Native Client

Kapil Anand
Motivation

• Web browsers employ languages like JavaScript.

• Not suitable for performance critical applications
  – Games (High speed rendering)
  – Fluid dynamics

• Inability to execute native machine code due to security concerns
  – User based security (Popup) not effective
NativeClient

- Sandbox environment for execution of native code
  - Native code performance
  - Portability across OS

- Two main parts
  - Constrained execution of native code
  - Runtime for allowing safe operations
Design

- **Sandbox**
  - Static analysis to detect security defects
  - Enforces rules on incoming un-trusted modules

- **Runtime Facilities**
  - Allows safe execution of possible “unsafe” operations
  - Inter-module communications
Sandbox

- Implements Software Fault Isolation
  - External state not accessible from these modules
  - Explicit control flow instructions

- Stipulates a set of rules need to be followed by the incoming binary
  - No unsafe instructions
  - Special indirect control transfers
  - Relies on compiler tool chain to stick to these rules

- Implements a validation tool to check these rules
  - Reliable disassembly
Binary Constrains

• No self modifying code
  – Not writable after loading into memory

• Statically linked with a fixed start address of text segment

• All instructions reachable by fall-through disassembly from starting address
  – Relocation information present (?), not mentioned in paper
  – Ensure reliable disassembly

• Other constraints
  – Syscall, Instructions modifying segment state, ret
  – No instruction overlapping a 32 byte
  – All functions, basic blocks aligned to 32 byte boundary
Data Flow Integrity

• Data Integrity:
  – Employs segmented memory support

  – Constrains data references to a particular sub range of virtual 32 bit addresses

  – Effective as segment registers not accessible
Control flow integrity

• All control transfers target an instruction identified during disassembly

• Direct control flow
  – Target should be one of reachable instruction

• Indirect Control flow
  – Segmented support (works because a fix start address)
  – No returns
  – Limit target alignment to 32 byte boundary
    
    \[
    \text{jmp eax} \rightarrow \text{and eax,0xffffffe0} \\
    \text{jmp eax}
    \]
  – Implemented as a special “NaCl” jump
  – Lack of segmented support in previous paper required use of “and” and “or” instruction to check the segment tag
CFI

• All instructions and pseudo-instructions (including NaCl jmp) are aligned to 32 byte boundary

• Any indirect jump would follow an “and” and hence goes to one of the instruction on 32 byte boundary

• All “32-byte” boundary instructions are statically analyzed

• Insures
  – No skipping of “and” before “jmp”
  – NaCl becomes atomic
Possible Attacks

• No Return based attacks as returns not allowed

• No Code Injection

• Return to libc or Gadgets
  – Not possible to obtain something like “system” command as middle of instructions not accessible
Exceptions

- Hardware exceptions not allowed as stack mechanism is invalidated
- Immediate termination and no recovery
- Supports C++ exceptions
Runtime Service

- Sandboxing limits unsafe operations
- Runtime services allow some “unsafe” operations to be operated in “safe” way
  - Required for usefulness of modules
- Isolates incoming binary from host resources
  - Resource abstractions
- Communication
  - Between Native Client and Browser
  - Communication between Native Client Modules
  - Client module and network like accessing URLs
Runtime Services

• Contains trusted code and data

• Implemented as a native executable invoked by browser plug-in

• The process is shared with Native client module

• Implements dynamic enforcements
  – Maintains integrity of sandbox
Runtime Service

- Small part of module space is used by runtime service

- Enables control transfer between untrusted user code and trusted runtime service

- Untrusted code allowed in these trampolines like system calls

- Modifies segment registers and native code executed with normal flat addressing
Performance

• Compiler tool chain need to be modified
  – Alignment constraints
  – Instruction constraints
  – Special control transfers

• Average 5% performance overhead on SPEC benchmarks
  – Good result
Limitations

• No hardware handled exceptions

• Tool-chain modifications

• Applications limited to computation oriented domains:
  – File system accesses, process creation not allowed
  – Not suitable for OS extensions (User mode only)
Fast Byte Granularity
Software Fault Isolation
Motivation

• Kernel Extensions are used for customizing operating systems
  – Device drivers
  – File systems
• Fully trusted by kernel for performance reasons
• Problems
  – Written in unsafe languages
  – Share address space with kernel
  – Most of OS bugs are related to extensions
Motivation

• Existing approaches:
  – Extensions in separate protection domain or in user mode processes
  – Either require hardware modifications or high overhead because extensions used very frequently
  – SFI in Native Client not applicable as it relied on segmented protections

• A software based Byte-Granularity SFI
  – Previous solutions required hardware changes
  – A complete software solution which isolates drivers with no change in source-code and with no performance overheads
Previous Mechanisms

• Not enough isolation
  – For example, a range of addresses are allowed to be modified
    – Do not prevent writes to kernel objects stored in these addresses

• No checking of call interface
  – Calling a function with wrong interface
BGI

• Runs code in separate protection domain to contain software faults

• Kernel and trusted extensions run in trusted domain

• An Access control list (ACL) specifying the rights of each domain to each byte
Protection Model

\[ \begin{array}{c}
\text{d}_0: \text{trusted domain} \\
\text{kernel} \\
\text{driver 1} \\
\text{d}_1: \text{untrusted domain} \\
\text{driver 2} \\
\text{driver 3} \\
\text{d}_2: \text{untrusted domain} \\
\text{driver 5} \\
\text{d}_3: \text{untrusted domain} \\
\text{driver 6} \\
\text{driver 7} \end{array} \]
Access Rights

- Read: available to everyone
- Write
- iCall
  - Ensures control flow integrity
  - An extension can call a function only if it has iCall rights on that function
  - Applicable to function pointers also by analyzing address creation points

- Type
  - Enforce dynamic type safety for kernel objects

- Ownership:
  - Maintains the domain which should free the object and type of de-allocation to use
Memory checks

• Write Checks
  – Prevent unauthorized write
  – Global variables rights
  – Local variables rights
  – Heap rights

• Ownership rights
  – Controlled by memory allocation/de-allocation wrappers
Memory checks

• Call checks:
  – Prevent threads from executing code in another domain
  – Checks whether current extension has call right before calling through indirect call
  – For function pointers passed to kernel from extensions, it is checked whether the extension has call right

• Type Checks:
  – Dynamic type for each object
  – Wrappers check that corresponding domain has proper type rights on incoming object
  – Controlled by allocators/de-allocator wrappers
Implementation

- **Compiler** changes to compile the driver source code with required changes
  - Assumes that driver developers are benign

- **Interpositional library** to be linked with driver

- **ACL Tables** for controlling accesses
Interposition Library

• Mediates communication between extensions and kernel
  – All communication through wrappers

• Kernel wrappers
  – Wrap kernel functions called by extension

• Extension wrappers
  – Wrap extension wrappers called by functions

• Wrappers grants, checks or revokes the access protection
  – Trust kernels (Guides the operation)
Compiler

- Instruments the extension to redirect kernel function calls to interposition wrappers
- Modifies the address creation points to point to library wrappers
- Access rights maintenance for local variables
- Access-Check instruction before each write and indirect call
- Disallows inline assembly
Attack Protection

• No indirect branches

• Information about possible functions in module needs to be in read only segment

• Safety of Access Tables

• Buffer overflows possible
  – As protection is only on domains, not on functions
  – Inject code to bypass inter-positional library

• Return based programming
Byte Granularity Protection

• Suggests fast table based mechanism for managing access rights

• A small integer to encode a pair of domain and access right

• Kernel table and a user table per process
  – Corresponding conflict tables

• An entry for each 8 byte memory in the corresponding table specifying access right
Example
Performance

- Contains ~98% blue screen faults and ~50% hangs

- Performance overhead of 12% in one benchmarks, <2% in others, 6.4% on average, as compared to around 20% in previous paper