

I Control Your Code – Attack Vectors Through the Eyes of Software-based Fault Isolation

Mathias Payer, ETH Zurich



Motivation

- Applications often vulnerable to security exploits
- Solution: restrict application access to the minimum amount of data needed
 - Least privilege principle

In a nutshell

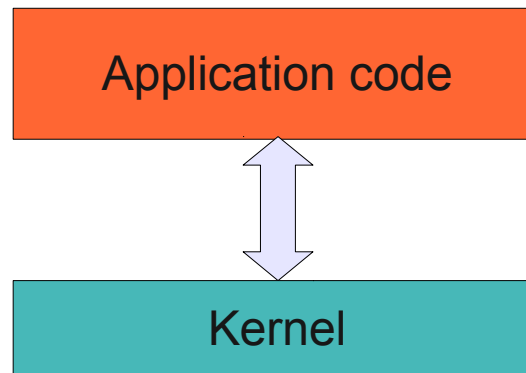
- Fine-grained *virtualization* layer confines security threats
 - All executed code is verified
 - Additional security guards are added to the runtime image
 - All system calls are verified according to a tight policy

Outline

- Introduction
- Security architecture
- Evaluation
- Attack Vectors
- Related work
- Conclusion

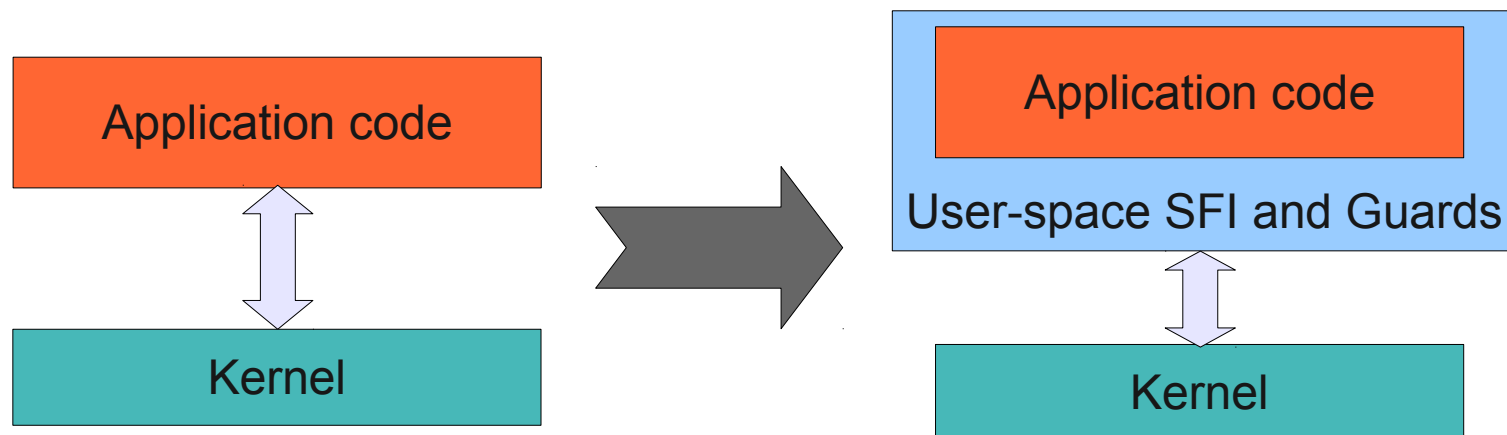
Introduction

- Software security is a challenging problem
 - Many different forms of attacks exist
 - Low-level bugs are omni-present
 - Current security practice is reactive
- We present a pro-active approach to security
 - Catch exploits before they can cause any harm



Protection through virtualization

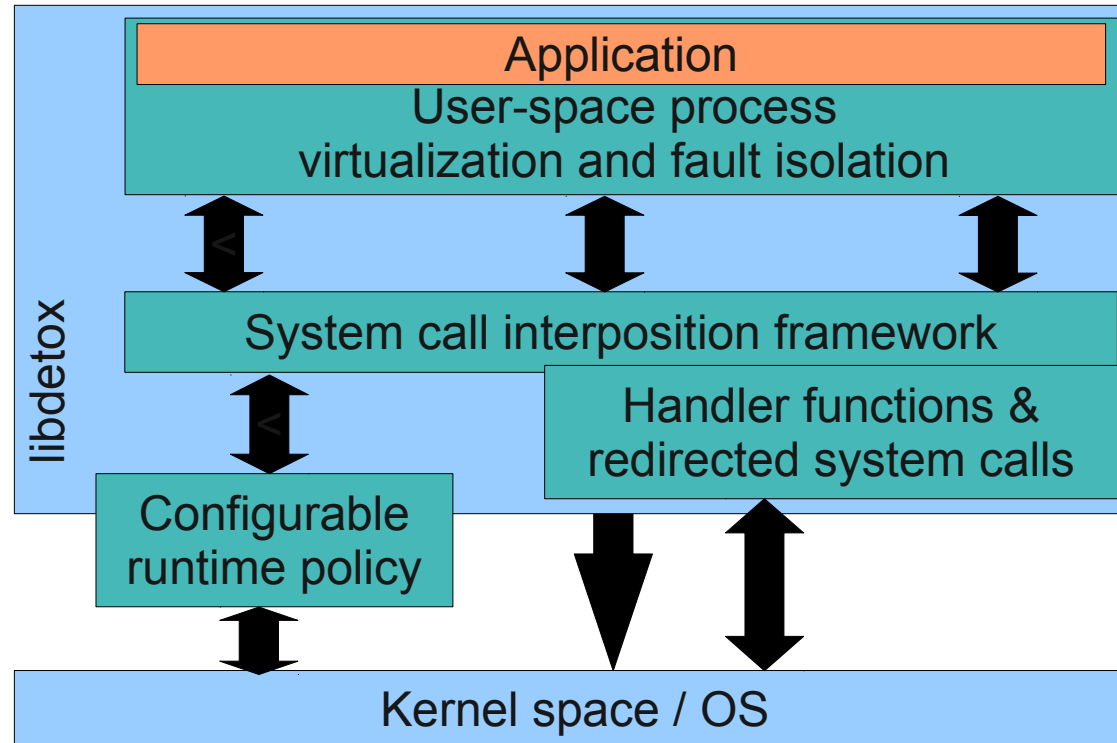
- Use virtualization to confine and secure applications
- Use a user-space virtualization system
 - Secure all code and authorize all system calls



Outline

- Introduction
- Security architecture
 - Software-based fault isolation (SFI)
 - System call interposition
- Evaluation
- Attack Vectors
- Related work
- Conclusion

Security Architecture

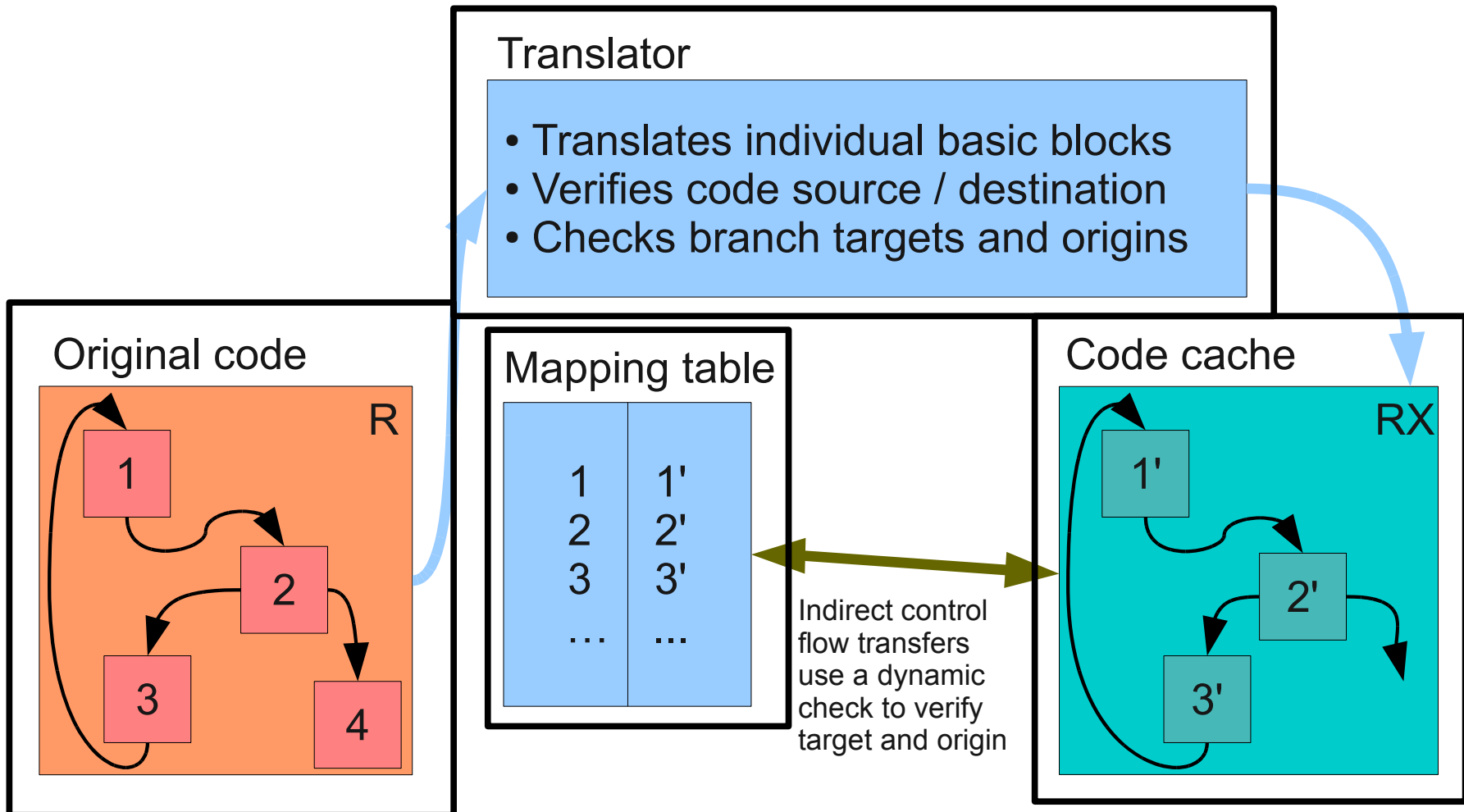


- Layered security concept
 - User-space software-based fault isolation
 - System call interposition framework
 - System call authorization

Software-based fault isolation

- SFI implemented as a user-space library
- All code is translated before it is executed
 - Code is checked and verified on the fly
 - All unsafe instructions are encapsulated or rewritten
 - Check targets and origins of control flow transfers
 - Illegal instructions halt the program

SFI in a nutshell



SFI: Additional guards

- Translator adds guards protect from malicious attacks against the SFI platform and enhances security guarantees
 - Secure control flow transfers
 - Signal handling
 - Executable bit removal
 - Address space layout randomization
 - Protecting internal data structures

SFI: Control transfers

- Verify return addresses on stack
 - Use a shadow stack to store original/translated addresses
 - Protects from Return Oriented Programming
- Secure control flow transfers
 - Check target and source locations for valid transfer points
 - Protects from code injection through heap-based/stack-based overflows

SFI: Signal handling

- Catch signals and exceptions
 - Redirect to installed handlers if signal is valid
 - Protects from break-outs out of the sandbox

SFI: Executable bit removal

- Executable bit removed for libraries and application
 - Only libdetox and code-cache contains executable code
- Part of the protection against code-injection

SFI: ASLR

- Address space layout randomization randomizes the runtime memory image
 - Probabilistic measure that makes attack harder

SFI: Internal data structures

- All internal data structures are protected
 - Context transfer to (translated) application code protects all internal data structures
 - Write permissions to all internal memory is removed
- Protects from code-injection and attacks against the virtualization platform

SFI: Added protection

- These additional guards protect from
 - Code injection (stack-based / heap-based)
 - Return-oriented programming
 - Execution of illegal code
 - Attacks against the virtualization platform

Outline

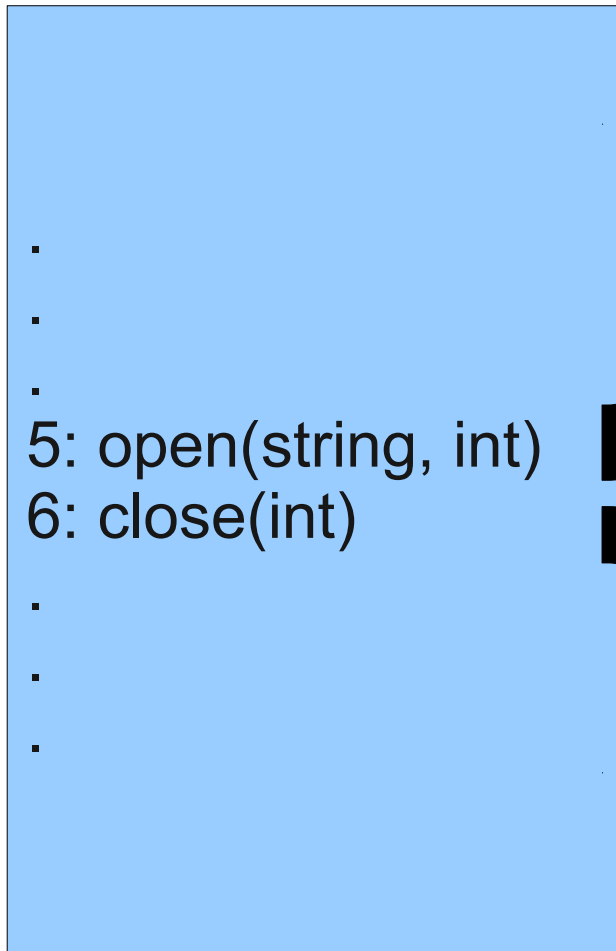
- Introduction
- Security architecture
 - Software-based fault isolation (SFI)
 - System call interposition
- Evaluation
- Attack Vectors
- Related work
- Conclusion

System call interposition

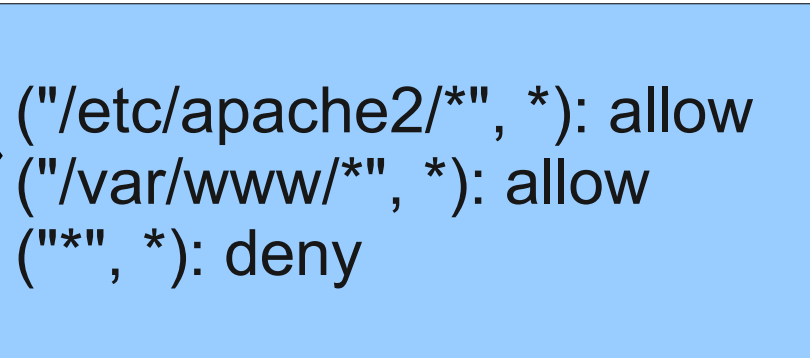
- Implemented on top of SFI platform
- All system calls & parameters are checked
 - Dangerous system calls are redirected to a special implementation inside the virtualization library
- System call authorization
 - System calls are authorized based on a user-definable per-process policy
- Protects from data attacks and privilege escalation

Policy

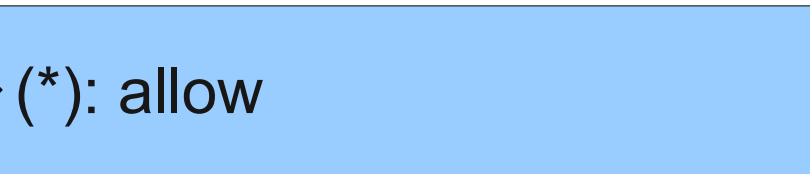
System call definition:



open:



close:



Outline

- Introduction
- Security architecture
- Evaluation
- Attack Vectors
- Related work
- Conclusion

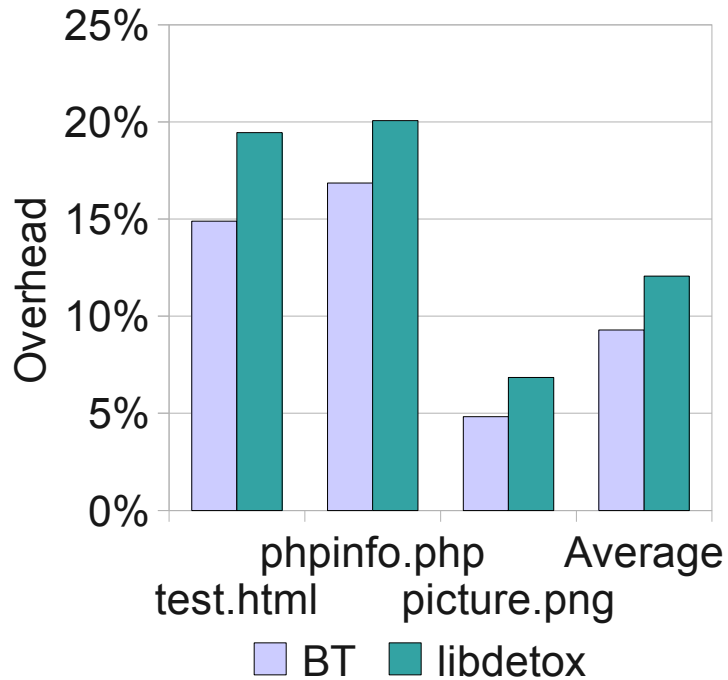
libdetox

- Approach implemented as a prototype
- Built on top of fastBT binary translation system
 - Additional security hardening
 - Guards implemented in the translation process
 - Dynamic guards extend the dynamic control flow transfer logic

Apache2

- Fully protected Apache 2.2.11 is evaluated using the ab benchmark
 - Each file is received 1'000'000 times
 - test.html (static, 1.7kB)
 - phpinfo.php (small, dynamic PHP file)
 - picture.png (static, 242 kB)
- All benchmarks were executed on Ubuntu 9.04 on an E6850 Intel Core2Duo CPU @ 3.00GHz, 2GB RAM and GCC version 4.3.3

Apache2



Throughput [mB/s]	native	BT	libdetox
test.html	22.5	19.6	18.8
phpinfo.php	3.28	2.80	2.73
picture.png	945	902	885
Average overhead	-	9.3%	12%

- Low overhead for real-world server application
- Throughput highly depends on payload
 - Both for virtualized and native executions

Outline

- Introduction
- Security architecture
- Evaluation
- **Attack Vectors**
 - Code injection
 - Return-oriented programming
 - Format string attacks
- Related work
- Conclusion

Attack vectors

- Attacks redirect control flow
 - New or alternate locations are reached
 - Execution is different from unaltered run
- An attack exploits the fact that the programmer or the runtime system is unable to check
 - the bounds of a buffer or
 - to detect a type overflow or
 - to detect an out-of-bounds access

Code injection

- Injects new executable code into the process image of a running process
 - Into buffer on the *stack*
 - Into *heap-based* data structures
- Redirects control flow to the injected code
 - Overwriting the `RIP` (return instruction pointer)
 - Overwriting function pointers, destructors, or data structures of the memory allocator

Code injection: libdetox perspective

- The BT would stop the program when the control flow transfer is detected
 - Before the shellcode is even translated
 - Two exceptions would be triggered
 - Code is (about to be) executed in a non-executable area and
 - Function call to an unexported/unknown symbol (heap-based exploit) or
 - RIP mismatch (stack-based exploit)
- Use BT to analyze exploits/shellcode
 - Catch new exploits and security holes
 - Use debugging info in application to fix bugs
 - Use BT to audit your own software / test your exploits

Return-oriented programming

- Exploit already existing code sequences
 - Prepare the stack so that tails of library functions are executed one after another
- Stack-based overflow is used to prepare multiple stack invocation frames
 - Control flow redirected to tails of library functions
 - Tails can be used to execute arbitrary code
- Constraints
 - Missing bound check for the initial stack-based overflow
 - RIP must not be checked
- See: Return-Oriented Programming (Shacham, Black Hat'08)

ROP: libdetox perspective

- The BT would stop the program when the control flow transfer is detected
 - Before the function tail or libC function is translated
 - Shadow stack guard detects mismatch
- Real attacks chain multiple `libc` calls
 - Can be used to inject code into the address space in a legal manner (use `mprotect` to update permissions)

Format string attack

- Exploit the parsing possibilities of the `printf`-family
 - If a user-controlled string is passed to a `printf` function
- A combination of `%x` and `%n` in strings that are passed to `printf` unfiltered result in random memory reads and random memory writes
 - Careful preparation of the input is needed
- The format string must be allowed to contain `%n` and, e.g., `%x` to write to memory
 - Random writes can be used to redirect the control flow by overwriting, e.g., the `RIP`, destructors, or the `vtable`

Format string: libdetox perspective

- BT stops the program when the control flow is redirected
 - Illegal control flow transfer
 - Shadow stack guard or control flow guard
 - System call guard checks system calls arguments
 - Policy violations are detected and the program is stopped
- Random writes to memory only detectable with full memory tracking

Outline

- Introduction
- Security architecture
- Evaluation
- Attack Vectors
- Related work
- Conclusion

Related Work

- Full system translation (VMWare, QEMU, Xen)
 - Virtualizes a complete system, management overhead, data sharing problem
- System call interposition (Janus, AppArmor)
 - Only system calls checked, code is unchecked
- Software-based fault isolation (Vx32, Strata)
 - Only a sandbox is not enough, additional guards and system call authorization needed
- Static binary translation (Google's NaCL)
 - Limits the ISA, special compilers needed

Conclusions

- Combining SFI and policy-based system call authorization builds low overhead virtualization platform
 - Virtualization based on programs, not systems
 - System image is shared with a single configuration
- Fine-grained access control to data / properties
- Opens door to new approaches of security
 - Highly customizable and dynamic

Questions



- Libdetox as an implementation prototype supports full IA-32 ISA without kernel module
 - Source: <http://nebelwelt.net/projects/libdetox/>