Practical Dynamic Software Updating for C

Michael Hicks

University of Maryland, College Park

with

Gareth Stoyle, Gavin Bierman, Peter Sewell, Iulian Neamtiu, and Manuel Oriol

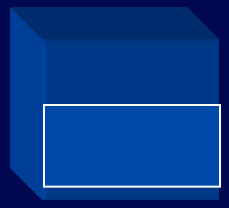
Dynamic Software Updating
(DSU)

- 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!

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

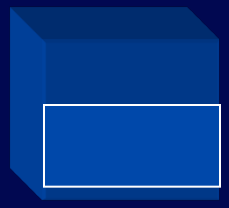


Running system

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

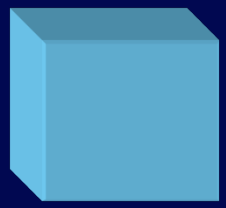


Running system

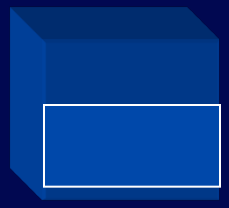
Developing for DSU

```
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

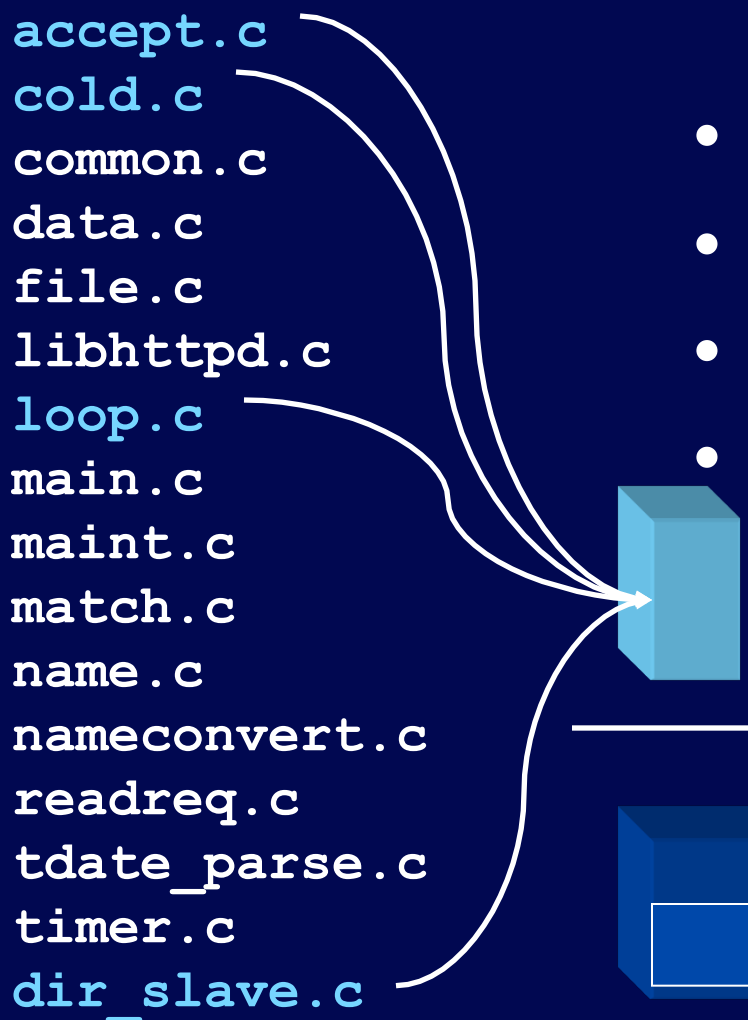


New version



Running system

Developing for DSU



- 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

Advantages

- General-purpose
  - Preserves arbitrary application state between updates
    - Load-balancing approach requires state externalization (e.g., DB, file system)
- No redundant hardware
  - Application is updated in place

The 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!
- Ease of Use
  - No need for unusual app restructuring.
  - Minimize per-update programmer work.

Contributions

- A complete infrastructure for generating and applying dynamic updates to C programs
- Analyses to reason about update safety
  - Considers how a badly-timed update could fail
    - Changes to type representations, signatures
    - Abstraction-violating casts and aliases
- Experimental evaluation for several substantial updates
  - Case studies: 3 years' worth of changes to vsftpd, opensshd, zebra
  - Performance costs

Dynamic Updates: Form

- Replace, add, or delete definitions
  - Functions, globals, and type definitions
  - Updated functions may have different types
- To update a type definition **t**, user provides a *type transformer* function **c**
  - Used by the runtime to convert values of type **t** to the new representation

Compilation Techniques

- Function indirection: compiler adds an indirection between each caller and called function
  - Each function call will always be to the most recent version
- 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

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;
}
```

Alternative: Add Indirection

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

struct T {
  unsigned int version;
  struct __T0 *data;
};

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

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);
}
```

Updating code on the stack

- Dynamic updates take effect at function calls
  - A function call is always to the most recent version
- What about code that is on the stack?
  - Long running loops
  - Code that is returned to

Loop extraction

- Extract out loop body into function
  - Argument is *loop state*: consists of all locals and parameters in the host function
- Loop actions (break, continue, etc.) become return codes handled in host
- Reuses existing updateability mechs.
- Can be used for arbitrary code **S** by changing that code to be
  - while (1) { **S**; break; }

Example: vsftpd

```
main() {
  … init …
  if (tunable_listen)
    standalone_main();
  … handle conn …
}
```

```
standalone_main() {
  … init listen sock l …
  while (1) {
    if (x = acceptconn(l))
      fork and return
      in child
  }
}
```

Example: vsftpd

```
main() {
  … init …
  if (tunable_listen)
    standalone_main();
  while (1) {
    … handle conn …
    break;
  }
}
```

```
standalone_main() {
  … init listen sock l …
  while (1) {
    if (x = acceptconn(l))
      fork and return
      in child
  }
}
```

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

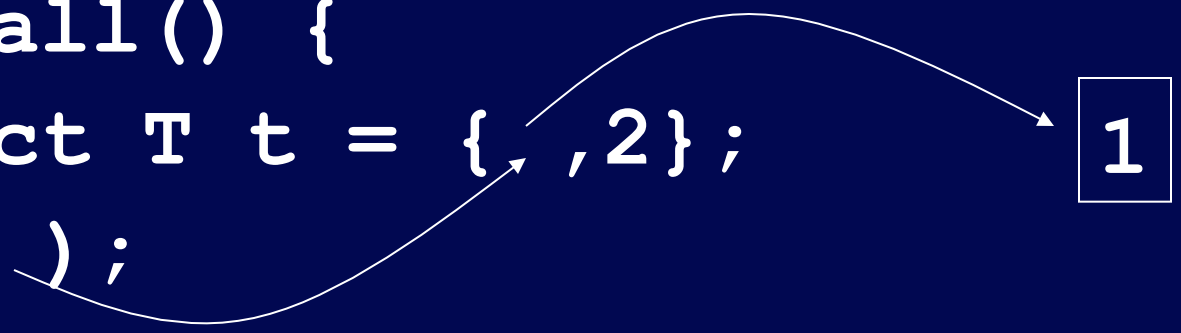
```
struct T { int x; int y; };
  struct T { int *x; int y; };
void foo(int* x) { *x = 1; }

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

Run type transformer

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

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

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

Taking the address

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

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

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



Call foo()

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

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

void call() {
  struct T t = { ,2};
  foo( );
}
```

Doing the assignment: error!

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

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

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

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

Possible Solution #1

- Copy and transform values whose types have changed to the new code, leaving the existing ones as is (Hicks 2001).
- Problems:
  - Old code could operate on stale data, or call old versions of functions
  - Update point must be chosen carefully

Possible Solution #2

- Allow it, but require *backward* type transformers for each updated type T (Duggan 2002) and *stubs* for functions that changed type (Segal 1990)
- Problems:
  - May not be possible to convert a type backwards, particularly since type changes often add information
  - Hard to reason about program behavior
    - Convert forward, back, forward = ?

Possible Solution #3

- Disallow updates to active code (Gilmore 1997, Malabarba 2000, …)
- Problems:
  - Updates less available (loops)

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 {}
  *x = 1;
2 {}
}

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

No type dependencies

Dependence on type of T and foo

Formalism: Proteus

[POPL 2005, TOPLAS 2007]

- Type system of a simple imperative language called Proteus
  - Update points explicit in program text
  - Efficient constraint-based inference
- Well-formed and well-timed updates will not cause the program to *go wrong*

Abstraction-violating Aliases

- Cannot transform a value when there exists an alias into it
  - Reveals its representation indirectly
- An alias into a value of type T should prevent T's update

Example revisited

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

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

Example revisited

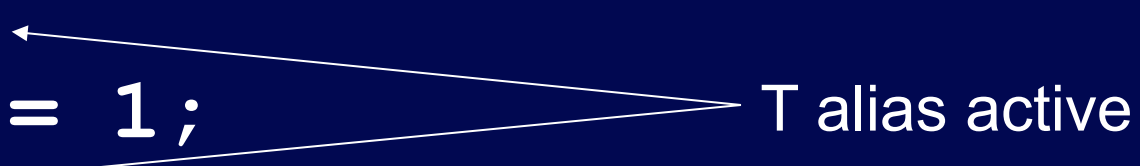
```
void foo(int*{T} x) {
1
  *x = 1;
2
}

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

Parameter is alias into T

Creates alias into T

Example revisited

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

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



Example revisited

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

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

T alias active

No active unsafe aliases

Combining the Analyses

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

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

Implementation

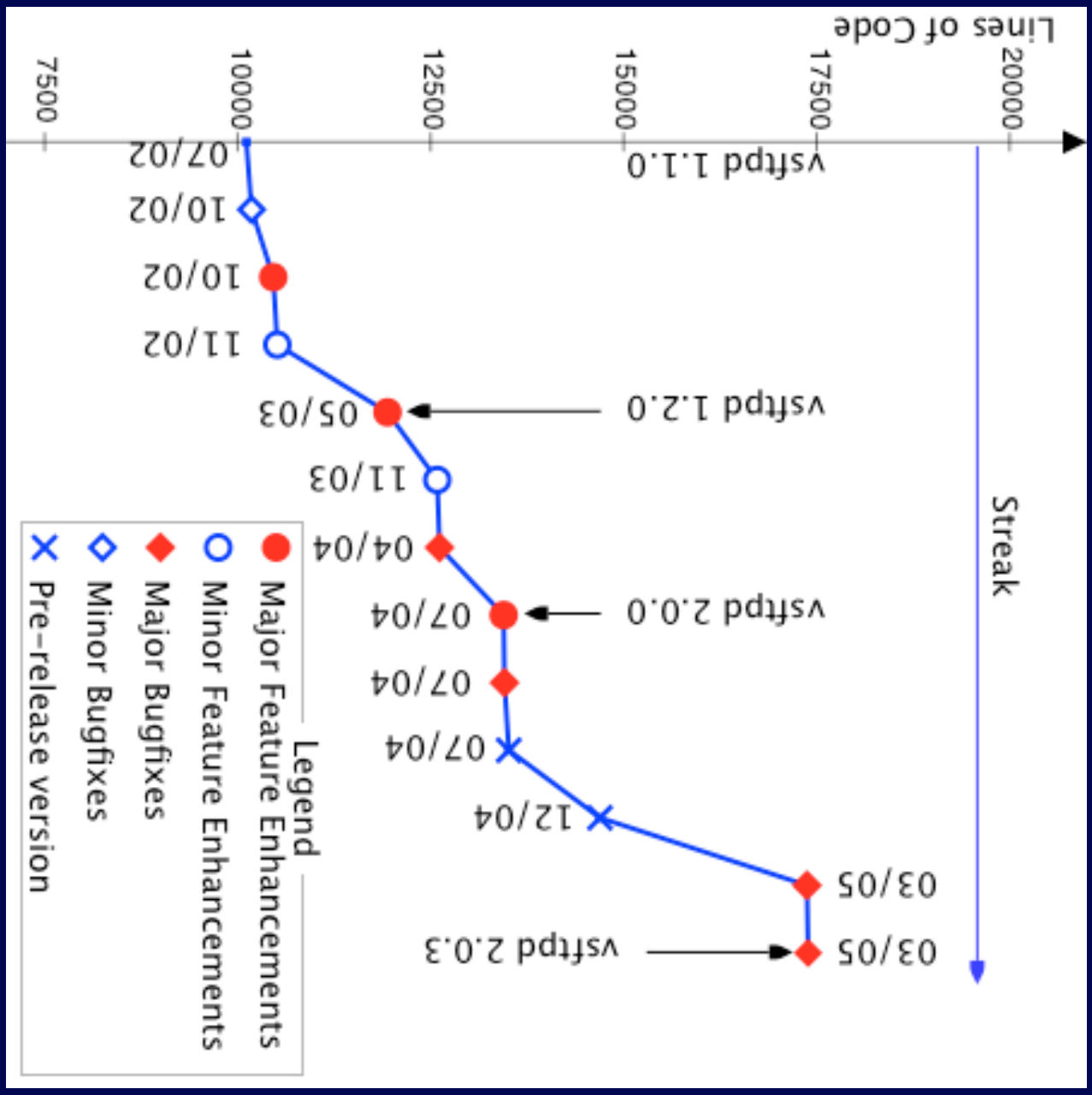
- Compiler
  - implemented using CIL framework
- Safety analysis
  - Constraints solved using BANSHEE
- Patch generation tool
  - Discovers new and changed definitions
  - Constructs default type transformers
- DLOPEN library for loading patches

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

Vsftpd Update Summary



OpenSSH Update Summary

Zebra 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

Why does it work?

- Quiescence
  - Event loops present natural quiescent points to transition versions
  - `assert()` in code helps identify global invariants
- Type safety and abstraction
  - Programs do not use unsafe casts extensively
  - Rarely rely on type having a particular compiled representation
- New state is a function of old state
  - But some features may take effect only with new connections

Performance

- Overhead introduced by updateability
  - Function indirections
  - Type wrapping
    - Effect of con_T functions
    - Effect of extra slop
- Incurred on running time and memory footprint

Experiments

- Measure
  - Request establishment time, transfer rate
  - Memory footprint
  - Patch load times
- Consider **stock**, **updateable**, **updated once**, and **updated** variants
  - The first two are the last version compiled directly; last two the result of performing the last one or all the updates, resp.
- Run on dual Xeon 2 GHz, 1 GB RAM, 100 Mbps Ethernet, Fedora Core 3, kernel v. 2.6.9

Latency and Transfer Rate

Application	Connection time (ms)			
	stock	updateable	upd. once	streak
vsftpd	6.71	6.9	7.04	8.4
sshd	47.62	49.26	49.5	62.89

Application	Transfer rate (MB/s)			
	stock	updateable	upd. once	streak
vsftpd	7.95	7.95	7.97	7.98
sshd	7.85	7.84	7.83	7.84

- Conn: worst-case -25% vsftpd, -32% sshd
- Zebra: worst-case -12% for route redist., route addition/deletion

Memory Footprint

Vsftpd

- Stock: 4.4 MB
- Updated (12 times): 5.3 MB (+21%)

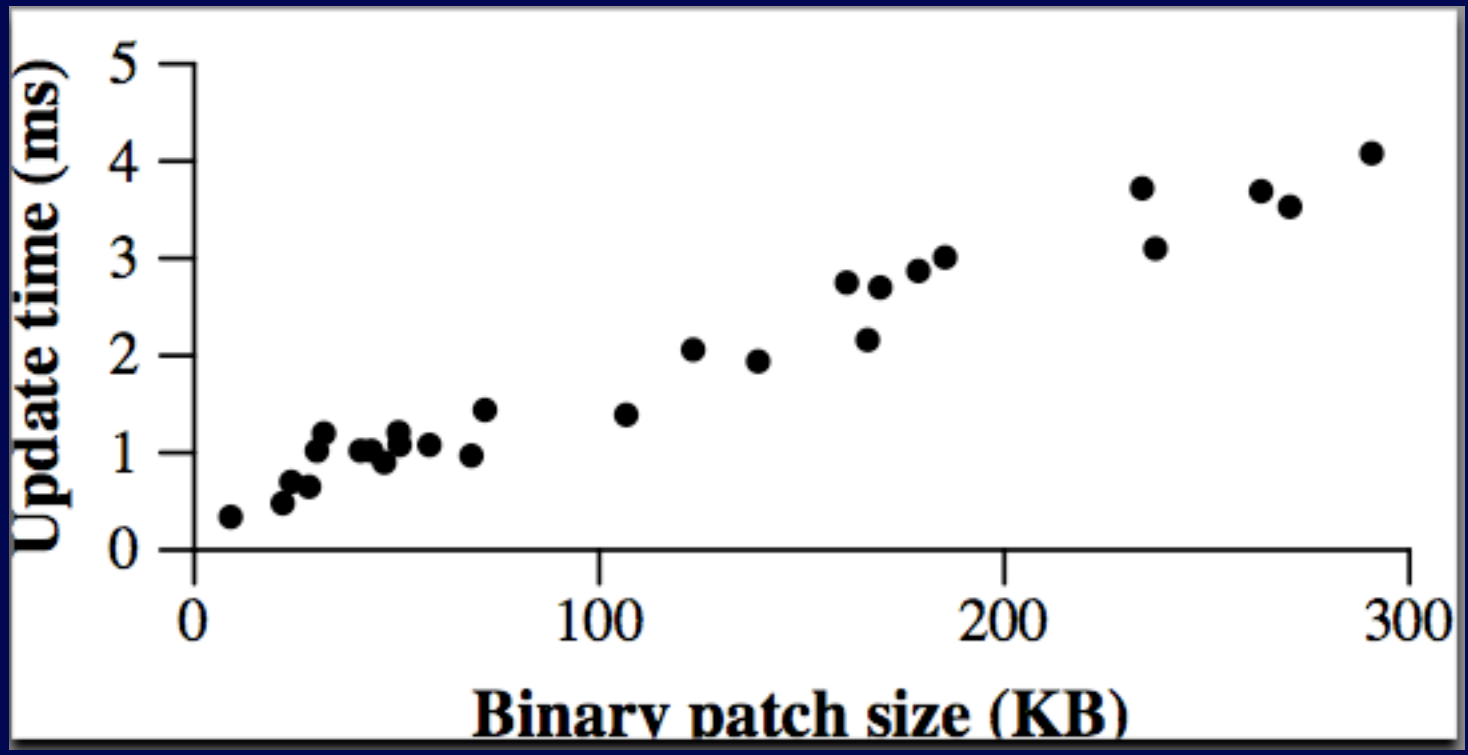
OpenSSH daemon

- Stock: 5.3 MB
- Updated (11 times): 7.4 MB (+40%)

Zebra

- Stock 3.0 MB
- Updated 3.8 MB (+27%)

Patch Load Times



- Roughly linear in size of patch
- Worst case 5 ms

Selected 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 safety analysis, limited flexibility in some cases

Selected Related Work

- Updates to Operating Systems
  - K42 [USENIX 2005]: custom construction
  - LucOS [VEE 2006]: updates coordinated by VM
  - DynamOS [EuroSys 2007]: kernel binary rewriting with shadow data structures
- Updates to Applications
  - OPUS [USENIX Security 2005]: security updates via binary modification to apps
- In all cases, updates fairly simple, with only limited or partial safety analysis

Future Work

- Support OSes and embedded systems
  - Deal with concurrency
  - Support non slop-based type updating
- Stronger reasoning about updates
  - Provable update availability, progress, encapsulation, …
- Optimizations
  - Remove redundant indirections and cons

Conclusion:
DSU can be practical

- Can support changes to applications as they occur in practice
  - Existing apps already constructed to be amenable to DSU
- Process is largely automated
- Performance overhead negligible for I/O-bound applications
- But still want stronger static reasoning