Practical Dynamic Software Updating for C

Iulian Neamtiu **Michael Hicks** **Gareth Stoyle** **Manuel Oriol**

University of Maryland University of Cambridge ETH Zurich

Motivation

- Software updates disruptive
  - Inconvenient, expensive
- Dynamic Software Updating (DSU)
- Where is DSU useful ?
  - Long-running servers (preserve state)
  - Operating systems
  - Transaction processing
  - Middleware
- Where is it not ?
  - Applications with externalized/short lived state: mail server, web server

DSU Challenges

1. No extensive changes to applications
2. Restrict the form of dynamic updates as little as possible
3. Dynamic updates should be
   - easy to write
   - easy to establish as correct

- Ginseng – a system for *practical* DSU

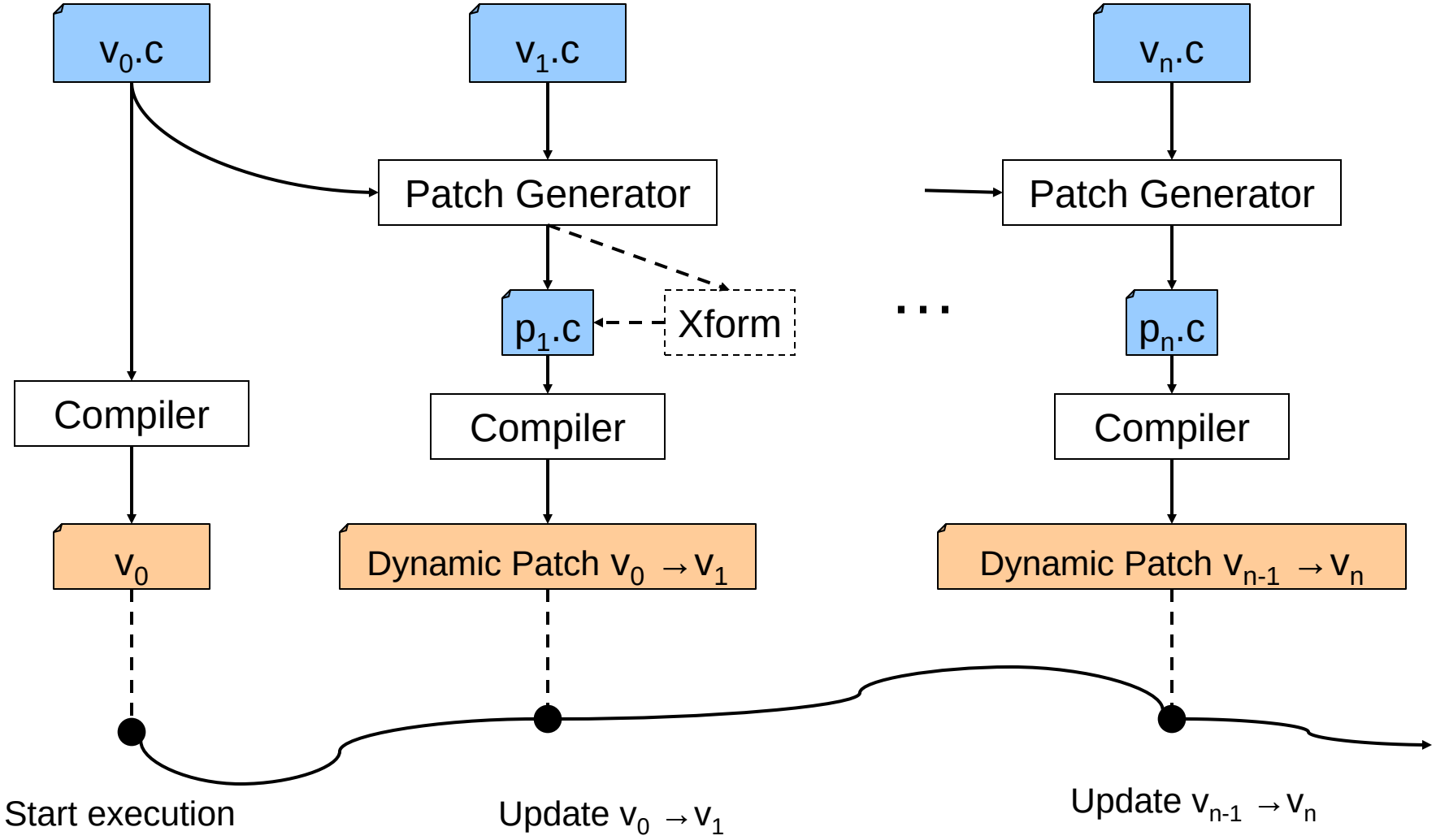
Our Results

- Applications
  - Very Secure FTP Daemon: 3 years, 13 releases
  - OpenSSH sshd: 3 years, 11 releases
  - GNU Zebra: 4 years, 6 releases
- Performance
  - Application throughput: unaffected
  - Application latency: 0..32% overhead
  - Memory footprint: 0..40% overhead
  - Update application time: less than 5 *ms*
- Ease of use
  - Applications largely unchanged
  - Less than 200 lines of application rewritten across all releases

Ginseng Novel Techniques

- Flexible system for dynamic updates
  - Can change types for all definitions
  - Updating types on the stack
  - Functions that refer to changed types
  - Function pointers updateable
- Loop extraction
  - Update long-running loops
- Strong safety guarantees
  - Updateability analysis [POPL'05] extended to full C
  - Abstraction-violating alias analysis

Ginseng Architecture



Updating Model

- Dynamic patches
  - Code/data, transformers
- Updates
  - Happen at user-specified program points
  - Atomic
  - Take effect at function call boundaries (next call always to most recent version)
- Updates to types are lazy
  - Check version, update the type
- Representation consistency
  - All program parts assume same type representation
- Single-threaded programs

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}


void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
  struct S s = {1};
  int n = (con(s)).i;
  *(assign__ptr)(&(con(s)).i, 20);
  update();
  *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Function indirection

Type accesses via **con**

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}


void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
  struct S s = {1};
  int n = (con(s)).i;
  *(assign__ptr)(&(con(s)).i, 20);
  update();
  *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}


void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
  struct S s = {1};
  int n = (con(s)).i;
> *(assign__ptr)(&(con(s)).i, 20);
  update();
  *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}


void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
  struct S s = {20};
  int n = (con(s)).i;
  *(assign__ptr)(&(con(s)).i, 20);
  update();
  *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {20};
  int n = (con(s)).i;
  *(assign__ptr)(&(con(s)).i, 20);
  update();
>
  *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Core Ginseng Mechanisms

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  assign(&s.i, 20);
  update();
  assign(&s.i, 42);
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {.};    {20}
  int n = (con(s)).i;
  *(assign__ptr)(&(con(s)).i, 20);
  update();
> *(assign__ptr)(&(con(s)).i, 42);
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Updating Long-running Loops

```
void main()
…
  while(1) {
    S1;
  }
…
}
```

Version 1

```
void main()
…
  while(1) {
    S1';
  }
…
}
```

Version 2

Updating Long-running Loops

```
#pragma __DSU_loop("LOOP1")
```

Loop extraction

```
LOOP1:while(1) {
  …
  S1;
  …
}
S2;
```

Updating Long-running Loops

```
#pragma __DSU_loop("LOOP1")

LOOP1:while(1) {
  …
  S1;
  …
}
S2;
```

Loop extraction →

```
int LOOP1_body__v0(struct loop_state ls) {
  …
  S1;
  …
}
int (*LOOP1_body__ptr)()=&LOOP1_body__v0;

LOOP1:while(1) {
  struct loop_state ls = & local_state;
  (*LOOP1_body__ptr)(ls); // S1
}

S2;
```

Original code

Ginseng-compiled code

"Hoist" `S1` into a function, use the function update mechanism

Can update arbitrary code on the stack (S2)

Safety Analysis

- Changes in types/prototypes problematic
- Representation consistency
  - Old code can access new types, new function signatures
- Problems
  - Direct access
    - Updateability analysis [Stoyle et al., POPL'05]
    - Extended for function types, full C
  - Revealing representation through an alias
    - Abstraction-violating analysis

Updateability Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}


void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
  struct S s = {1};
  int n = (con(s)).i;
> update();
  *(assign__ptr)(&(con(s)).i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Updateability Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {1};
  int n = (con(s)).i;
  update();
> *(assign__ptr)(&(con(s)).i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Updateability Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {.};      {1}
  int n = (con(s)).i;
  update();
> *(assign__ptr)(&(con(s)).i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Updateability Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {.};        {1}
  int n = (con(s)).i;
  update();
> *(assign__ptr)(&(con(s)).i, 20);
  update();
  …                        int **
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Type Error !

Updateability Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){…}
void assign__v1(int *p, int val){…}

void (*assign__ptr)(int)=&assign__v1;
void foo__v0() {
  struct S s = {1};                    {S, assign}
  int n = (con(s)).i;                  {S, assign}
  update();        NOT SAFE            {S, assign}
  *(assign__ptr)(&(con(s)).i, 20); {S,
                                        assign}
  update();  {}  SAFE
  …
}
```

“Capability” to use a type *concretely*

Abstraction-violating Analysis

- Aliases might expose representation
- Live alias into `T` → can't update `T`
- Pointer analysis to detect live aliases
  - Interprocedural, flow-sensitive, context-insensitive
  - Live aliases at each program point

Abstraction-violating Analysis

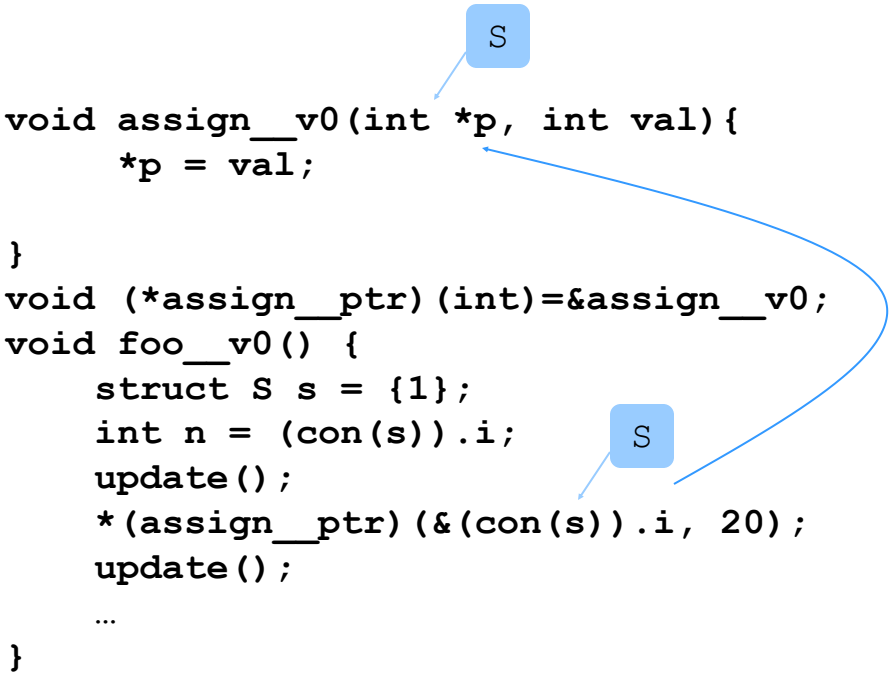
Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Ginseng-compiled code



Abstraction-violating Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

Ginseng-compiled code

```
void assign__v0(int *p, int val){
 {S} *p = val;

}
void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
    struct S s = {1};
    int n = (con(s)).i;
 {} update();
{S} *(assign__ptr)(&(con(s)).i, 20);
 {} update();
    …

}
```

S

S

Live aliases into types

Abstraction-violating Analysis

Original code
Version 1

```
struct S {
  int i;
};

void assign(int *p, int val){
  *p = val;
}

void foo() {
  struct S s = {1};
  int n = s.i;
  update();
  assign(&s.i, 20);
  update();
  …
}
```

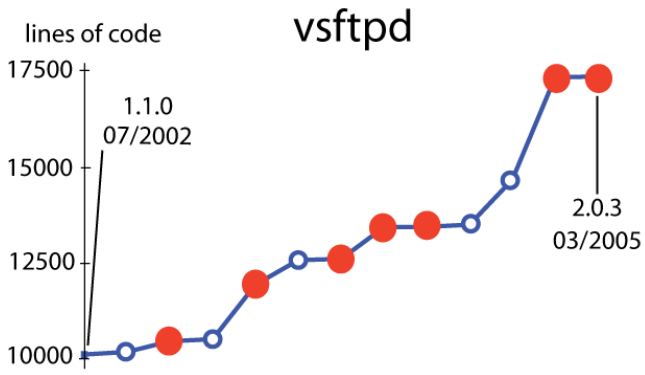
Original code
Version 2

```
struct S {
  int *i;
};

void assign(int *p, int val) {
  *p = val + 1;
}
```

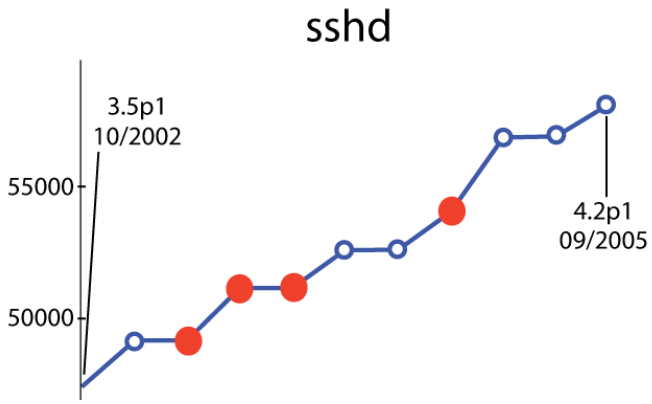
Ginseng-compiled code

```
void assign__v0(int *p, int val){
 {S} *p = val;                          {S}

}
void (*assign__ptr)(int)=&assign__v0;
void foo__v0() {
    struct S s = {1};          {S, assign}
    int n = (con(s)).i;        {S, assign}
 {} update();                  {S, assign}
{S} *(assign__ptr)(&(con(s)).i, 20);{S,
                                    assign}
 {} update(); {}
    …
}
```

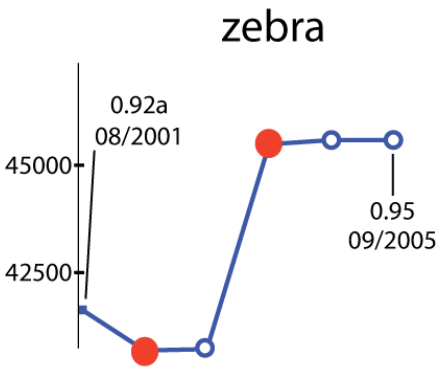
Programs Change a Lot in Three Years



3 years
13 releases

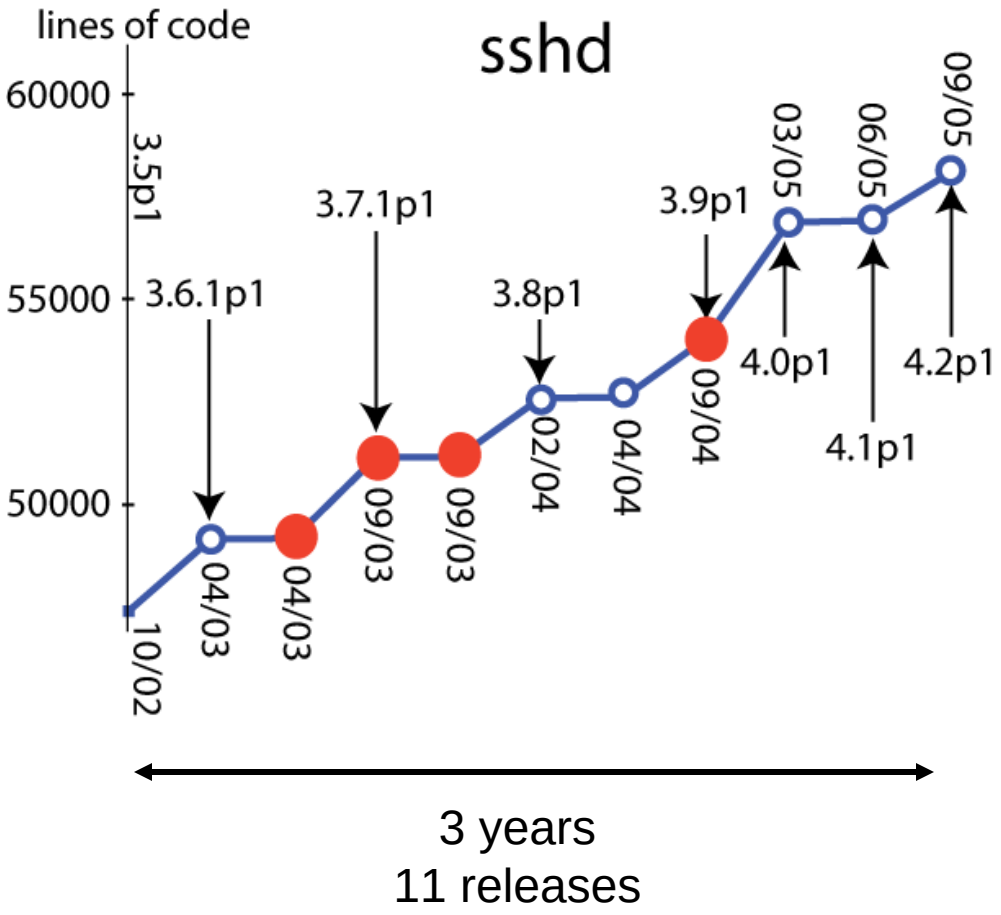


3 years
11 releases



4 years
6 releases

Sshd Evolution History



- Functions
  - 131 added, 19 deleted
  - 85 proto changed
  - 752 body changed
- Types
  - 27 added, 2 deleted
  - 19 changed
- Global variables
  - 70 added, 19 deleted
  - 29 changed

Dynamic Update Catalysts

1. Quiescence
2. Functional state transformation
3. Type-safe programs
4. Robust design

Quiescence

- No in-flight transactions
- Consistent global state
- Shallow stack
- Quiescent point → update point

Quiescent Points Are Easy to Find

```
while(1) {
    …
    newsock = accept();
    fork_child(newsock);

    …

    update();
}
```


quiescent point

Functional State Transformation

- Assumption: can convert global state
  - New_state = f(Old_state)
- No guarantees
  - Assumption might not hold (2 out of 27 updates)
  - Can recover/compensate

Type-safe Programs

- Low-level idioms hamper updateability
  - Illegal casts, inline assembly
  - Non-updateable types
  - Restrict range of updates
- **`void *`**
  - C lacks polymorphism
  - Usually benign

Robust Design

- Global invariants
  - Updates must preserve invariants
  - Usually implicit
  - Explicit invariants - `assert`
- Test suites

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
  - Less than 5 ms

Programming Effort

	Source code (LOC)		
Application	Application changes	Type+state transformers	Patch generator (automatic)
vsftpd	< 50	162	83965
sshd	< 50	125	248587
zebra	< 50	49	43173

Related Work

- **Dynamic software updating** - Hicks et al., PLDI'01
- **K42 Operating System** - IBM
- **Lazy modular upgrades in persistent object stores** - Boyapati et al., OOPSLA'03
- **OPUS: Online patches and updates for security** - Altekar et al., USENIX Security'05
- **Devirtualizable virtual machines enabling general, single-node, online maintenance** - Lowell et al., ASPLOS'04
- **Reducing Downtime Due to System Maintenance and Upgrades** - Potter et al., LISA'05
- **Live Updating Operating Systems Using Virtualization** - Chen et al., VEE'06

Conclusions

- Ginseng – a system for building DSU software
- 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
  - Three years worth of updates, 27 updates
  - Low performance overhead
- Download Ginseng at
  - http://www.cs.umd.edu/projects/dsu/