Safe and Automated State Transfer for Secure and Reliable Live Update

Cristiano Giuffrida  Andrew S. Tanenbaum

Vrije Universiteit Amsterdam

Fourth International Workshop on Hot Topics in Software Upgrades

Zurich, Switzerland
June 3, 2012
Filling the gap between software evolution and high availability

Goal: deploying updates online with no downtime or loss of state

New features, performance improvements, bug fixes, security patches

High availability originally a property of mission critical systems

Nowadays, the demand for 24/7 operation is pervasive
Updating Your Computer FAIL

Safe and Automated State Transfer for Secure and Reliable Live Update

Cristiano Giuffrida, Andrew S. Tanenbaum
State Transfer

- An important problem in live update systems
- State transfer code executes in a unique context
- Nonstandard entry points and programming model
- Minimizing state transfer complexity is critical
- Prior solutions: limited support to automate/safeguard state transfer
- Security and reliability issues
A New State Transfer Framework

- Automated state transfer and checking with minimal manual effort
- Automated pointer transfer and dynamic object reallocation
- Handles arbitrarily complex state changes
- Tainted state management
- Hot rollback
- Support for all the standard C idioms
The Framework in a Nutshell

Process-level updates

V1 → Update → V2
The Framework in a Nutshell

Nonintrusive compiler-based instrumentation
Precise run-time state introspection
Support for different update models
The State Transfer Process

State transfer monitor
The State Transfer Process

V1
State
Metadata
Update point

SYNC

State transfer monitor

V2
State
Metadata
The State Transfer Process

V1
State
Metadata
Update point

V2
State
Metadata

INIT

State transfer monitor

Safe and Automated State Transfer for Secure and Reliable Live Update

Cristiano Giuffrida, Andrew S. Tanenbaum
The State Transfer Process

V1
State Metadata
Update point

IPC-based Metadata transfer

V2
State Metadata

TRACE

State transfer monitor
The State Transfer Process

Safe and Automated State Transfer for Secure and Reliable Live Update

Cristiano Giuffrida, Andrew S. Tanenbaum
The State Transfer Process

State transfer monitor

V1
State
Metadata
Update point

Control flow transfer

V2
State
Metadata
Update point

TRACE
The State Transfer Process

V1
State
Metadata
Update point

V2
State
Metadata
Update point

CLEANUP

TRACE

State transfer monitor
State transfer monitor
State Instrumentation

- Types
- Global/static variables
- String constants
- Functions
- Local variables
- Shared libraries
Dynamic State Objects

**Instrumentation**
- Every dynamic object is associated to a run-time type
- Instrument (only) allocation sites in the original code
- Support for malloc- and mmap-like allocator abstractions

**Example**
- Allocation: `struct s *p = malloc(10 * sizeof(struct s));`
### Dynamic State Objects

#### Instrumentation
- Every dynamic object is associated to a run-time type
- Instrument (only) allocation sites in the original code
- Support for malloc- and mmap-like allocator abstractions

#### Example
- Allocation: `struct s *p = malloc(10 * sizeof(struct s));`
- Instrumentation: `malloc(...)`
Dynamic State Objects

Instrumentation

- Every dynamic object is associated to a run-time type
- Instrument (only) allocation sites in the original code
- Support for malloc- and mmap-like allocator abstractions

Example

- Allocation: `struct s *p = malloc(10 * sizeof(struct s));`
- Instrumentation: `malloc_wrapper(...)`
Dynamic State Objects

Instrumentation

- Every dynamic object is associated to a run-time type
- Instrument (only) allocation sites in the original code
- Support for malloc- and mmap-like allocator abstractions

Example

- Allocation: `struct s *p = malloc(10 * sizeof(struct s));`
- Instrumentation: `malloc_wrapper(...)`
- Static type assigned: `struct s`
Dynamic State Objects

Instrumentation

- Every dynamic object is associated to a run-time type
- Instrument (only) allocation sites in the original code
- Support for malloc- and mmap-like allocator abstractions

Example

- Allocation: `struct s *p = malloc(10 * sizeof(struct s));`
- Instrumentation: `malloc_wrapper(...)`
- Static type assigned: `struct s`
- Run-time type assigned: `[ 10 x struct s ]`
State Objects Pairing

Process-independent naming scheme

Process-independent naming scheme
struct example {
    short flags;
    int data;
    ...
} obj;
void *ptr_t = (void *) &obj.data;

obj: |
| flags | data |
| ... |

ptr_t: |
| 0xf010 |

Pointers with a valid target
Pointer Transfer

```c
#define NULL ((void *) 0)
#define MAP_FAILED ((void *) -1)

void *ptr_n = NULL;
void *ptr_i = MAP_FAILED;
```

| ptr_n: 0 |   |
|----------|
|   X      |

| ptr_i: -1 |   |
|-----------|
|   X       |

Integers as pointers
#define BUFFER_LENGTH 32

cchar buff[BUFFER_LENGTH];
cchar* ptr_g = buff[BUFFER_LENGTH - 1];
...
ptr_g++;
struct example {
    short flags;
    int data;
    ...
} obj;
long int_p = (long) &obj.data;

Pointers as integers
Pointer Transfer

```c
union example_u {
    long magic;
    int *ptr;
} u = { .magic = MAGIC_WORD };
...
u.ptr = &obj.data;
```

Unions with pointers
 Transfer Strategy

```c
struct s { //old version
    int flags;
    char string[3];
    short id;
    union IXFER(my_u) u;
    void *userdata;
    PXMLER(int) address;
} my_s;
short *p = &my_s.id;
```

```c
struct s { //new version
    int flags;
    int id;
    char string[2];
    union IXFER(my_u) u;
    PXMLER(int) address;
    int newfield;
} my_s;
int *p = &my_s.id;
```
State Transfer Extensions

- Type-based/object-based callbacks and annotations
- Handle cases of pointer ambiguity
- Change the default pairing rules
- Selective state transfer
- Custom type transformations
- Tainted state management
Hot Rollback

- State transfer execution sandboxed in the new version
- Monitor detects run-time errors (i.e., crashes, panics, timeouts)
- State checking detects tainted state and unsafe conditions
- Hot rollback restores normal execution in the old version
- Process-based isolation prevents error propagation to the old version
A new safe and automated state transfer framework for C

Automates complex state changes with minimal manual effort

Automates pointer transfer for secure and reliable live update

Supports convenient programming model for extensions

Supports state checking and hot rollback
Safe and Automated State Transfer for Secure and Reliable Live Update

Thank you!
Any questions?

Cristiano Giuffrida, Andy Tanenbaum
{giuffrida,ast}@cs.vu.nl

Vrije Universiteit Amsterdam
State Transfer Discussion Topics

- What if the update time is longer than the restart time?
- Update time vs. manual state transfer effort. A new tradeoff?
- State annotations: a burden or a blessing?
- Hot rollback: a feature you always wanted and never dared to ask?