Will appear in the 2009 IEEE Symposium on Security and Privacy

Native Client: A Sandbox for Portable, Untrusted x86 Native Code

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth,
Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar
Google Inc.

Abstract

This paper describes the design, implementation and evaluation of Native Client, a sandbox for untrusted x86 native code. Native Client aims to give browser-based applications the computational performance of native applications without compromising safety. Native Client uses software fault isolation and a secure runtime to direct system interaction and side effects through interfaces managed by Native Client. Native Client provides operating system portability for binary code while supporting performance-oriented features generally absent from web application programming environments, such as thread support, instruction set extensions such as SSE, and use of compiler intrinsics and hand-coded assembler. We combine these properties in an open architecture that encourages community review and 3rd-party tools.

1. Introduction

As an application platform, the modern web browser brings together a remarkable combination of resources, including seamless access to Internet resources, high-productivity programming languages such as JavaScript, and the richness of the Document Object Model (DOM) [64] for graphics presentation and user interaction. While these strengths put the browser in the forefront as a target for new application development, it remains handicapped in a critical dimension: computational performance. Thanks to Moore's Law and the zeal with which it is observed by the hardware community, many interesting applications get adequate performance in a browser despite this handicap. But there remains a set of computations that are generally infeasible for browser-based applications due to performance constraints, for example: simulation of Newtonian physics, computational fluid-dynamics, and high-resolution scene rendering. The current environment also tends to preclude use of the large bodies of high-quality code developed in languages other than JavaScript.

Modern web browsers provide extension mechanisms such as ActiveX [15] and NPAPI [48] to allow native code to be loaded and run as part of a web application. Such architectures allow plugins to circumvent the security mechanisms otherwise applied to web content, while giving them access to full native performance, perhaps as a secondary consideration. Given this organization, and the absence of effective technical measures to constrain these plugins, browser applications that wish to use native-code must rely on non-technical measures for security; for example, manual establishment of trust relationships through pop-up dialog boxes, or manual installation of a console application. Historically, these non-technical measures have been inadequate to prevent execution of malicious native code, leading to inconvenience and economic harm [10], [54]. As a consequence we believe there is a prejudice against native code extensions for browser-based applications among experts and distrust among the larger population of computer users.

While acknowledging the insecurity of the current systems for incorporating native-code into web applications, we also observe that there is no fundamental reason why native code should be unsafe. In Native Client, we separate the problem of safe native execution from that of extending trust, allowing each to be managed independently. Conceptually, Native Client is organized in two parts: a constrained execution environment for native code to prevent unintended side effects, and a runtime for hosting these native code extensions through which allowable side effects may occur safely.

The main contributions of this work are:

- an infrastructure for OS and browser-portable sandboxed x86 binary modules,
- support for advanced performance capabilities such as threads, SSE instructions [32], compiler intrinsics and hand-coded assembler,
- an open system designed for easy retargeting of new compilers and languages, and
- refinements to CISC software fault isolation, using x86 segments for improved simplicity and reduced overhead.

We combine these features in an infrastructure that supports safe side effects and local communication. Overall, Native Client provides sandboxed execution of native code and portability across operating systems, delivering native code performance for the browser.

The remainder of the paper is organized as follows. Section 1.1 describes our threat model. Section 2 develops some essential concepts for the NaCl[1] system architecture and

1. We use "NaCl" as an adjective reference to the Native Client system.

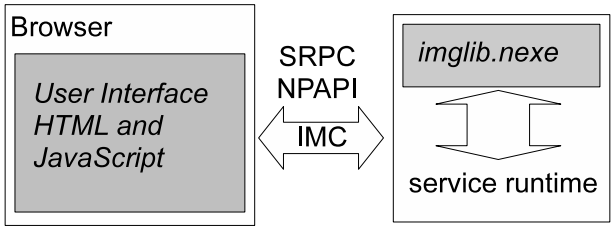

Figure 1: Hypothetical NaCl-based application for editing and sharing photos. Untrusted modules have a grey background.

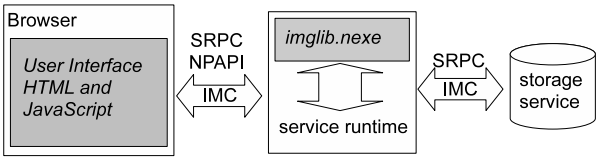

Figure 2: The hypothetical photo application of Figure 1 with a trusted storage service.

programming model. Section 3 gives additional implementation details, organized around major system components. Section 4 provides a quantitative evaluation of the system using more realistic applications and application components. In Section 5 we discuss some implications of this work. Section 6 discusses relevant prior and contemporary systems. Section 7 concludes.

1.1. Threat Model

Native Client should be able to handle untrusted modules from any web site with comparable safety to accepted systems such as JavaScript. When presented to the system, an untrusted module may contain *arbitrary* code and data. A consequence of this is that the NaCl runtime must be able to confirm that the module conforms to our validity rules (detailed below). Modules that don't conform to these rules are rejected by the system.

Once a conforming NaCl module is accepted for execution, the NaCl runtime must constrain its activity to prevent unintended side effects, such as might be achieved via unmoderated access to the native operating system's system call interface. The NaCl module may arbitrarily combine the entire variety of behaviors permitted by the NaCl execution environment in attempting to compromise the system. It may execute any reachable instruction block in the validated text segment. It may exercise the NaCl application binary interface to access runtime services in any way: passing invalid arguments, etc. It may also send arbitrary data via our intermodule communication interface, with the communicating peer responsible for validating input. The NaCl module may allocate memory and spawn threads up to resource limits. It may attempt to exploit race conditions in subverting the system.

We argue below that our architecture and code validity rules constrain NaCl modules within our sandbox.

2. System Architecture

A NaCl application is composed of a collection of trusted and untrusted components. Figure 1 shows the structure of a hypothetical NaCl-based application for managing and sharing photos. It consists of two components: A user interface, implemented in JavaScript and executing in the web browser, and an image processing library (imglib.nexe), implemented as a NaCl module. In this hypothetical scenario, the user interface and image processing library are part of the application and therefore untrusted. The browser component is constrained by the browser execution environment and the image library is constrained by the NaCl container. Both components are portable across operating systems and browsers, with native code portability enabled by Native Client. Prior to running the photo application, the user has installed Native Client as a browser plugin. Note that the NaCl browser plugin itself is OS and browser specific. Also note it is trusted, that is, it has full access to the OS system call interface and the user trusts it to not be abusive.

When the user navigates to the web site that hosts the photo application, the browser loads and executes the application JavaScript components. The JavaScript in turn invokes the NaCl browser plugin to load the image processing library into a NaCl container. Observe that the native code module is loaded silently—no pop-up window asks for permission. Native Client is responsible for constraining the behavior of the untrusted module.

Each component runs in its own private address space. Inter-component communication is based on Native Client's reliable datagram service, the IMC (Inter-Module Communications). For communications between the browser and a NaCl module, Native Client provides two options: a Simple RPC facility (SRPC), and the Netscape Plugin Application Programming Interface (NPAPI), both implemented on top of the IMC. The IMC also provides shared memory segments and shared synchronization objects, intended to avoid messaging overhead for high-volume or high-frequency communications.

The NaCl module also has access to a "service runtime" interface, providing for memory management operations, thread creation, and other system services. This interface is analogous to the system call interface of a conventional operating system.

In this paper we use "NaCl module" to refer to untrusted native code. Note however that applications can use multiple NaCl modules, and that both trusted and untrusted components may use the IMC. For example, the user of the photo

application might optionally be able to use a (hypothetical) trusted NaCl service for local storage of images, illustrated in Figure 2. Because it has access to local disk, the storage service must be installed as a native browser plugin; it can't be implemented as a NaCl module. Suppose the photo application has been designed to optionally use the stable storage service; the user interface would check for the stable storage plugin during initialization. If it detected the storage service plugin, the user interface would establish an IMC communications channel to it, and pass a descriptor for the channel to the image library, enabling the image library and the storage service to communicate directly via IMC-based services (SRPC, shared memory, etc.). In this case the NaCl module will typically be statically linked against a library that provides a procedural interface for accessing the storage service, hiding details of the IMC-level communications such as whether it uses SRPC or whether it uses shared memory. Note that the storage service must assume that the image library is untrusted. The service is responsible for insuring that it only services requests consistent with the implied contract with the user. For example, it might enforce a limit on total disk used by the photo application and might further restrict operations to only reference a particular directory.

Native Client is ideal for application components requiring pure computation. It is not appropriate for modules requiring process creation, direct file system access, or unrestricted access to the network. Trusted facilities such as storage should generally be implemented outside of Native Client, encouraging simplicity and robustness of the individual components and enforcing stricter isolation and scrutiny of all components. This design choice echoes microkernel operating system design [2], [12], [25].

With this example in mind we will now describe the design of key NaCl system components in more detail.

2.1. The Inner Sandbox

Native Client is built around an x86-specific intra-process "inner sandbox." We believe that the inner sandbox is robust; regardless, to provide defense in depth [13], [16] we have also developed a second "outer sandbox" that mediates system calls at the process boundary. The outer sandbox is substantially similar to prior structures (systrace [50] and Janus [24]) and we will not discuss it in detail in this paper.

The inner sandbox uses static analysis to detect security defects in untrusted x86 code. Previously, such analysis has been challenging for arbitrary x86 code due to such practices as self-modifying code and overlapping instructions. In Native Client we disallow such practices through a set of alignment and structural rules that, when observed, insure that the native code module can be disassembled reliably, such that all reachable instructions are identified during disassembly. With reliable disassembly as a tool, our validator can then insure that the executable includes only the subset of legal instructions, disallowing unsafe machine instructions.

The inner sandbox further uses x86 segmented memory to constrain both data and instruction memory references. Leveraging existing hardware to implement these range checks greatly simplifies the runtime checks required to constrain memory references, in turn reducing the performance impact of safety mechanisms.

This inner sandbox is used to create a security subdomain within a native operating system process. With this organization we can place a trusted service runtime subsystem within the same process as the untrusted application module, with a secure trampoline/springboard mechanism to allow safe transfer of control from trusted to untrusted code and vice-versa. Although in some cases a process boundary could effectively contain memory and system-call side effects, we believe the inner sandbox can provide better security. We generally assume that the operating system is not defect free, such that the process barrier might have defects, and further that the operating system might deliberately map resources such as shared libraries into the address space of all processes, as occurs in Microsoft Windows. In effect our inner sandbox not only isolates the system from the native module, but also helps to isolate the native module from the operating system.

2.2. Runtime Facilities

The sandboxes prevent unwanted side effects, but some side effects are often necessary to make a native module useful. For interprocess communications, Native Client provides a reliable datagram abstraction, the "Inter-Module Communications" service or IMC. The IMC allows trusted and untrusted modules to send/receive datagrams consisting of untyped byte arrays along with optional "NaCl Resource Descriptors" to facilitate sharing of files, shared memory objects, communication channels, etc., across process boundaries. The IMC can be used by trusted or untrusted modules, and is the basis for two higher-level abstractions. The first of these, the Simple Remote Procedure Call (SRPC) facility, provides convenient syntax for defining and using subroutines across NaCl module boundaries, including calls to NaCl code from JavaScript in the browser. The second, NPAPI, provides a familiar interface to interact with browser state, including opening URLs and accessing the DOM, that conforms to existing constraints for content safety. Either of these mechanisms can be used for general interaction with conventional browser content, including content modifications, handling mouse and keyboard activity, and fetching additional site content; substantially all the resources commonly available to JavaScript.

As indicated above, the service runtime is responsible for providing the container through which NaCl modules

interact with each other and the browser. The service runtime provides a set of system services commonly associated with an application programming environment. It provides sysbrk() and mmap() system calls, primitives to support malloc()/free() interface or other memory allocation abstractions. It provides a subset of the POSIX threads interface, with some NaCl extensions, for thread creation and destruction, condition variables, mutexes, semaphores, and thread-local storage. Our thread support is sufficiently complete to allow a port of Intel's Thread Building Blocks [51] to Native Client. The service runtime also provides the common POSIX file I/O interface, used for operations on communications channels as well as web-based read-only content. As the name space of the local file system is not accessible to these interfaces, local side effects are not possible.

To prevent unintended network access, network system calls such as connect() and accept() are simply omitted. NaCl modules can access the network via JavaScript in the browser. This access is subject to the same constraints that apply to other JavaScript access, with no net effect on network security.

The NaCl development environment is largely based on Linux open source systems and will be familiar to most Linux and Unix developers. We have found that porting existing Linux libraries is generally straightforward, with large libraries often requiring no source changes.

2.3. Attack Surface

Overall, we recognize the following as the system components that a would-be attacker might attempt to exploit:

- inner sandbox: binary validation
- outer sandbox: OS system-call interception
- service runtime binary module loader
- service runtime trampoline interfaces
- IMC communications interface
- NPAPI interface

In addition to the inner and outer sandbox, the system design also incorporates CPU and NaCl module black-lists. These mechanisms will allow us to incorporate layers of protection based on our confidence in the robustness of the various components and our understanding of how to achieve the best balance between performance, flexibility and security.

In the next section we hope to demonstrate that secure implementations of these facilities are possible and that the specific choices made in our own implementation work are sound.

3. Native Client Implementation

3.1. Inner Sandbox

In this section we explain how NaCl implements software fault isolation. The design is limited to explicit control flow, expressed with calls and jumps in machine code. Other types of control flow (e.g. exceptions) are managed in the NaCl service runtime, external to the untrusted code, as described with the NaCl runtime implementation below.

C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

Table 1: Constraints for NaCl binaries.

Our inner sandbox uses a set of rules for reliable disassembly, a modified compilation tool chain that observes these rules, and a static analyzer that confirms that the rules have been followed. This design allows for a small trusted code base (TCB) [61], with the compilation tools outside the TCB, and a validator that is small enough to permit thorough review and testing. Our validator implementation requires less than 600 C statements (semicolons), including an x86 decoder and `cpuid` decoding. This compiles into about 6000 bytes of executable code (Linux optimized build) of which about 900 bytes are the `cpuid` implementation, 1700 bytes the decoder, and 3400 bytes the validator logic.

To eliminate side effects the validator must address four sub-problems:

- Data integrity: no loads or stores outside of data sandbox
- Reliable disassembly
- No unsafe instructions
- Control flow integrity

To solve these problems, NaCl builds on previous work on CISC fault isolation. Our system combines 80386 segmented memory [14] with previous techniques for CISC software fault isolation [40]. We use 80386 segments to constrain data references to a contiguous subrange of the virtual 32-bit address space. This allows us to effectively implement a data sandbox without requiring sandboxing of load and store instructions. VX32 [20], [21] implements its data sandbox in a similar fashion. Note that NaCl modules are 32-bit x86 executables. The more recent 64-bit executable model is not supported.

Table 1 lists the constraints Native Client requires of untrusted binaries. Together, constraints C1 and C6 make disassembly reliable. With reliable disassembly as a tool, detection of unsafe instructions is straightforward. A partial list of opcodes disallowed by Native Client includes:

- `syscall` and `int`. Untrusted code cannot invoke the

operating system directly.

- all instructions that modify x86 segment state, including `lds`, far calls, etc.
- `ret`. Returns are implemented with a sandboxing sequence that ends with an indirect jump.

Apart from facilitating control sandboxing, excluding `ret` also prevents a vulnerability due to a race condition if the return address were checked on the stack. A similar argument requires that we disallow memory addressing modes on indirect `jmp` and `call` instructions. Native Client does allow the `hlt` instruction. It should never be executed by a correct instruction stream and will cause the module to be terminated immediately. As a matter of hygiene, we disallow all other privileged/ring-0 instructions, as they are never required in a correct user-mode instruction stream. We also constrain x86 prefix usage to only allow known useful instructions. Empirically we have found that this eliminates certain denial-of-service vulnerabilities related to CPU errata.

The fourth problem is control flow integrity, insuring that all control transfers in the program text target an instruction identified during disassembly. For each direct branch, we statically compute the target and confirm it is a valid instruction as per constraint C6. Our technique for indirect branches combines 80386 segmented memory with a simplified sandboxing sequence. As per constraint C2 and C4, we use the `CS` segment to constrain executable text to a zero-based address range, sized to a multiple of 4K bytes. With the text range constrained by segmented memory, a simple constant mask is adequate to ensure that the target of an indirect branch is aligned mod 32, as per constraints C3 and C5:

```
and         %eax, 0xffffffe0
jmp         *%eax
```

We will refer to this special two instruction sequence as a `nacljmp`. Encoded as a three-byte `and` and a two-byte `jmp` it compares favorably to previous implementations of CISC sandboxing [40], [41], [56]. Without segmented memory or zero-based text, sandboxed control flow typically requires two six-byte instructions (an `and` and an `or`) for a total of fourteen bytes.

Considering the pseudo-code in Figure 3, we next assert and then prove the correctness of our design for control-flow integrity. Assuming the text in question was validated without errors, let S be the set of instructions addresses from the list StartAddr.

Theorem: S contains all addresses that can be reached from an instruction with address in S.

Proof: By contradiction. Suppose an address IP not in S is reached during execution from a predecessor instruction A with address in S. Because execution is constrained by x86 segmentation, IP must trivially be in [0:TextLimit). So IP can only be reached in one of three ways.

```
// TextLimit = the upper text address limit
// Block(IP) = 32-byte block containing IP
// StartAddr = list of inst start addresses
// JumpTargets = set of valid jump targets

// Part 1: Build StartAddr and JumpTargets
IP = 0; icount = 0; JumpTargets = { }
while IP <= TextLimit:
  if inst_is_disallowed(IP):
    error "Disallowed instruction seen"
  StartAddr[icount++] = IP
  if inst_overlaps_block_size(IP):
    error "Block alignment failure"
  if inst_is_indirect_jump_or_call(IP):
    if !is_2_inst_nacl_jmp_idiom(IP) or
      icount < 2 or
      Block(StartAddr[icount-2]) != Block(IP):
      error "Bad indirect control transfer"
  else
    // Note that indirect jmps are inside
    // a pseudo-inst and bad jump targets
    JumpTargets = JumpTargets + { IP }
  // Proceed to the fall-through address
  IP += InstLength(IP)

// Part 2: Detect invalid direct transfers
for I = 0 to length(StartAddr)-1:
  IP = StartAddr[I]
  if inst_is_direct_jump_or_call(IP):
    T = direct_jump_target(IP)
    if not(T in [0:TextLimit))
       or not(T in JumpTargets):
      error "call/jmp to invalid address"
```

Figure 3: Pseudo-code for the NaCl validator.

case 1: IP is reached by falling through from A. This implies that IP is InstAddr(A) + InstLength(A). But this address would have been in S from part 1 of the construction. Contradiction.

case 2: IP is reached by a direct jump or call from an instruction A in S. Then IP must be in JumpTargets, a condition checked by part 2 of the construction. Observe that JumpTargets is a subset of S, from part 1 of the construction. Therefore IP must be in S. Contradiction.

case 3: IP is reached by an indirect transfer from an instruction at A in S. Since the instruction at A is an indirect call or jump, any execution of A always immediately follows the execution of an `and`. After the `and` the computed address is aligned 0 mod 32. Since no instruction can straddle a 0 mod 32 boundary, every 0 mod 32 address in [0, TextLimit) must be in S. Hence IP is in S. Contradiction.

Hence any instruction reached from an instruction in S is also in S. ■

Note that this analysis covers explicit, synchronous control flow only. Exceptions are discussed in Section 3.2.

If the validator were excessively slow it might discourage people from using the system. We find our validator can check code at approximately 30MB/second (35.7 MB in 1.2

seconds, measured on a MacBook Pro with MacOS 10.5, 2.4GHz Core 2 Duo CPU, warm file-system cache). At this speed, the compute time for validation will typically be very small compared to download time, and so is not a performance issue.

We believe this inner sandbox needs to be extremely robust. We have tested it for decoding defects using random instruction generation as well as exhaustive enumeration of valid x86 instructions. We also have used "fuzzing" tests to randomly modify test executables. Initially these tests exposed critical implementation defects, although as testing continues no defects have been found in the recent past. We have also tested on various x86 microprocessor implementations, concerned that processor errata might lead to exploitable defects [31], [38]. We did find evidence of CPU defects that lead to a system "hang" requiring a power-cycle to revive the machine. This occurred with an earlier version of the validator that allowed relatively unconstrained use of x86 prefix bytes, and since constraining it to only allow known useful prefixes, we have not been able to reproduce such problems.

3.2. Exceptions

Hardware exceptions (segmentation faults, floating point exceptions) and external interrupts are not allowed, due in part to distinct and incompatible exception models in Linux, MacOS and Windows. Both Linux and Windows rely on the x86 stack via `%esp` for delivery of these events. Regrettably, since NaCl modifies the `%ss` segment register, the stack appears to be invalid to the operating system, such that it cannot deliver the event and the corresponding process is immediately terminated. The use of x86 segmentation for data sandboxing effectively precludes recovery from these types of exceptions. As a consequence, NaCl untrusted modules apply a failsafe policy to exceptions. Each NaCl module runs in its own OS process, for the purpose of exception isolation. NaCl modules cannot use exception handling to recover from hardware exceptions and must be correct with respect to such error conditions or risk abrupt termination. In a way this is convenient, as there are very challenging security issues in delivering these events safely to untrusted code.

Although we cannot currently support hardware exceptions, Native Client does support C++ exceptions [57]. As these are synchronous and can be implemented entirely at user level there are no implementation issues. Windows Structured Exception Handling [44] requires non-portable operating support and is therefore not supported.

3.3. Service Runtime

The service runtime is a native executable invoked by an NPAPI plugin that also supports interaction between the service runtime and the browser. It supports a variety of web browsers on Windows, MacOS and Linux. It implements the dynamic enforcement that maintains the integrity of the inner sandbox and provides resource abstractions to isolate the NaCl application from host resources and operating system interface. It contains trusted code and data that, while sharing a process with the contained NaCl module, are accessible only through a controlled interface. The service runtime prevents untrusted code from inappropriate memory accesses through a combination of x86 memory segment and page protection.

Platform	"null" Service Runtime call time
Linux, Ubuntu 6.06 Intel™ Core™ 2 6600 2.4 GHz	156
Mac OSX 10.5 Intel™ Xeon™ E5462 2.8 GHz	148
Windows XP Intel™ Core™ 2 Q6600 2.4 GHz	123

Table 2: Service runtime context switch overhead. The runtimes are measured in nanoseconds. They are obtained by averaging the measurements of 10 runs of a NaCl module which measured the time required to perform 10,000,000 "null" service runtime calls.

When a NaCl module is loaded, it is placed in a segment-isolated 256MB region within the service runtime's address space. The first 64 KB of the NaCl module's address space (NaCl "user" address space) is reserved for initialization by the service runtime. The first 4 KB is read and write protected to detect NULL pointers. The remaining 60 KB contains trusted code that implements our "trampoline" call gate and "springboard" return gate. Untrusted NaCl module text is loaded immediately after this 64 KB region. The `%cs` segment is set to constrain control transfers from the zero base to the end of the NaCl module text. The other segment registers are set to constrain data accesses to the 256 MB NaCl module address space.

Because it originates from and is installed by the trusted service runtime, trampoline and springboard code is allowed to contain instructions that are forbidden elsewhere in untrusted executable text. This code, patched at runtime as part of the NaCl module loading process, uses segment register manipulation instructions and the `far call` instruction to enable control transfers between the untrusted user code and the trusted service runtime code. Since every *0 mod 32* address in the first 64 KB of the NaCl user space is a potential computed control flow target, these are our entry points to a table of system-call trampolines. One of these entry points is blocked with a `hlt` instruction, so that the remaining space may be used for code that can only be invoked from the service runtime. This provides space for the springboard return gate.

Invocation of a trampoline transfers control from untrusted code to trusted code. The trampoline sequence resets `%ds` and then uses a `far call` to reset the `%cs` segment register and transfer control to trusted service handlers, reestablishing the conventional flat addressing model expected by the code in the service runtime. Once outside the NaCl user address space, it resets other segment registers such as `%fs`, `%gs`, and `%ss` to re-establish the native-code threading environment, fully disabling the inner sandbox for this thread, and loads the stack register `%esp` with the location of a trusted stack for use by the service runtime. Note that the per-thread trusted stack resides outside the untrusted address space, to protect it from attack by other threads in the untrusted NaCl module.

Just as trampolines permit crossing from untrusted to trusted code, the springboard enables crossing in the other direction. The springboard is used by the trusted runtime

- to transfer control to an arbitrary untrusted address,
- to start a new POSIX-style thread, and
- to start the main thread.

Alignment ensures that the springboard cannot be invoked directly by untrusted code. The ability to jump to an arbitrary untrusted address is used in returning from a service call. The return from a trampoline call requires popping an unused trampoline return addresses from the top of the stack, restoring the segment registers, and finally aligning and jumping to the return address in the NaCl module.

Table 2 shows the overhead of a "null" system call. The Linux overhead of 156 ns is slightly higher than that of the Linux 2.6 getpid syscall time, on the same hardware, of 138 ns (implemented via the vsyscall table and using the `sysenter` instruction). We note that the user/kernel transfer has evolved continuously over the life of the x86 architecture. By comparison, the segment register operations and far calls used by the NaCl trampoline are somewhat less common, and may have received less consideration over the history of the x86 architecture.

3.4. Communications

The IMC is the basis of communications into and out of NaCl modules. The implementation is built around a *NaCl socket*, providing a bi-directional, reliable, in-order datagram service similar to Unix domain sockets [37]. An untrusted NaCl module receives its first NaCl socket when it is created, accessible from JavaScript via the Document-Object Model object used to create it. The JavaScript uses the socket to send messages to the NaCl module, and can also share it with other NaCl modules. The JavaScript can also choose to connect the module to other services available to it by opening and sharing NaCl sockets as NaCl descriptors. NaCl descriptors can also be used to create shared memory segments.

Number of Descriptor	Linux	OSX	Windows
1	3.3	31.5	38
2	5.3	38.6	51
3	6.6	47.9	64
4	8.2	50.9	77
5	9.7	54.1	90
6	11.1	60.0	104
7	12.6	63.7	117
8	14.2	66.2	130

Table 3: NaCl resource descriptor transfer cost. The times are in microseconds. In this test, messages carrying zero data bytes and a varying number of I/O descriptors are transferred from a client NaCl module to a server NaCl module. On OSX, a request/ack mechanism is needed as a bug workaround in the OSX implementation of sendmsg. On Windows, a DuplicateHandle() system call is required per I/O object transferred.

Using NaCl messages, Native Client's SRPC abstraction is implemented entirely in untrusted code. SRPC provides a convenient syntax for declaring procedural interfaces between JavaScript and NaCl modules, or between two NaCl modules, supporting a few basic types (int, float, char) as well as arrays in addition to NaCl descriptors. More complex types and pointers are not supported. External data representation strategies such as XDR [18] or Protocol Buffers [26] can easily be layered on top of NaCl messages or SRPC.

Our NPAPI implementation is also layered on top of the IMC and supports a subset of the common NPAPI interface. Specific requirements that shaped the current implementation are the ability read, modify and invoke properties and methods on the script objects in the browser, support for simple raster graphics, provide the createArray() method and the ability to open and use a URL like a file descriptor. The currently implemented NPAPI subset was chosen primarily for expedience, although we will likely constrain and extend it further as we improve our understanding of related security considerations and application requirements.

3.5. Developer Tools

3.5.1. Building NaCl Modules. We have modified the standard GNU tool chain, using version 4.2.2 of the gcc collection of compilers [22], [29] and version 2.18 of binutils [23] to generate NaCl-compliant binaries. We have built a reference binary from newlib[2] using the resulting tool chain, rehosted to use the NaCl trampolines to implement system services (e.g., read(), brk(), gettimeofday(), imc_sendmsg()). Native Client supports an insecure "debug" mode that allows additional file-system interaction not otherwise allowed for secure code.

We modified gcc for Native Client by changing the alignment of function entries (`-falign-functions`) to

2. See http://sourceware.org/newlib/

32 bytes and by changing the alignment of the targets branches (`-falign-jumps`) to 32 bytes. We also changed gcc to use `nacljmp` for indirect control transfers, including indirect calls and all returns. We made more significant changes to the assembler, to implement Native Client's block alignment requirements. To implement returns, the assembler ensures that call instructions always appear in the final bytes of a 32 byte block. We also modified the assembler to implement indirect control transfer sequences by expanding the `nacljmp` pseudo-instruction as a properly aligned consecutive block of bytes. To facilitate testing we added support to use a longer `nacljmp` sequence, align the text base, and use an `and` and `or` that uses relocations as masks. This permits testing applications by running them on the command line, and has been used to run the entire gcc C/C++ test suite. We also changed the linker to set the base address of the image as required by the NaCl loader (64K today).

Apart from their direct use the tool chain also serves to document by example how to modify an existing tools chain to generate NaCl modules. These changes were achieved with less than 1000 lines total to be patched in gcc and binutils, demonstrating the simplicity of porting a compiler to Native Client.

3.5.2. Profiling and Debugging. Native Client's open source release includes a simple profiling framework to capture a complete call trace with minimal performance overhead. This support is based on gcc's `-finstrument-functions` code generation option combined with the `rdtsc` timing instruction. This profiler is portable, implemented entirely as untrusted code. In our experience, optimized builds profiled in this framework have performance somewhere between `-O0` and `-O2` builds. Optionally, the application programmer can annotate the profiler output with methods similar to printf, with output appearing in the trace rather than stdout.

Native Client does not currently support interactive debugging of NaCl binary modules. Commonly we debug NaCl module source code by building with standard tools and a library that exports all the interfaces to the NaCl service runtime, allowing us to build debug and NaCl modules from identical source. Over time we hope to improve our support for interactive debugging of release NaCl binaries.

4. Experience

Unless otherwise noted, performance measurements in this section are made without the NaCl outer sandbox. Sandbox overhead depends on how much message-passing and service runtime activity the application requires. At this time we do not have realistic applications of Native Client to stress this aspect of the system.

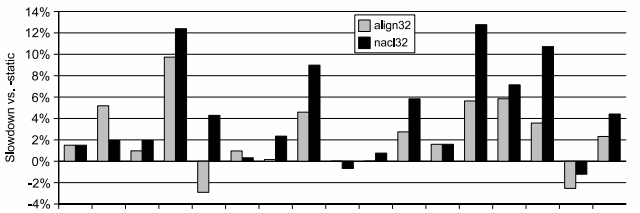

Figure 4: SPEC2000 performance. "Static" results are for statically linked binaries; "align32" results are for binaries aligned in 32-byte blocks, and "nacl32" results are for NaCl binaries.

	static	aligned	NaCl	increase
ammp	200	203	203	1.5%
art	46.3	48.7	47.2	1.9%
bzip2	103	104	104	1.9%
crafty	113	124	127	12%
eon	79.2	76.9	82.6	4.3%
equake	62.3	62.9	62.5	0.3%
gap	63.9	64.0	65.4	2.4%
gcc	52.3	54.7	57.0	9.0%
gzip	149	149	148	-0.7%
mcf	65.7	65.7	66.2	0.8%
mesa	87.4	89.8	92.5	5.8%
parser	126	128	128	1.6%
perlbmk	94.0	99.3	106	13%
twolf	154	163	165	7.1%
vortex	112	116	124	11%
vpr	90.7	88.4	89.6	-1.2%

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.

4.1. SPEC2000

A primary goal of Native Client is to deliver substantially all of the performance of native code execution. NaCl module performance is impacted by alignment constraints, extra instructions for indirect control flow transfers, and the incremental cost of NaCl communication abstractions.

We first consider the overhead of making native code side effect free. To isolate the impact of the NaCl binary constraints (Table 1), we built the SPEC2000 CPU benchmarks using the NaCl compiler, and linked to run as a standard Linux binary. The worst case for NaCl overhead is CPU bound applications, as they have the highest density of alignment and sandboxing overhead. Figure 4 and Table 4 show the overhead of NaCl compilation for a set of benchmarks from SPEC2000. The worst case performance overhead is crafty at about 12%, with other benchmarks averaging about 5% overall. Hardware performance counter measurements indicate that the largest slowdowns are due to instruction cache misses. For crafty, the instruction fetch unit is stalled during 83% of cycles for the NaCl build, compared to 49% for the default build. Gcc and vortex are also significantly impacted by instruction cache misses.

As our current alignment implementation is conservative,

	static	aligned	NaCl	increase
ammp	657	759	766	16.7%
art	469	485	485	3.3%
bzip2	492	525	526	7.0%
crafty	756	885	885	17.5%
eon	1820	2016	2017	10.8%
equake	465	475	475	2.3%
gap	1298	1836	1882	45.1%
gcc	2316	3644	3646	57.5%
gzip	492	537	537	9.2%
mcf	439	452	451	2.8%
mesa	1337	1758	1769	32.3%
parser	641	804	802	25.2%
perlbmk	1167	1752	1753	50.2%
twolf	773	937	936	21.2%
vortex	1019	1364	1351	32.6%
vpr	668	780	780	16.8%

Table 5: Code size for SPEC2000, in kilobytes.

aligning some instructions that are not indirect control flow targets, we hope to make incremental code size improvement as we refine our implementation. "NaCl" measurements are for statically linked binaries, 32-byte block alignment, and using the `nacljmp` instruction for indirect control flow transfers. To isolate the impact of these three constraints, Figure 4 also shows performance for static linking only, and for static linking and alignment. These comparisons make it clear that alignment is the main factor in cases where overhead is significant. Impact from static linking and sandboxing instruction overhead is small by comparison.

The impact of alignment is not consistent across the benchmark suite. In some cases, alignment appears to improve performance, and in others it seems to make things worse. We hypothesize that alignment of branch targets to 32-byte boundaries sometimes interacts favorably with caches, instruction prefetch buffers, and other facets of processor microarchitecture. These effects are curious but not large enough to justify further investigation. In cases where alignment makes performance worse, one possible factor is code size, as mentioned above. Table 5 shows that increases in NaCl code size due to alignment can be significant, especially in benchmarks like gcc with a large number of static call sites. Similarly, benchmarks with a large amount of control flow branching (e.g., crafty, vortex) have a higher code size growth due to branch target alignment. The incremental code size increase of sandboxing with `nacljmp` is consistently small.

Overall, the performance impact of Native Client on these benchmarks is on average less than 5%. At this level, overhead compares favorably to untrusted native execution.

4.2. Compute/Graphics Performance Tests

We implemented three simple compute+animation benchmarks to test and evaluate our CPU performance for threaded code.[3] They are:

- Earth: a ray-tracing workload, projecting a flat image of the earth onto a spinning globe
- Voronoi: a brute force Voronoi tessellation[4]
- Life: cellular automata simulation of Conway's Game of Life

Sample	Native Client	Linux Executable
Voronoi	12.4	13.9
Earth	14.4	12.6
Life	21.9	19.4

Table 6: Compute/graphics performance tests. Times are user time in seconds.

Executable	1 thread	2 threads	4 threads
Native Client	42.16	22.04	12.4
Linux Binary	46.29	24.53	13.9

Table 7: Voronoi thread performance. Times are user time in seconds.

These workloads have helped us refine and evaluate our thread implementation, in addition to providing a benchmark against standard native compilation.

We used the Linux `time` command to launch and time standalone vs. NaCl release build executables. All measurements are for a Ubuntu Dapper Drake Linux system with a 2.4GHz Intel Q6600 quad core processor. VSYNC was disabled.[5] The normal executables were built using g++ version 4.0.3, the NaCl versions with nacl-g++ version 4.2.2. All three samples were built with `-O3 -mfpmath=sse -msse -fomit-frame-pointer`.

Voronoi used four worker threads and ran for 1000 frames. Earth ran with four worker threads for 1000 frames. Life ran as a single thread, for 5000 frames. Table 6 shows the average for three consecutive runs.

Voronoi ran faster as a NaCl application than as a normal executable. The other two tests, Earth and Life, ran faster as normal executables than their Native Client counterparts. Overall these preliminary measurements suggest that, for these simple test cases, the NaCl thread implementation behaves reasonably compared to Linux. Table 7 shows a comparison of threaded performance between Native Client and a normal Linux executable, using the Voronoi demo. Comparing Native Client to Linux, performance scales comparably with increased thread count.

4.3. H.264 Decoder

We ported an internal implementation of H.264 video decoding to evaluate the difficulty of the porting effort.

3. These benchmarks will be included in our open source distribution.

4. See http://en.wikipedia.org/wiki/Voronoi

5. It is important to disable VSYNC when benchmarking rendering applications. If VSYNC is enabled, the application's rendering thread may be put to sleep until the next vertical sync occurs on the display.

The original application converted H.264 video into a raw file format, implemented in about 11K lines of C for the standard GCC environment on Linux. We modified it to play video. The port required about twenty lines of additional C code, more than half of which was error checking code. Apart from rewriting the Makefile, no other modifications were required. This experience is consistent with our general experience with Native Client; legacy Linux libraries that don't inherently require network and disk generally port with minimal effort. Performance of the original and NaCl versions were comparable and limited by video frame-rate.

4.4. Bullet

Bullet [8] is an open source physics simulation system. It has accuracy and modeling features that make it appropriate for real-time applications like computer games. As a complex, performance sensitive legacy code base it is representative of a type of system that we would like to support with Native Client.

The effort required to build Bullet for Native Client was non-trivial but generally straightforward. We used Bullet v2.66 for our experiments which is configurable via autotools [5], allowing us specify use of the NaCl compiler. We also had to build the Jam build system [35], as it is required by the Bullet build. A few #defines also had to be adjusted to eliminate unsupported profiling system calls and other OS specific code. Overall it took a couple of hours of effort to get the library to build for Native Client.

Our performance test used the HelloWorld demo program from the Bullet source distribution, a simulation of a large number of spheres falling and colliding on a flat surface. We compared two builds using GCC v4.2.2 capable of generating NaCl compliant binaries. Measuring 100,000 iterations, we observed 36.5 seconds for the baseline build (-static) vs. 32-byte aligned blocks (as required by Native Client) at 36.1 seconds, or about a 1% speedup for alignment. Incorporating the additional opcode constraints required by Native Client results in runtime of 37.3 seconds, or about a 2% slowdown overall. These numbers were obtained using a two processor dual-core Opteron 8214 with 8GB of memory.

4.5. Quake

We profiled sdlquake-1.0.9 (from www.libsdl.org) using the built-in "timedemo demo1" command. Quake was run at 640x480 resolution on a Ubuntu Dapper Drake Linux box with a 2.4GHz Intel Q6600 quad core CPU. The video system's vertical sync (VSYNC) was disabled. The Linux executable was built using gcc version 4.0.3, and the Native Client version with nacl-gcc version 4.2.2, both with `-O2` optimization.

With Quake, the differences between Native Client and the normal executable are, for practical purposes, indistinguishable. See Table 8 for the comparison. We observed very little non-determinism between runs. The test plays the same sequence of events regardless of frame rate. Slight variances in frame rate can still occur due to the OS thread scheduler and pressure applied to the shared caches from other processes. Although Quake uses software rendering, the performance of the final bitmap transfer to the user's desktop may depend on how busy the video device is.

Run #	Native Client	Linux Executable
1	143.2	142.9
2	143.6	143.4
3	144.2	143.5
Average	143.7	143.3

Table 8: Quake performance comparison. Numbers are in frames per second.

5. Discussion

As described above, Native Client has inner and outer sandboxes, redundant barriers to protect native operating system interfaces. Additional measures such as a CPU blacklist and NaCl module blacklist will also be deployed, and we may deploy whitelists if we determine they are needed to secure the system. We have also considered more elaborate measures, although as they are speculative and unimplemented we don't describe them here. We see public discussion and open feedback as critical to hardening this technology, and informing our decisions about what security mechanisms to include in the system.

We expect Native Client to be well suited to simple, computationally intensive extensions for web applications, specifically in domains such as physical simulation, language processing, and high-performance graphics rendering. Over time, if we can provide convenient DOM access, we hope to enable web-based applications that run primarily in native code, with a thin JavaScript wrapper. There are also applications of this technology outside of the browser; these are beyond our current focus.

We have developed and tested Native Client on Ubuntu Linux, MacOS and Microsoft Windows XP. Overall we are satisfied with the interaction of Native Client with these operating systems. That being said, there are a few areas where operating system support might helpful. Popular operating systems generally require all threads to use a flat addressing model in order to deliver exceptions correctly. Use of segmented memory prevents these systems from interpreting the stack pointer and other essential thread state. Better segment support in the operating system might allow us to resolve this problem and allow for better hardware exception support in untrusted code. If the OS recognized a distinguished thread to receive all exceptions, that would allow Native Client to receive exceptions in a trusted thread.

Native Client would also benefit from more consistent enabling of LDT access across popular x86 operating sys-

tems. As an interesting alternative to maintaining system call access as provided by most current systems, a system call for mapping the LDT directly into user space would remove a kernel system call from the path for NaCl thread creation, relevant for modules with a large number of threads.

With respect to programming languages and language implementations, we are encouraged by our initial experience with Native Client and the GNU tool chain, and are looking at porting other compilers. We have also ported two language interpreters, Lua and awk, and are aware of efforts to port other popular interpreted languages. While it would be challenging to support JITted languages such as Java, we are hopeful that Native Client might someday allow developers to use their language of choice in the browser rather than being restricted to only JavaScript.

6. Related Work

Techniques for safely executing 3rd-party code generally fall into three categories: system request moderation, fault isolation (including virtualization), and trust with authentication.

6.1. System Request Moderation

Kernel-based mechanisms such as user-id based access control, systrace [50] and ptrace [60] are familiar facilities on Unix-like systems. Many previous projects have explored use of these mechanisms for containing untrusted code [24], [33], [34], [36], [52], most recently Android [9], [27] from Google and Xax [17] from Microsoft Research. Android uses a sandbox for running 3rd party applications. Each Android application is run as a different Linux user, and a containment system partitions system call activity into permission groups such as "Network communication" and "Your personal information". User acknowledgment of required permissions is required prior to installing a 3rd party application. User separation inherently denies potentially useful intercommunication. To provide intercommunication, Android formed a permissions model atop the Binder interprocess communication mechanism, the Intent sytem and ContentProvider data access model. [9]

Xax is perhaps the most similar work to Native Client in terms of goals, although their implementation approach is quite different, using system call interception based on ptrace on Linux and a kernel device driver on Windows. We considered such a kernel-based approach very early in our work but rejected it as impractical due to concerns about supportability. In particular we note that the Xax Windows implementation requires a kernel-mode device driver that must be updated for each supported Windows build, a scheme we imagine onerous even if implemented by the OS vendor themselves. There are known defects in ptrace containment[6] that Xax does not address. Although the Xax authors do recognize one such issue in their paper, a simple search at Mitre's Common Vulnerabilities and Exposures site[7] documents forty-one different ptrace-related issues. Because of its pure user-space inner sandbox, Native Client is less vulnerable to these difficult kernel issues. Xax is also vulnerable to denial-of-service attacks based on x86 errata that can cause a machine to hang or reboot [31], [38]. Because Native Client examines every instruction and rejects modules with instructions it finds suspect, it significantly reduces the attack surface with respect to invalid instructions, and further it includes relevant mechanism for defending against new exploits should they be found.

Because the Xax sandbox functions at the process boundary, it fails to isolate untrusted code when shared application components such as DLLs are involuntarily injected by the operating system, an issue both for security and for portability of untrusted code. In contrast, the Native Client inner sandbox creates a security sub-domain within a native operating system process. Apart from these security differences we note that Xax does not support threading, which we considered essential given the trend towards multicore CPUs.

The Linux seccomp[8] facility also constrains Linux processes at the system call interface, allowing a process to enter a mode where only _exit(), read(), and write() system calls are permitted.

6.2. Fault Isolation

Native Client applies concepts of software fault isolation and proof-carrying code that have been extensively discussed in the research literature. Our data integrity scheme is a straightforward application of segmented memory as implemented in the Intel 80386 [14]. Our current control flow integrity technique builds on the seminal work by Wahbe, Lucco, Anderson and Graham [62]. Like Wahbe et al., Native Client expresses sandboxing constraints directly in native machine instructions rather than using a virtual machine or other ISA-portable representation. Native Client extends this previous work with specific mechanisms to achieve safety for the x86 [4], [14], [32] ring-3 instruction set architecture (ISA), using several techniques first described by McCamant and Morrisett [40]. Native Client uses a static validator rather than a trusted compiler, similar to validators described for other systems [19], [40], [41], [49], applying the concept of proof-carrying code [46].

After the notion of software fault isolation was popularized by Wahbe et al., researchers described complementary and alternative systems. A few [1], [19], [40], [41],

6. http://www.linuxhq.com/kernel/v2.4/36-rc1/Documentation/ptrace.txt

7. For example, see http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ptrace

8. See linux/kernel/seccomp.c

[49], [56] work directly with x86 machine code. Others are based on intermediate program representations, such as type-safe languages [28], [45], [47], [59], abstract virtual machines [3], [20], [21], [39], or compiler intermediate representations [53]. They use a portable representation, allowing ISA portability but creating a performance obstacle that we avoid by working directly with machine code. A further advantage of expressing sandboxing directly in machine code is that it does not require a trusted compiler. This greatly reduces the size of the trusted computing base [61], and obviates the need for cryptographic signatures from the compiler. Apart from simplifying the security implementation, this has the further benefit in Native Client of opening the system to 3rd-party tool chains.

Compared to Native Client, CFI [1] provides finer-grained control flow integrity. Whereas our system only guarantees indirect control flow will target an aligned address in the text segment, CFI can restrict a specific control transfer to a fairly arbitrary subset of known targets. While this more precise control is possibly useful in some scenarios, such as insuring integrity of translation from a high-level language, it is not useful for Native Client, since we intend to permit quite arbitrary control flow, even hand-coded assembler, as long as execution remains in known text and targets are aligned. At the same time, CFI overhead is a factor of three higher on average (15% vs. 5% on SPEC2000), not surprising since its instrumentation sequences are much longer than Native Client's, both in terms of size and instruction count. XFI [19] adds data sandboxing to CFI control flow checks, with additional overhead. By contrast Native Client gets data integrity for free from x86 segments.

Other recent systems have explored mechanisms for enabling safe side effects with measured trust. NaCl resource descriptors are analogous to capabilities in systems such as EROS [55]. Singularity channels [30] serve an analogous role. DTrace [11], Systemtap [49] and XFI [19] have related mechanisms.

A number of projects have explored isolating untrusted kernel extensions. SPIN and VINO take different approaches to implementing safety. SPIN chose a type-safe language, Modula-3 [47], together with a trusted compiler tool chain, for implementing extensions. VINO, like Native Client and the original work by Wahbe et al., used software fault isolation based on sandboxing of machine instructions. Like Native Client, VINO used a modified compilation toolchain to add sandboxing instructions to x86 machine code, using C++ for implementing extensions. Unlike Native Client, VINO had no binary validator, relying on a trusted compiler. We note that a validator for VINO would be more difficult than that of Native Client, as its validator would have had to enforce data reference integrity, achieved in Native Client with 80386 segments.

The Nooks system [58] enhances operating system kernel reliability by isolating trusted kernel code from untrusted device driver modules using a transparent OS layer called the Nooks Isolation Manager (NIM). Like Native Client, NIM uses memory protection to isolate untrusted modules. As the NIM operates in the kernel, x86 segments are not available. The NIM instead uses a private page table for each extension module. To change protection domains, the NIM updates the x86 page table base address, an operation that has the side effect of flushing the x86 Translation Lookaside Buffer (TLB). In this way, NIM's use of page tables suggests an alternative to segment protection as used by Native Client. While a performance analysis of these two approaches would likely expose interesting differences, the comparison is moot on the x86 as one mechanism is available only within the kernel and the other only outside the kernel. A critical distinction between Nooks and Native Client is that Nooks is designed only to protect against unintentional bugs, not abuse. In contrast, Native Client must be resistant to attempted deliberate abuse, mandating our mechanisms for reliable x86 disassembly and control flow integrity. These mechanisms have no analog in Nooks.

There are many environments based on a virtual-machine architecture that provide safe execution and some fraction of native performance [3], [6], [7], [20], [28], [39], [53], [63]. While recognizing the excellent fault-isolation provided by these systems, we made a deliberate choice against virtualization in Native Client, as it is generally inconsistent with, or irrelevant to, our goals of OS neutrality, browser neutrality, and peak native performance.

More recently, kernel extensions have been used for operating system monitoring. DTrace [11] incorporated a VM interpreter into the Solaris kernel for safe execution, and provided a set of kernel instrumentation points and output facilities analogous to Native Client's safe side effects. Systemtap [49] provides similar capabilities to DTrace, but uses x86 native code for extensions rather than an interpreted language in a VM.

6.3. Trust with Authentication

Perhaps the most prevalent example of using native code in web content is Microsoft's ActiveX [15]. ActiveX controls rely on a trust model to provide security, with controls cryptographically signed using Microsoft's proprietary Authenticode system [43], and only permitted to run once a user has indicated they trust the publisher. This dependency on the user making prudent trust decisions is commonly exploited. ActiveX provides no guarantee that a trusted control is safe, and even when the control itself is not inherently malicious, defects in the control can be exploited, often permitting execution of arbitrary code. To mitigate this issue, Microsoft maintains a blacklist of controls deemed unsafe [42]. In contrast, Native Client is designed to prevent such exploitation, even for flawed NaCl modules.

7. Conclusions

This paper has described Native Client, a system for incorporating untrusted x86 native code into an application that runs in a web browser. In addition to creating a barrier against undesirable side effects, NaCl modules are portable both across operating systems and across web browsers, and supports performance-oriented features such as threading and vectorization instructions. We believe the NaCl inner sandbox is extremely robust; regardless we provide additional redundant mechanisms to provide defense-in-depth.

In our experience we have found porting existing Linux/gcc code to Native Client is straightforward, and that the performance penalty for the sandbox is small, particularly in the compute-bound scenarios for which the system is designed.

By describing Native Client here and making it available as open source, we hope to encourage community scrutiny and contributions. We believe this feedback together with our continued diligence will enable us to create a system that achieves a superior level of safety than previous native code web technologies.

Acknowledgments

Many people have contributed to the direction and the development of Native Client; we acknowledge a few of them here. The project was conceived based on an idea from Matt Papakipos. Jeremy Lau, Brad Nelson, John Grabowski, Kathy Walrath and Geoff Pike have made valuable contributions to the implementation and evaluation of the system. Thanks also to Danny Berlin, Chris DiBona, and Rebecca Ward. We thank Sundar Pichai and Henry Bridge for their role in shaping the project direction. We'd also like to thank Dick Sites for his thoughtful feedback on an earlier version of this paper.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. pages 93–112, 1986.

[3] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. *SIGPLAN Not.*, 31(5):127–136, 1996.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*. Advanced Micro Devices, September 2007. Publication number: 24592.

[5] Autoconf. http://www.gnu.org/software/autoconf/.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[8] Bullet physics SDK. http://www.bulletphysics.com.

[9] J. Burns. Developing secure mobile applications for android. http://isecpartners.com/files/iSEC_Securing_Android_Apps.pdf, 2008.

[10] K. Campbell, L. Gordon, M. Loeb, and L. Zhou. The economic cost of publicly announced information security breaches: empirical evidence from the stock market. *Journal of Computer Security*, 11(3):431–448, 2003.

[11] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *2004 USENIX Annual Technical Conference*, June 2004.

[12] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31:314–333, 1988.

[13] F. B. Cohen. Defense-in-depth against computer viruses. *Computers and Security*, 11(6):565–584, 1993.

[14] J. Crawford and P. Gelsinger. *Programming 80386*. Sybex Inc., 1991.

[15] A. Denning. *ActiveX Controls Inside Out*. Microsoft Press, May 1997.

[16] Directorate for Command, Control, Communications and Computer Systems, U.S. Department of Defense Joint Staff. Information assurance through defense-in-depth. Technical report, Directorate for Command, Control, Communications and Computer Systems, U.S. Department of Defense Joint Staff, February 2000.

[17] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 2008 Symposium on Operating System Design and Implementation*, December 2008.

[18] M. Eisler (editor). XDR: External data representation. Internet RFC 4506, 2006.

[19] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI '06: 7th Symposium on Operating Systems Design And Implementation*, pages 75–88, November 2006.

[20] B. Ford. VXA: A virtual architecture for durable compressed archives. In *USENIX File and Storage Technologies*, December 2005.

[21] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference*, June 2008.

[22] The GNU compiler collection. http://gcc.gnu.org.

[23] GNU binutils. http://www.gnu.org/software/binutils/.

[24] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure enviroment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[25] D. Golub, A. Dean, R. Forin, and R. Rashid. UNIX as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, 1990.

[26] Google Inc. Protocol buffers. http://code.google.com/p/protobuf/.

[27] Google Inc. Android—an open handset alliance project. http://code.google.com/android, 2007.

[28] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[29] B. Gough and R. Stallman. *An Introduction to GCC*. Network Theory, Ltd., 2004.

[30] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[31] Intel Corporation. Intel Pentium processor invalid instruction errata overview. http://support.intel.com/support/processor/pentium/ppiie/index.html.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture*. Intel Corporation, August 2007. Order Number: 253655-024US.

[33] S. Ioannidis and S. M. Bellovin. Building a secure web browser. In *USENIX Annual Technical Conference, FREENIX Track*, pages 127–134, 2001.

[34] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: A new approach to application security. In *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002.

[35] Jam 2.1 user's guide. http://javagen.com/jam/.

[36] C. Jensen and D. Hagimont. Protection wrappers: a simple and portable sandbox for untrusted applications. In *Proceedings of the 8th ACM SIGOPS European Systems Conference*, pages 104–110, 1998.

[37] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher. 4.2 BSD system manual. Technical report, Computer Systems Research Group, University of California, Berkeley, 1983.

[38] K. Kaspersky and A. Chang. Remote code execution through Intel CPU bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[39] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[40] S. McCamant and G. Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. Technical Report MIT-CSAIL-TR-2005-030, 2005.

[41] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, August 2006.

[42] Microsoft Corporation. The kill-bit faq - part 1 of 3. *Microsoft Security Vulnerability Research and Defense (Blog)*.

[43] Microsoft Corporation. Signing and checking code with Authenticode. http://msdn.microsoft.com/en-us/library/ms537364(VS.85).aspx.

[44] Microsoft Corporation. Structured exception handling. http://msdn.microsoft.com/en-us/library/ms680657(VS.85).aspx, 2008.

[45] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[46] G. Necula. Proof carrying code. In *Principles of Programming Languages*, 1997.

[47] G. Nelson (editor). *System Programming in Modula-3*. Prentice-Hall, 1991.

[48] Netscape Corporation. Gecko plugin API reference. http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference.

[49] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*, pages 49–64, July 2005.

[50] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, August 2003.

[51] J. Reinders. *Intel Thread Building Blocks*. O'Reilly & Associates, 2007.

[52] J. G. Richard West. User-level sandboxing: a safe and efficient mechanism for extensibility. Technical Report TR-2003-014, Boston University, Computer Science Department, Boston, MA, 2003.

[53] J. Richter. *CLR via C#, Second Edition*. Microsoft Press, 2006.

[54] M. Savage. Cost of computer viruses top $10 billion already this year. *ChannelWeb*, August 2001.

[55] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *Symposium on Operating System Principles*, pages 170–185, 1999.

[56] C. Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[57] B. Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, 1997.

[58] M. Swift, M. Annamalai, B. Bershad, and H. Levy. Recovering device drivers. In *6th USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[59] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 181–192, New York, NY, USA, 1996. ACM.

[60] W. Tarreau. ptrace documentation. http://www.linuxhq.com/kernel/v2.4/36-rc1/Documentation/ptrace.txt, 2007.

[61] U. S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[62] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[63] C. Waldspurger. Memory resource management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[64] *Document Object Model (DOM) Level 1 Specification*. Number REC-DOM-Level-1-19981001. World Wide Web Consortium, October 1998.