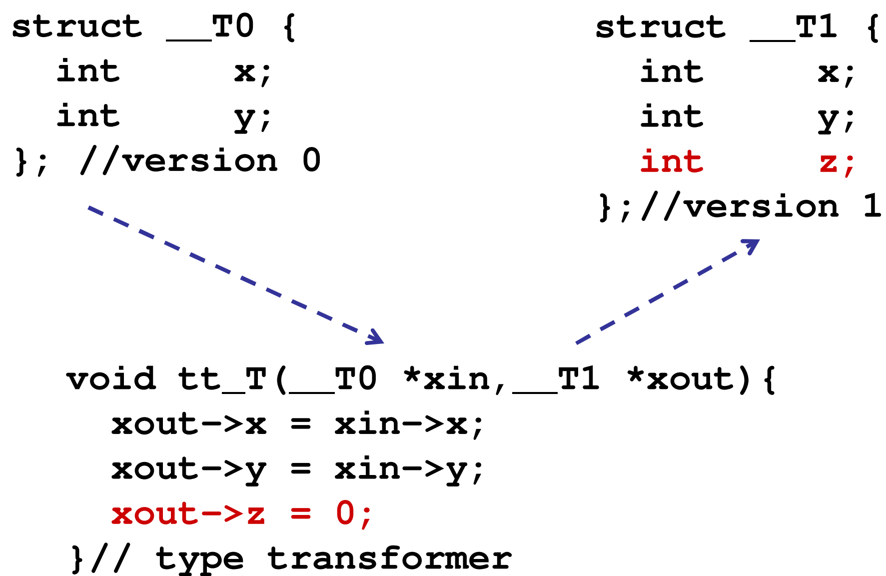
Example: Accessing Types

```
void call() {
  struct T t =
    { 0, {.data={1,2}} };
  foo(&t.x);
}
```

Example: Accessing Types

```
void call() {
  struct T t =
    { 0, {.data={1,2}} };
  foo(&(__con_T(&t))->x);
}
```

Type Transformers



Example: Function Indirection

```
void foo(int* x) { *x = 1; }

void call() {
  struct T t = …;
  foo(&(con_T(&t))->x);
}
```

Example: Function Indirection

```
void foo_v0(int* x) { *x = 1; }

void call_v0() {
  struct T t = …;
  foo(&(con_T(&t))->x);
}
```

Example: Function Indirection

```
void* foo_fptr = foo_v0;

void foo_v0(int* x) { *x = 1; }

void call_v0() {
  struct T t = …;
  foo_fptr(&(con_T(&t))->x);
}
```

Example: Function Indirection

```
void* foo_fptr = foo_v1;
void foo_v1(int* x) { . . . }
void foo_v0(int* x) { . . . }

void call_v0() {
  struct T t = …;
  foo_fptr(&(con_T(&t))->x);
}
```

Problem: Bad Timing

- Updating **t** when some existing code still expects the old representation could lead to a type error.
  - This situation is timing dependent.

***Question**: when during a program's execution is it safe to update the representation of a type **t**?*

Example: version 1

```
struct T { int x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  struct T t = {1,2};
  foo(&t.x);
}
```

Example: version 2

```
struct T { int *x; int y; };

void foo(int* x) { *x = 1; }

void call() {
  int z = 1;
  struct T t = {&z,2};
  foo(t.x);
}
```

Starting execution

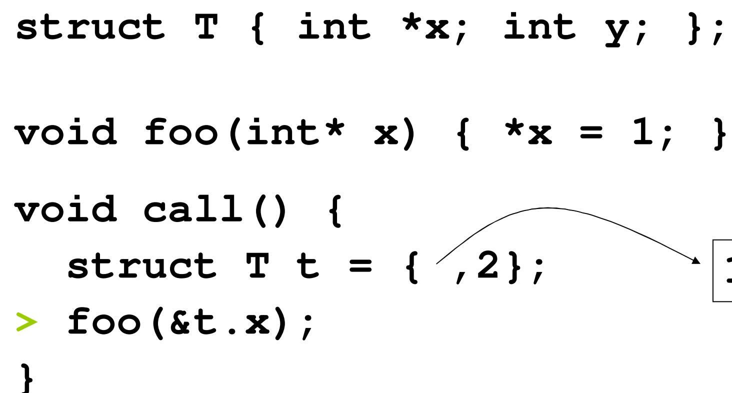
```
struct T { int x; int y; };
  struct T { int *x; int y; };
void foo(int* x) { *x = 1; }

void call() {
> struct T t = {1,2};
  foo(&t.x);
}
```

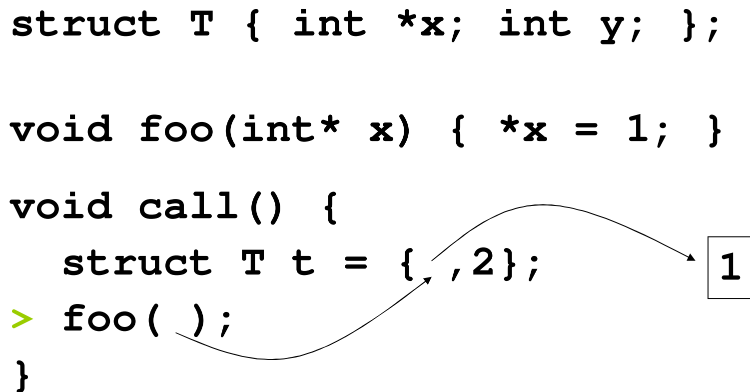
Attempting update now

```
struct T { int x; int y; };
  struct T { int *x; int y; };
void foo(int* x) { *x = 1; }
void call() {
  struct T t = {1,2};
> foo(&t.x);
}
```

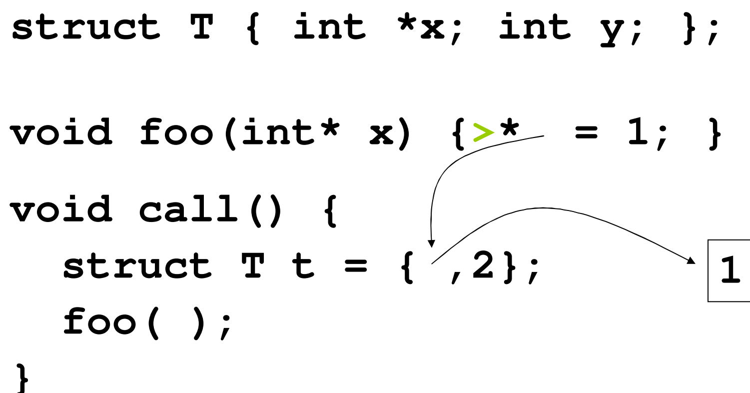
Run type transformer



Taking the address



Call foo()



Doing the assignment: error!

```
struct T { int *x; int y; };

void foo(int* x) {>*  = 1; }

void call() {
  struct T t = {1,2};
  foo( );
}
foo: expected int*, got int **
```

The problem

- The new program was type correct
- But the old version of **`call`** was active at the time of the update, and expected the *old* **`struct T`** rep
  - It uses it *concretely*
- A similar situation occurs when changing the types of functions or global variables

Our Approach: Safety Analysis

- *con_T* functions identify when a type is used concretely
- Dynamically prevent updates that could lead to old code concretely using a transformed value
  - Calculate dependencies at compile-time
  - Apply same idea to function calls, global variable references

Example revisited

```
void foo(int* x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3
  foo(&t.x);
4
}
```

Example revisited

```
void foo(int* x) {
1
  *x = 1;
2
}

void call() {
  struct T t = {1,2};
3 {T,foo}
  foo(&t.x);
4
}
```

Dependence on type of T and foo

Example revisited

```
void foo(int* x) {
1 {T}
  *x = 1;
2 {T}
}

void call() {
  struct T t = {1,2};
3 {T,foo}
  foo(&t.x);
4 {}
}
```

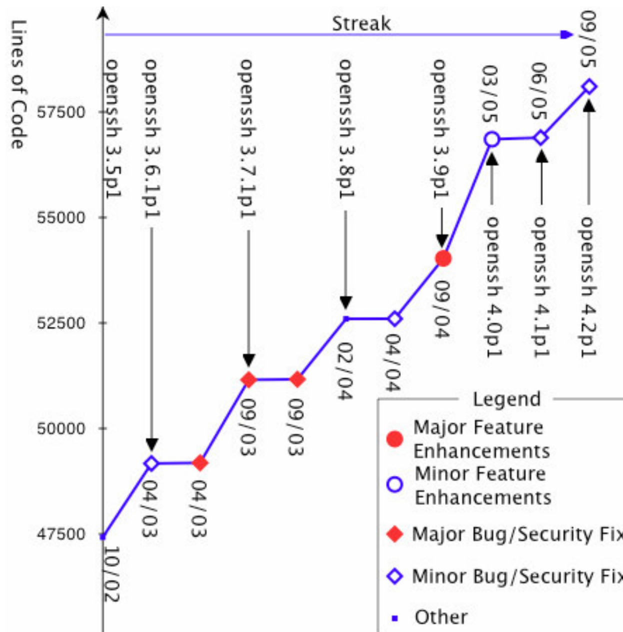
Dependence on type of T

Application Experience

Last 3 years' worth of updates for 3 apps:

- Very secure FTP daemon
  - 10K-17K LOC, 13 releases
    - 6 type changes, 308 function body changes, 33 function proto changes
- OpenSSH daemon
  - 47K-58K LOC, 11 releases
    - 19 type changes, 752 function body changes, 85 function proto changes
- Zebra routing daemon
  - 41K-45K LOC, 6 releases
    - 4 type changes, 321 function body changes, 13 function prototype changes

OpenSSH update summary



Writing Updateable Apps

- Choose *update points*
  - Updates only permitted at conclusion of main event loops
    - two in vsftpd, one in sshd, one in zebra
- Make small application changes
  - Adjustments to potentially unsafe idioms; overrode analysis in a few cases (1-2 per app)
  - Relax application invariants to accommodate updateability
  - Total: 40-50 lines total per app for all three years

Writing Dynamic Patches

- Type transformers
  - Typical change was to add new fields; wrote initializers for these
  - Most frequent was loop state
    - Initialize storage for new variables
    - Carry old state forward
- Update-time initialization
  - Routines normally called in **`main`**() called at update-time

Experiments

- Throughput
  - Transfer rate in `vsftpd`, `sshd`: **unaffected**
- Overhead
  - Connection setup+tear in `vsftpd`, `sshd`: 0..32%
  - Route setup/route redistribution in `zebra`: 4..12%
- Memory footprint
  - 0..40%
- Update application time
  - Linear with patch size; less than 5 ms

DSU for Operating Systems

- Motivation
  - Application-friendly (no app restart)
  - Preserve state (network, HW)
  - User-friendly (servers)
- Problems
  - Size, complexity, concurrency, low-level
  - DSU-specific: indirection, wrapping

Selective Indirection

- Performance/flexibility trade-off
- Types
  – Fixed representation/no change expected
    - E.g., page table entry, IP address
  – Non-indirected types updated manually
- Functions
  – Indirection/patch size trade-off

When is it safe to update the OS ?

Linux Kernel Modus Operandi

- Mostly: run user processes
- Exceptions: process inline
- “Short” interrupts: process inline
- “Long” interrupts: top half + bottom half
  - Example: timer
- No bottom halves pending ? **`schedule()`**

Linux Kernel as Event Processor

```
while(1) {
    switch(event):
        interrupt: do_interrupt(); break;

        exception: do_exception(); break;

        bottom_half: do_bottom_half(); break;

        default: schedule();
}
```

Identify “transactions” (one or more events)
Updates to **T** only permitted outside transactions using **`T`**

Performing an OS Update

- Dynamic patch = kernel module
- Update signaling
- Wait for safe update point
- Patch loading
  - – Block interrupts
  - – Load module, redirect
  - – Unblock interrupts

Related Work

- Instrumenting or changing running code
  - – Unsanity Application Enhancer (APE)
    - Used to fix Apple security bugs on-the-fly, rather than wait for monthly patch
  - – Fix-and-continue language environments
    - In .NET, the JVM, Smalltalk, Common Lisp
  - – Dynamic Instrumentation (e.g., DynInst)
- Lack/partial safety analysis, limited flexibility (updates fairly simple)

Conclusions

- Software updates disruptive
- Approaches: checkpointing, DSU, redundant HW, binary rewriting
- DSU can be practical (Ginseng)
  - Can support changes to applications as they occur in practice
  - Process is largely automated
  - Performance overhead negligible for I/O-bound applications
- OSes harder, but not infeasible