Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code

Jason Ansel
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
jansel@csail.mit.edu

Petr Marchenko
University College London
p.marchenko@cs.ucl.ac.uk

Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, Bennet Yee
Google Inc.
{ulfar, elijahtaylor, bradchen, dschuff, sehr, cbiffle, bsy}@google.com

Abstract

When dealing with dynamic, untrusted content, such as on the Web, software behavior must be sandboxed, typically through use of a language like JavaScript. However, even for such specially-designed languages, it is difficult to ensure the safety of highly-optimized, dynamic language runtimes which, for efficiency, rely on advanced techniques such as Just-In-Time (JIT) compilation, large libraries of native-code support routines, and intricate mechanisms for multi-threading and garbage collection. Each new runtime provides a new potential attack surface and this security risk raises a barrier to the adoption of new languages for creating untrusted content.

Removing this limitation, this paper introduces general mechanisms for safely and efficiently sandboxing software, such as dynamic language runtimes, that make use of advanced, low-level techniques like runtime code modification. Our *language-independent sandboxing* builds on Software-based Fault Isolation (SFI), a traditionally static technique. We provide a more flexible form of SFI by adding new constraints and mechanisms that allow safety to be guaranteed despite runtime code modifications.

We have added our extensions to both the x86-32 and x86-64 variants of a production-quality, SFI-based sandboxing platform; on those two architectures SFI mechanisms face different challenges. We have also ported two representative language platforms to our extended sandbox: the Mono common language runtime and the V8 JavaScript engine. In detailed evaluations, we find that sandboxing slowdown varies between different benchmarks, languages, and hardware platforms. Overheads are generally moderate and they are close to zero for some important benchmark/platform combinations.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection

General Terms Languages, Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06. . . $10.00

Keywords Sandboxing, Security, Software Fault Isolation, Self-Modifying Code, Just-In-Time Compilation

1. Introduction

The safe confinement of software behavior, or *sandboxing*, is a key requirement in many contexts. On the Web, sandboxing is especially important since Web applications are a form of highly-dynamic, untrusted software, and non-isolated Web software, such as browser plugins and ActiveX controls, have been, and remain, a leading source of security vulnerabilities [6, 17]. As a result, Web applications are mostly written in one of a handful of high-level, dynamic programming languages specifically designed for untrusted content—most commonly JavaScript [16].

Software safety is in tension with efficiency: even languages like Java and JavaScript are executed not through simple interpretation, but on top of highly-efficient runtime platforms. The safety of those language platforms depends on large amounts of trusted (possibly flawed) native code—implementing extensive libraries, as well as advanced mechanisms like runtime code generation and garbage collection [3, 33]. Thus, each such language adds new potential means of attack, as evidenced by the frequent exploits of Web-based languages [13, 55].

Fortunately, as we show in this paper, sandboxing can be *language independent*, provide strong safety guarantees, low overhead, and do this without restricting language choice or language implementation options. The entirety of dynamic software execution can be sandboxed, including the language platform, even if it uses just-in-time compilation, runtime code modification, or large bodies of legacy code. Despite being comprehensive, such sandboxing need induce only moderate slowdowns. As shown by our experimental results, in many important cases sandboxing may incur no overhead. Such language-independent sandboxing promises more technology options for untrusted content development—in particular, on the Web.

Our language-independent sandboxing is not based on hardware protection domains, such as the common process abstraction. These offer neither high assurance nor universal applicability, particularly in cases where a fine granularity of protection domains is desired. Their use is intricate and error-prone, leading to non-portable implementations and partial protection [4, 20, 54]. Also, the performance impact of hardware context switching can severely limit applicability [19].

Instead, our work is based on Software-based Fault Isolation

(SFI) [53], which provides high-assurance safety guarantees and is largely independent of the operating system and other system-level details. SFI relies on machine code verification through static analysis and, to date, has not been applicable to sandboxing software that modify machine code at runtime. Removing this obstacle, we present extensions to SFI techniques that allow efficient, safe sandboxing of mechanisms such as just-in-time code generation and runtime code modification. Key to our extensions are new safety constraints on the structure of machine code that apply, inductively, even across code modification.

We have implemented our extensions in the context of the Native Client open-source project: a production-quality SFI-based sandbox [44, 56]. Our extended sandbox provides efficient ways to add, modify, and delete code. It transparently allows safe reuse of code memory without race conditions and without suspending threads, even on hardware-threaded, concurrent systems. Our extensions add little overhead, partly because our new safety constraints are not onerous, but also because they are verified only when code is modified, not when code is used.

We have ported two popular, representative JIT-compiled language systems to our sandbox: the Mono CLR runtime [41], which supports C#, and the V8 JavaScript engine [27]. V8 motivated our work. Its speed compared to previous JavaScript engines clearly demonstrates the advantages of runtime code generation over even highly-optimized interpretation. Our design was further influenced by V8's use of code rewriting for inline *code caches*—and the 12x slowdown that we saw V8 incur when we disabled its code cache mechanisms.

Porting Mono and V8 consisted primarily of changing them to output verifiably-safe code, and inductively maintaining safety across modifications to that code. We targeted two commodity hardware platforms: the x86-32 and x86-64 instruction set architectures (ISAs) [25, 28]. These two ISAs have significantly different characteristics for the purposes of Native Client sandboxing. On x86-32, hardware segments allow low runtime overheads and simplified implementation of our extensions. On x86-64, extensive use is made of inline code that performs runtime sandboxing, and this both reduces performance and complicates porting efforts.

We found that the Mono and V8 platforms, and their x86-32 and x86-64 variants, spanned a wide range in terms of the porting effort required and the sandboxing slowdown incurred. At one end, Mono-32, porting effort was low and the measured overhead is near negligible. At the other extreme, V8-64, porting took a few weeks and sandboxing slowdown is between 51% and 60% on average, but 196% in one benchmark (due to a porting shortcut, as explained in Section 4). Notably, in all cases sandboxing overhead was much lower than what might be expected of a highly-efficient interpreter, where even favorable workloads incur factors of overhead [21, 43].

1.1 Contributions

This paper makes the following contributions:

- A set of locally-verifiable machine code constraints that allow safe, SFI-based sandboxing even of runtime code modification.
- A technique for safe runtime modification of machine code on commodity hardware, despite the simultaneous execution of that code on other processors by untrusted user threads.
- A technique for the safe deletion of code memory, and its reuse for new instructions, despite potentially concurrent execution of untrusted user threads in that code memory.
- Implementation of our language-independent sandbox for x86-32 and x86-64, as well as ports of the V8 and Mono runtimes, allowing software to embed languages like JavaScript and C# as untrusted components.
- Experience showing that it can be straightforward to port modern, advanced language runtimes to execute within language-independent sandboxing.
- Experimental results showing that, while overhead varies between platforms, sandboxing overhead is generally modest, and that a sandboxed advanced language runtime incurs far less overhead than the published slowdown of using interpretation.
- A somewhat surprising optimization technique relying on careful selection of even unexecuted NOP instructions.

To create safe, low-overhead runtime code-modification mechanisms that were easy to use, we had to simultaneously satisfy a great number of constraints. These include the constraints imposed by the NaCl SFI sandbox, the concrete properties of the NaCl runtime system and the operating systems and hardware CPUs it runs upon, as well as the practical characteristics of language platform implementations. While the primitives introduced are straightforward, the intricacy of the underlying mechanisms—and the lack of similar mechanisms in previous SFI systems—is evidence of the difficulty in creating their implementation.

1.2 Motivating Application

Our work has been motivated by the tantalizing possibility of language-independent software development of safe, untrusted Web applications. The foundation for our implementation, Native Client has been integrated with the Chrome Web browser, where it provides access to JavaScript Web interfaces [44, 56]. Our extended sandboxing allows safe execution of ported language runtimes, and their native libraries, even when they make use of advanced code modification techniques for efficiency. Contrast this with today's reality, where rich, interactive Web clients must be written in JavaScript, either from scratch, or via source-to-source translation, with resulting difficulties and inefficiencies [9, 26]. (The alternatives are neither practical nor appealing: asking users to install Web browser plugins for new languages, or using partially-deployed languages like Java.) Adding our extended sandbox to a Web browser could enable writing a Web page in any language, as long as that language runtime is downloaded as part of the page.

1.3 Outline

The remainder of this paper is organized as follows. Section 2 provides background on SFI and the Native Client sandbox. Section 3 covers the design and implementation of our sandboxing extensions. Section 4 describes our experience porting language runtimes to our extended sandbox. Section 5 shows experimental results, examines the sources of overhead, and explains optimization via NOP instruction selection. Finally, Sections 6 and 7 describe related work and draw conclusions.

2. Background

This work builds on the Native Client open-source project [44, 56] commonly abbreviated "NaCl" when used as an adjective. The NaCl sandbox uses Software-based Fault Isolation (SFI) [32, 53] to restrict what instructions can be executed, in what sequence, and constrain the memory addresses used by instructions. SFI provides high-assurance safety guarantees by combining static analysis with *software guards*: short, inline instruction sequences that perform runtime safety checks or sandboxing operations. In this, SFI is similar to language-runtime mechanisms ranging from dynamic array-bounds checks [39] to concurrency controls such as software-transactional memory [48].

SFI allows the creation of *statically verifiable* machine code: code that, through static analysis, can be independently verified to permit only constrained executions, irrespective of how it was

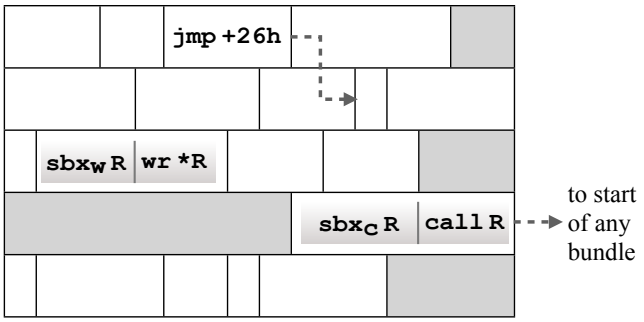


Figure 1. Overview of abstract NaCl machine code, with each fixed-width (32 byte) row denoting a NaCl instruction bundle starting at an aligned address. Vertical lines show instruction boundaries, while shaded regions show NOP padding; dashed arrows indicate permitted direct and indirect control flow. Two NaCl pseudo instructions are shown as gradient-shaded boxes.

created (e.g., through compilation from a high-level language). As with Java bytecode verification [30], SFI establishes safety primarily by checking simple, local properties of code, where those properties, in aggregate, imply global execution invariants. To ensure that safety holds for all possible execution paths, SFI restricts both direct and indirect control transfers, implementing a form of control-flow integrity [1]. SFI must, in particular, prevent control flow from circumventing inline software guards—as this would allow unconstrained execution, including control transfers to disallowed instructions.

On top of its SFI sandboxing, Native Client provides a production-quality, fully-functional platform for the safe execution of untrusted, multi-threaded user-level machine code. Native Client is designed to minimize the overhead of safety: even untrusted, hand-optimized media codecs can be executed at near full speed, since SFI allows safe use of hand-written assembly and model-specific instructions, such as the MMX and SSE extensions on x86. The NaCl platform also provides a programming model for high-level languages using ILP32 (32-bit Integers, Longs and Pointers) primitive types and a portable subset of POSIX-like system calls.

Native Client implements a relatively simple variant of SFI, fundamentally based on *reliable disassembly*: the use of alignment and other machine-code constraints to ensure runtime-reachable instructions can be statically identified. Unlike in more complex SFI implementations [20], each NaCl software guard is combined into a *pseudo instruction* with an adjacent, potentially-unsafe instruction, making each guard constrain the behavior of only a single instruction. As in PittSFIeld [32], NaCl machine code is structured as aligned *instruction bundles* of fixed size (32 bytes). NOP padding is used to ensure that no instruction overlaps the boundary of such an aligned instruction bundle. Control-flow instructions must target valid code, and all computed control transfers must target the start of a instruction bundle. Indirect jumps are allowed only via a pseudo-instruction that aligns the target address, and NOP padding is used to ensure CALL instructions appear only at the end of instruction bundles, so that return addresses are always aligned. In addition, NaCl execution is constrained to a single, contiguous code region, with access to a restricted set of NaCl platform support routines, and memory access is limited to a single, contiguous region of untrusted data memory, which also contains the stacks for threads.

Figure 1 gives an overview of the structure of NaCl machine code for an abstract ISA, showing the details of one NaCl *code region*, comprising five NaCl instruction bundles. Native Client independently verifies the safety of each such code region, one at a time; in our extended sandboxing, this is the region being added or modified. For the first instruction, a direct jump forward by 38 bytes (or 0x26), NaCl verification establishes that the target is a valid instruction boundary within the code region. The two remaining instructions are NaCl pseudo instructions that use the register `R` to write to memory at address `R` and call the function at address `R`. For these pseudo instructions, NaCl verification establishes the correctness of the software guards used to sandbox the register `R`. For CALL instructions, both their target address and the pushed return address must be the start of an aligned instruction bundle. NaCl verification also imposes many structural constraints on machine code; for instance, direct jumps may not target the inner boundaries of pseudo instructions. NOP padding may need to be used to ensure proper alignment and, as clearly shown in Figure 1, large amounts of such padding may be required, especially around CALL instructions.

Native Client additionally prohibits use of the x86 RET opcode. Returns should be implemented with a POP/JMP sequence. Checking the return address in a register avoids a potential time-of-check/time-of-use race if it were checked on the stack.

Native Client supports three architectures, x86-32, x86-64, and ARM. Like other SFI systems, Native Client also makes use of operating-system and hardware support where appropriate, e.g., to ensure that data is not executable, and that code is not writable and is loaded at the correct address. In particular, on the 32-bit x86 platform, Native Client relies on hardware segments, not software guards, to constrain reads and writes to data memory, as well as code execution. However, the baseline overhead of Native Client will vary with each platform: on ARM and 64-bit x86, bounded segments aren't available, and software guards must be used for each computed access to memory.

Our work in this paper extends SFI techniques to be applicable to runtime code generation and modification through the addition of new, inductively-maintained safety constraints on the structure of machine code. Our motivation is primarily performance: compared to a JIT-compiled runtime, even the most highly-optimized interpreter on its most favorable workload is likely to be twice as slow [21, 43]. Optimized JIT compilation also makes use of runtime code modification—for instance, to allow machine-code constants to be modified to point to new code emitted by the JIT compiler, or to make more extensive updates, such as to the V8 code caches. Instead of modifying immediate constants in machine code, JIT compilers can emit code based on additional levels of indirection; however, such indirection comes at significant performance cost, by pulling in additional cache lines and increasing the rate of memory accesses. Furthermore, language runtimes often make use of runtime code generation or modification for various purposes other than performance, ranging from debugging and profiling, through various instrumentation, to specific mechanisms such as runtime barriers [2, 10, 23, 29, 38]. Even though this paper focuses on JIT compilation, flexibility is another motivation for our work.

3. Core Mechanisms: Design and Implementation

This section describes the design and implementation of our extensions to Native Client to support runtime code modification. After outlining our extensions to the Native Client system call interface, we then describe the implementation in detail, and how it supports dynamic code creation, modification, and deletion.

3.1 Programming Interface

Table 1 lists the interfaces added to the trusted runtime of Native Client to support dynamic code manipulation. NaCl machine code comprises one or more code regions. Our extensions allow the addition and deletion of code regions, as well as the modification

```
int nacl_dyncode_create(void* target,
                        void* src,
                        size_t size);

int nacl_dyncode_modify(void* target,
                        void* src,
                        size_t size);

int nacl_dyncode_delete(void* target,
                        size_t size);
```

Table 1. The interfaces to our NaCl extensions for dynamically adding and deleting code regions, and modifying the instructions within a code region.

of code within a region. All code regions reside within the same single, contiguous address range of executable memory, whose size is set at link time. Executable memory can be modified only through the trusted interfaces, not directly from NaCl code.

As an example of using our interfaces, consider the JIT compilation of a single function. The JIT compiler, running as untrusted NaCl code, generates machine-code in a temporary buffer in data memory. To install the code, the JIT invokes `nacl_dyncode_create`, transferring control into the NaCl trusted runtime. This runtime validates the code and installs it in executable memory, described in detail below.

After creation, the JIT may wish to modify the code, for example, to update a pointer address stored in an immediate pointer. The trusted `nacl_dyncode_modify` interface supports these modifications.

If at some point the code is no longer needed, it can be deleted and the memory reused using `nacl_dyncode_delete`. As Native Client supports concurrent untrusted user threads, the runtime must verify that no threads are executing in the code before it can be safely reused. As `nacl_dyncode_delete` does not block, it will return an error code if it cannot verify that all threads have left the deleted region. Section 3.5 describes the implementation in detail.

3.2 Dynamic Code Creation

Supporting dynamic creation of code required only simple extensions to Native Client. Our implementation leverages existing NaCl verification outlined in Section 2. Code creation involves the following operations:

1. The target code address is verified to be in NaCl executable memory, and aligned to an instruction bundle boundary.
2. To avoid a time-of-test/time-of-use race condition, the code is copied to the private memory of the trusted NaCl runtime.
3. The code is verified using the standard NaCl validator.
4. The target address range in NaCl executable memory is checked to be unused and also reserved, in one atomic operation.
5. The code is safely copied to the target address.

Each NaCl code region is verified independently, and all control flow between code regions, whether direct or computed, must target the start of a NaCl instruction bundle. For this, NaCl verification relies on a property of HLT instructions: in Native Client, executing a HLT (the "halt" instruction) results in the immediate, permanent termination of all NaCl execution threads. To ensure safety, unused executable NaCl memory is filled with HLT instructions. For safe copying of verified code regions, the first byte of each instruction bundle is written as a HLT instruction until all other code bytes have been copied. Then the first byte of each instruction bundle is written with the intended value.

3.3 NaCl Verification of Runtime Code Modifications

Practical sandboxing of dynamic language runtimes requires our extended Native Client platform to support runtime code modification. With our design, verification overhead is proportional to the number of changed instructions. Verification requires inspection only of modified instruction bundles and of instruction bundles targeted by direct control-flow transfers from modified instructions. Table 2 lists the complete set of code safety constraints necessary for NaCl verification of dynamically-modified code. These constraints imply immutability of the instruction boundaries and the NaCl guard instructions in the modified code. They maintain the NaCl safety guarantees across code modification, since they preserve all properties established by runtime guards, as well as the structural properties of NaCl machine code.

These immutability properties restrict the set of allowed code modifications, notably on platforms with variable-length instructions like x86-32. For example, the constraints would prevent two short, adjacent instructions from being overwritten with a single larger instruction, unless the entire code region is deleted first. In practice we have not found these restrictions to be onerous, as demonstrated by our porting work described in Section 4, Immutability also helps prevent race conditions: our safety guarantees hold even in the presence of concurrent, untrusted threads that are executing the code under modification, as long as individual instructions are updated atomically.

3.4 Atomic Modification of Machine-Code Instructions

If a thread uses `nacl_dyncode_modify` to modify an instruction, then concurrent threads should execute either the old or new instructions, but no other instructions.

It is difficult to provide atomicity without a cost to efficiency, especially on multi-core or otherwise hardware-threaded systems. Runtime code modification is relatively uncommon; when used, it is usually performed in a synchronous fashion, via the operating system or a trusted language runtime. Therefore, CPU designers have had few incentives to provide reliable, simple primitives that allow code modification to appear atomic across concurrently-executing hardware threads. Indeed, in order to support mechanisms such as instruction prefetch buffers, AMD and Intel processor documentation outlines a special memory model for updates to concurrently-executing machine-code memory [25, 28].

On multicore and multiprocessor systems, a naive implementation of runtime code modification can lead to the execution of corrupted instructions. To understand the possibility of such instruction stream corruption, we ran highly-concurrent code modification experiments, using straightforward code modification. We found that corruption was indeed possible on all processors that we tested; Sundaresan et al. made similar observations in their development of a Java JIT compiler [49].

Fortunately, with careful implementation, both AMD and Intel processors can support safe "cross-modification" of hardware-threaded machine code from user mode. On Intel processors, regular atomic writes to memory can be used to reliably modify machine-code instructions that fall entirely within a 16-byte aligned region of code memory—although instructions that span two such aligned memory regions cannot be updated atomically. The same holds true for AMD processors, except that only smaller, 8-byte code memory regions are supported. We confirmed these properties through both our experiments and also through a conservative reading of the relevant documentation [25, 28].

When instructions span these code memory alignment boundaries, they can still be modified without instruction-stream corruption, through a careful implementation and use of memory synchronization barriers. We crafted such a code-modification mechanism, and conservatively chose to always apply AMD's

- NEW must satisfy all NaCl safety verification constraints, as outlined in Section 2.
- Both NEW and OLD must start at the same address, be of equal size, and lie within a single code region.
- Any direct control-transfer instructions in NEW must target valid instruction boundaries in the same code region.
- NEW and OLD must start and end at instruction boundaries, and all instruction boundaries between must be identical.
- No pseudo instructions are added or removed. NEW may not introduce new pseudo instructions. All pseudo-instructions in OLD must occur in NEW and have identical guard instructions.

Table 2. Constraints imposed on runtime code modifications by our extended NaCl sandboxing, when machine code OLD is replaced with machine code NEW.

8-byte alignment constraints, which ensures our implementation executes correctly on both AMD and Intel processors.

The pseudo code in Figure 2 shows our mechanism for safe, atomic instruction replacement. One instruction is copied at a time. When possible, we copy an entire instruction with a single 8-byte, aligned write to memory, using a "fast path" modification sequence. This fast path performs a read-modify-write update of the 8-byte, aligned block of of memory around the old instruction. This block of code memory is read into a temporary variable, and the old instruction bytes within it are replaced with those of the new instruction; then the temporary variable is written using a single aligned 8-byte store to code memory.

The above fast path mechanism cannot be used if the old instruction straddles an 8-byte alignment boundary. In this case, our implementation works as shown in Figure 2: we modify the target instruction bytes in three steps, keeping a one-byte HLT as the first byte of the instruction while the instruction is being modified. We use serialization barriers to synchronize the instruction stream and view of code memory for all hardware threads, including other cores or processors [25, 28].

Considering all possible interleavings, it is easy to see why the above code allows a series of machine-code instructions to be atomically replaced with new, equal-size instructions:

- Before the first serialization barrier: The one-byte write will be atomic; a concurrent thread will execute either the original instruction or the HLT instruction.
- Between the serialization barriers: No threads can execute the target instruction without executing the HLT instruction encoded in its first byte.
- After the second serialization barrier: As the one-byte write will be atomic, other threads will execute either the new instruction or the HLT instruction.

Our technique relies on a serialization barrier primitive. The common approach on x86 processors is to use a serializing kernel-mode instruction (such as cpuid) on all hardware threads [25, 28], requiring execution of kernel-mode code. As NaCl sandboxing is user-mode only, we require a serialization barrier that can be triggered from user-mode. Conveniently, certain system calls serialize all processors as a side-effect. We used the mprotect system call, triggering inter-processor interrupts of remote hardware threads for a "TLB shoot-down", serializing all processors.

Finally, as mentioned previously, any execution of a HLT instruction will result in complete termination of all Native Client threads. Therefore, no thread may ever attempt to execute NaCl code memory while it is being updated: doing so may cause

```
// For an instruction pair OLDI and NEWI,
// with both instructions of size N-1 bytes.
//
if (diff of (OLDI, NEWI) lies in aligned qword)
{
  // Fast path: Read the aligned, 8-byte region
  // around OLDI, then update with NEWI bytes.
  //
  atomic aligned qword write to update OLDI;
}
else
{
  // Slow path: Three-part update sequence,
  // with two memory serialization barriers.
  //
  OLDI[0] = 0xf4;         /* HLT instruction */

  serialize();            /* barrier */

  OLDI[1:N] = NEWI[1:N];

  serialize();            /* barrier */

  OLDI[0] = NEWI[0];
}
```

Figure 2. Pseudo code for instruction replacement in the presence of untrusted concurrent hardware threads.

the thread to execute a HLT. Preventing concurrent execution and modification of the same code is the responsibility of the systems using our NaCl code-modification primitives. This is not an onerous duty: language runtimes should already be taking such measures to prevent instruction-stream corruption like that we have seen in our experiments.

3.5 Dynamic Code Deletion

NaCl executable memory comprises a set of code regions, and a dynamically-generated code region can be deleted to reclaim its executable memory. Thereby, executable memory can be reused for new machine code without the nacl_dyncode_modify constraint of preserving instruction boundaries.

Safety dictates that it must not be possible for a sleeping thread to wake up and find that it is executing in the middle of an instruction, because the executable memory at the thread's instruction-pointer address has been reused for a new NaCl code region. To prevent this, we utilize a thread "wind down" mechanism: we note all running threads, mark the code region for deletion, and do not allow the executable memory to be reused until we've confirmed that no thread is executing in the code region to be deleted. To confirm this is the case, we wait for each thread to invoke the trusted Native Client runtime. Being in the trusted runtime, we know a thread is not in the deleted region, and further, that any attempt to resume execution in the deleted region, while obviously incorrect, will also be safe with respect to instruction boundaries as it will target the aligned start of an instruction bundle.

Concretely, our code deletion mechanism will:

1. Ensure that the code region was created dynamically.
2. To ensure that no new threads enter the code region, write a HLT instruction at the start of each instruction bundle. Recall that NaCl computed control flow and returns always target the start of an instruction bundle.
3. Increment the *global generation number*, and record it in the *deletion generation* for the code region to be deleted.
4. As each thread makes a call to the trusted NaCl service runtime,

	LoC total	LoC added/changed
V8-32	190526	1972 (1.04%)
V8-64	189969	5005 (2.63%)
Mono-32	386300	2469 (0.64%)
Mono-64	388123	3240 (0.83%)

Table 3. Changed lines of code for NaCl ports of V8 and Mono.

update that thread's *thread generation number*.

5. The executable memory can be reused when all threads have a generation number greater than or equal to that of the deleted code region.

If only one thread is executing, then invoking our code deletion mechanism will immediately perform all of the above. With multiple threads, the first call to `nacl_dyncode_delete` will return an error code `EAGAIN`. Re-invoking the interface with the same arguments and seeing a success return code indicates deletion is complete. The non-blocking property of the `nacl_dyncode_delete` interface allows user threads to perform useful work, while waiting for executable memory to become reusable.

4. Experience Porting Language Runtimes

This section describes our experience porting both the V8 JavaScript engine [27] and the Mono Common Language Runtime [41] to run inside our extended NaCl sandbox. Each language platform makes use of JIT compilation and also makes use of dynamic code modification techniques. After providing background on each platform, we describe our porting experiences. Because the x86-32 and x86-64 JIT compilers are substantially different on both platforms, our work comprised porting and debugging four distinct JIT implementations.

Table 3 provides data about code modifications required for our sandboxed versions of V8 and Mono. Only about 1% or less of code required changes, evidence of the relative simplicity of adapting a JIT to support our language-independent sandboxing.

4.1 V8 JavaScript Engine

The V8 JavaScript engine [27] is used in the Chromium/Google Chrome Web browser, in the Web browser on Android phones, and in Palm WebOS. In addition to being JIT compiled, V8 achieves a large performance benefit from *inline caching*, a technique first introduced in an implementation of Smalltalk 80 [15]. Inline caches store object properties such as member offsets directly in the machine code of functions. The optimized machine code of each inline cache first checks that object properties haven't changed, invalidating the cache; then, in the common case, object members are accessed directly. Inline caches in V8 rely heavily on runtime machine-code modification, making V8 a good stress test for our language-independent sandboxing.

Inline caches are vital to the efficiency of the V8 engine, a motivation for our support of runtime code modification. Disabling V8 inline caches in an otherwise unmodified V8 x86-32 system induces a 12.22x slowdown on the V8 Benchmark Suite. With our sandboxing enabled, the slowdown is 14.08x. In both cases the positive impact of inline caching is an order of magnitude higher, on all platforms, than the slowdown induced by our language-independent sandboxing.

4.2 Mono Common Language Runtime

Mono [41] is an open source, cross-platform implementation of Microsoft's .NET Framework. Mono provides both an offline compiler that translates C# to Common Intermediate Language (CIL) bytecode and a JIT compiler that reduces CIL bytecode to a number of native targets including x86-32 and x86-64. Many languages, including Microsoft Managed C++ and VB.NET have CIL bytecode compilers. Mono also supports "Ahead-Of-Time" (AOT) compilation of CIL bytecode, a useful facility when JIT compilation is either not possible or not permitted.

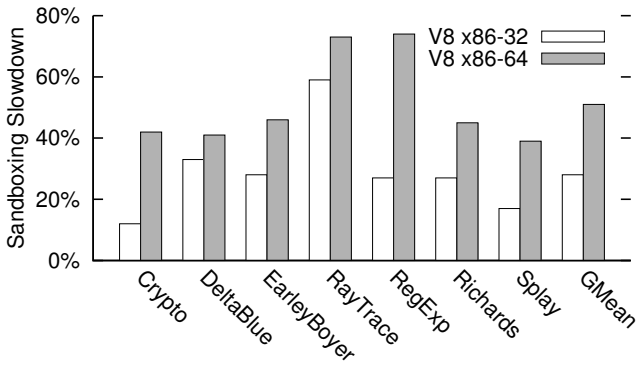
4.3 Porting Experiences

A primary task in each of our four porting efforts was modifying the JIT compiler to emit code that would satisfy NaCl code safety verification. For this, only minimal, isolated modifications were required, because NaCl verification is based on checking local machine-code properties. We modified the instruction emission phase of each JIT to satisfy the NaCl alignment constraints and to emit NaCl pseudo instructions for memory references and indirect control flow, as required.

Another task arose from a fundamental constraint imposed by the Native Client platform: only machine-code instructions—and no data—may reside in the NaCl executable memory address range. We modified each of the four JITs to allocate their code, and only their code, in NaCl code regions. For V8, this required moving relocation records and other meta-data from executable memory into appropriate data segments. We implemented similar modifications for Mono, although in a few cases we worked around the constraint by embedding data as immediate constants in a non-executed, legal instruction. Such "data instructions" allow for the creation of code that uses instruction-relative positioning for data constants and still passes NaCl code safety verification. We constructed these data instructions by prefixing a push immediate opcode to the data that needed to be embedded in the code segment; the resulting push instruction is never executed at runtime.

A number of implementation challenges were specific to x86-64. The x86-64 implementation of Native Client uses the ILP32 data model, to facilitate source code portability between x86-32 and x86-64 sandboxes. This tends to conflict with the LP64 or LLP64 data models assumed by the x86-64 implementations of V8 and Mono. In particular, the language implementations assume that a pointer and a register are the same size, which is false in the x86-64 NaCl model. Our Mono and V8 ports took different approaches to resolving these issues. For Mono we introduced an explicit distinction between the register size and pointer size. This approach supports the Mono requirement for C-style structures that are accessed both from managed and unmanaged code. For V8, pointers occupy four bytes on the heap, as per the ILP32 model, but eight bytes on the stack. While this representation accommodates the fact that x86-64 does not support four-byte push or pop operations, it also requires numerous changes to code generation, resulting in a significant, albeit straightforward, engineering effort.

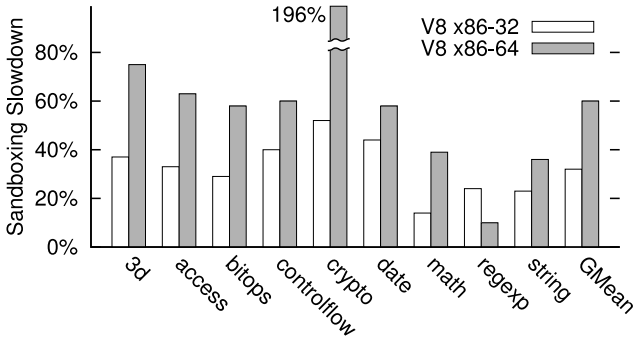
The ILP32 data model also interferes with the optimized representation of 32-bit integers in x86-64 V8. On x86-64, V8 encodes type information in the low-order bits of 64-bit registers to differentiate between immediate 32-bit integer values and pointers to boxed integer objects on the heap. For implementation expediency, we modified the x86-64 version of V8 to use the same integer representation as the x86-32 version. V8-32 uses the least-significant bit of a 32-bit word to differentiate immediate and pointers, storing integers larger than 31-bits as objects. This change also required us to reimplement some arithmetic operations. The difference in integer representations has performance implications for code that operates with large integer values, since they are stored as objects on the heap. This had a large impact on single benchmarks in the SunSpider100 benchmarks, as discussed in Section 5.



	x86-32	x86-64	x86-32	x86-64
Crypto	4380	4511	3910 (12%)	3176 (42%)
DeltaBlue	6555	5380	4921 (33%)	3825 (41%)
EarleyBoyer	20332	19370	15827 (28%)	13247 (46%)
RayTrace	9277	7979	5849 (59%)	4615 (73%)
RegExp	3367	3534	2660 (27%)	2035 (74%)
Richards	4911	4659	3864 (27%)	3217 (45%)
Splay	15305	14160	13098 (17%)	10188 (39%)
GMean	7528	7059	5868 (28%)	4683 (51%)

(a) Unsafe **(b)** Sandboxed

Figure 3. A chart of NaCl sandboxing slowdown (lower is better) for the V8 JavaScript Benchmark Suite, relative to unmodified V8. GMean is the geometric mean of the overheads of all benchmarks in the suite. Also shown is a table of the chart's underlying data: raw benchmark scores (where higher is better) for native and sandboxed execution, with sandboxing slowdown in parentheses.



	x86-32	x86-64	x86-32	x86-64
3D	3097	3434	4230 (37%)	5996 (75%)
Access	2867	3248	3815 (33%)	5296 (63%)
BitOps	2240	2047	2892 (29%)	3235 (58%)
ControlFlow	179	199	250 (40%)	318 (60%)
Crypto	1193	857	1812 (52%)	2538 (196%)
Date	2060	2236	2970 (44%)	3541 (58%)
Math	2310	2374	2639 (14%)	3295 (39%)
RegExp	1097	957	1359 (24%)	1057 (10%)
String	5147	5269	6325 (23%)	7186 (36%)
GMean	1693	1676	2241 (32%)	2689 (60%)

(a) Unsafe **(b)** Sandboxed

Figure 4. A chart of NaCl sandboxing slowdown (lower is better) for the SunSpider100 JavaScript benchmarks, relative to unmodified V8. Each bar represents a group of up to four benchmarks. GMean is the geometric mean of all benchmark overheads. Also shown is a table of the chart's underlying data: raw benchmark scores (here, lower scores are better) for native and sandboxed execution, with sandboxing slowdown in parentheses.

5. Experimental Results and Optimizations

Results in this section are from a quad-core Intel Xeon X5550 Nehalem 2.67 GHz processor, except Table 6 as noted. Our test system ran Ubuntu 10.04, kernel version 2.6.32. Performance results are an average of 10 runs. The observed standard deviation ranged from 0 to 2.32%. We summarize performance measurements using the geometric mean [22].

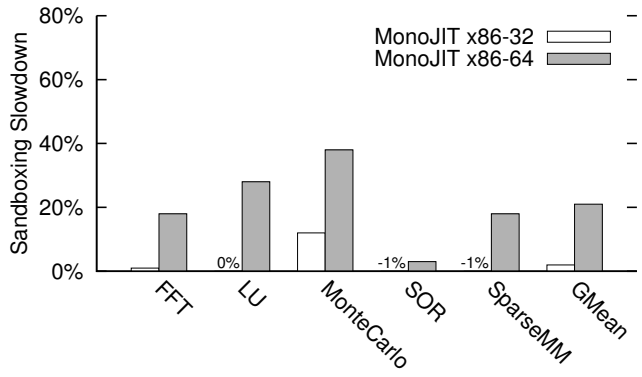
Our implementation extends the latest, up-to-date version of Native Client, which, on x86-64, has added sandboxing of indirect memory reads, as well as writes. Therefore, it is not surprising that our extended NaCl sandboxing for x86-64 has higher measured overheads than those reported in [44].

5.1 Sandboxing Overhead for V8 JavaScript and Mono C#

Figures 3 and 4 show the overheads for our sandboxed version of V8 when running the V8 Benchmark Suite [27], version 6, and the SunSpider Benchmark Suite [50], version 0.9.1. The V8 Benchmark Suite is distributed with V8 and the SunSpider Benchmark Suite is distributed with WebKit. The SunSpider suite consists of very small microbenchmarks; therefore, we created and used a variant, SunSpider100, where each benchmark is run 100 times, instead of once, to facilitate more accurate timing and to emphasize steady state execution over startup delays. We measure overhead relative to the V8 2.2.19 development version that we forked to make our changes on June 21st, 2010.

In the V8 Benchmark Suite, RayTrace showed some of the largest overheads. We investigated the cause of this slowdown on V8-64 using PIN [31], a binary instrumentation tool, and pfmon [52], a performance monitoring tool based on hardware performance counters. Using PIN, we counted executed instructions for RayTrace, both with and without the sandbox. With NaCl sandboxing, RayTrace instruction counts on V8-64 grew to a factor of 1.8x. The largest contributor is NOP padding to align instruction bundles and CALL instructions, adding 39% of the new instructions. The second largest source of additional instructions is software guards for memory references, 38% of the total new instruction. The remaining new instructions came from other software guards and the Native Client runtime. To understand the impact of branch and cache behavior, we used pfmon to measure related hardware events during the RayTrace benchmark. NaCl sandboxing requires the use of a software-guarded indirect jump, instead of a RET instruction for function returns, which we thought would likely increase branch mispredictions. Indeed, we observed a 4x increase in the branch misprediction rate. Sandboxing also increased V8-64 instruction cache pressure, with 2x change in L1 instruction cache misses and 2x change in instruction TLB misses.

The SunSpider100 suite showed similar average performance overhead as the V8 Benchmark Suite, with 32% slowdown for sandboxed V8-32 and 60% for V8-64, as compared to unmodified V8. On x86-64, the crypto benchmarks were an outlier, with over twice the overhead of any other benchmark. The crypto benchmarks are composed of 3 smaller benchmarks: crypto-aes, with 1.8x slowdown; crypto-md5, with 5.0x slowdown; and crypto-sha1, with 3.9x slowdown. The second and third benchmarks depend heavily on 32-bit integer arithmetic operations, and implement these operations through nested use of very small functions, resulting in multiple function invocations per arithmetic operation. Therefore, for these benchmarks, branch misprediction on function returns must be a large source of overhead.



	x86-32	x86-64	x86-32		x86-64	
FFT	323	332	321	(1%)	281	(18%)
LU	668	763	665	(0%)	595	(28%)
MonteCarlo	109	126	97	(12%)	91	(38%)
SOR	886	892	893	(−1%)	863	(3%)
SparseMM	450	507	452	(−1%)	429	(18%)
GMean	393	429	384	(2%)	355	(21%)

(a) Unsafe **(b)** Sandboxed

Figure 5. A chart of NaCl sandboxing slowdown (lower is better) for the SciMark C# Benchmark Suite, relative to unmodified Mono JIT. GMean is a geometric mean of overheads of all benchmarks in the suite. Also shown is a table of the chart's underlying data: raw benchmark scores (where higher is better) for native and sandboxed execution, with sandboxing slowdown in parentheses.

However, the main source of overhead results from the change in representation described in Section 4.3: our V8-64 port uses the representation of small integers from V8-32, and therefore stores integers larger than 2^{31} as heap objects. We chose this implementation for expediency, without realizing its effects. As a result of our porting shortcut, half of the 32-bit arithmetic in the crypto benchmarks will involve boxed heap objects, instead of register values, and, compounding this overhead, NaCl sandboxing on x86-64 requires use of a software guard for each indirect access to heap memory. We could have supported the 64-bit integer representation used in V8-64, since Native Client allows safe use of handwritten machine code or assembly modules; we plan to do so in a future version of our V8-64 port.

Turning to Mono, Figure 5 shows our NaCl sandbox overhead for the SciMark C# Benchmark Suite. For x86-32 the mean overhead was only 2%, while for x86-64 the overhead was 21%. Overheads were higher on x86-64 because of the additional software guards required for NaCl sandboxing of loads and stores. The largest overheads were for the MonteCarlo benchmark, whose kernel uses a tight inner loop that calls two small functions, resulting in NOP padding and branch misprediction overheads. While we haven't yet studied all original causes of this slowdown, we note it is consistent with measured performance of the x86-64 NaCl sandbox for C and C++ benchmarks [44].

5.2 Analysis of the Sources of Sandboxing Overhead

Table 4 shows the estimated breakdown of sandboxing overhead for the V8 Benchmark Suite. This table was generated by disabling NaCl sandboxing features one at a time. Since the performance impact of these different changes may not be independent, this breakdown serves only as an estimate. The largest estimated slowdown is from NOPs inserted for `CALL` and instruction bundle alignment. We estimate that fetch and execution penalties from these NOPs cause about half of the total slowdown. Return address masking is the next largest source of slowdown. NaCl sandboxing requires replacing `RET` instructions with a `POP/JMP` sequence. This interferes with the return address prediction hardware in modern processors. Other overheads (not listed) include invocations of the trusted Native Client runtime, slower dynamic code generation, copies required to modify code, and object layout changes required to separate code and code-meta-data. These overheads are small compared to those listed in Table 4.

Source of overhead	V8-32	V8-64
NOP padding to align bundles	4%	16%
NOP padding to align calls	19%	21%
Software guards for function returns	25%	22%
Software guards for indirect jumps and indirect memory accesses[1]	17%	24%
Runtime validation of code modifications	2%	5%

[1] Indirect memory accesses require software guards only on x86-64.

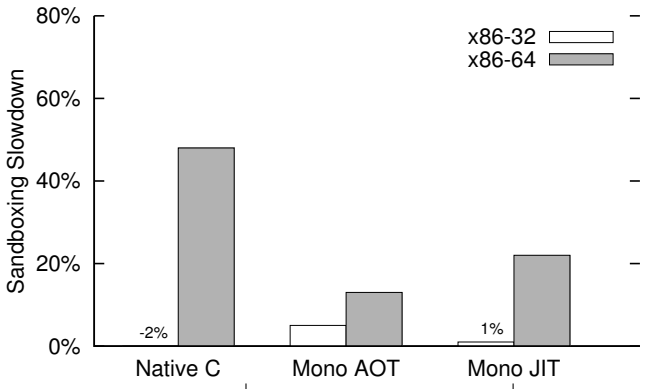
Table 4. Analysis of the sources of NaCl sandboxing slowdown, measured using the V8 Benchmark Suite (Figure 3). These measurements were generated by disabling NaCl sandboxing features one at a time. Since the performance impact of these features is not independent, these overheads are not additive.

5.3 Comparing Sandboxing and Language Overheads

Past, published measurements of SFI-based sandboxing slowdown have considered only fully-optimized, ahead-of-time compiled versions of C and C++ benchmarks like SPEC [20, 44, 46, 56]. However, the SciMark Benchmark Suite includes C as well as C# implementations of its benchmarks. This allows us to examine one data point for the relative cost of sandboxing native code vs. JIT compiled code in a modern language runtime. To facilitate this examination, we measured single-threaded SciMark execution, since its C variant runs only single threaded.

Table 5 examines the performance of three variations of SciMark, for both x86-32 and x86-64, executing both with and without NaCl sandboxing. Native is the original C implementation of SciMark; Mono AOT is the C# port, compiled ahead of time and without dynamic code modification; last, Mono JIT is the C# port, executed using JIT compilation. For x86-32, the NaCl sandboxing slowdown ranges from 0 to 5%, for all implementations. For x86-64, the slowdown ranges from 13% to 48%, and is lower for the C# implementations than for C. Despite the higher relative sandboxing overhead, the absolute performance is significantly better for the C implementation. On x86-32, our language-independent sandboxing of Mono has comparable relative overhead to that of conventionally compiled C and C++ under Native Client [56]. For NaCl sandboxed Mono-64, performance of benchmarks such as `LU` and `SparseMM` is also consistent with NaCl overheads for other benchmarks whose execution time is sensitive to path length and branch prediction effects [44].

Notably, the JIT version of the C# port is actually faster than the AOT version, especially on 64-bit platforms. The JIT compiler has knowledge of the locations of most of the data and code in the program when it compiles a method. This allows it to embed direct calls and data references to these locations in the emitted code. It can further optimize the code by backpatching existing calls to new code as it is generated. By contrast the AOT compiler uses indirection techniques such as tables of pointers, resulting in slower execution. This JIT advantage still holds in the sandboxed environment, even if offset slightly by the extra cost of NaCl safety



	x86-32	x86-64	x86-32	x86-64
Native (C)	717	868	729 (-2%)	586 (48%)
Mono AOT	383	360	365 (5%)	318 (13%)
Mono JIT	388	426	383 (1%)	350 (22%)

(a) Unsafe **(b)** Sandboxed

Table 5. Performance of SciMark C/C# Benchmark Suite compiled from native C code, with gcc 4.4.3, or with the Mono AOT and JIT compilers for C#. Benchmarks were executed natively and on the unmodified Mono CLR (Unsafe), as well as with NaCl sandboxing. Numbers show benchmark scores, in megaflops (higher is better), and also the NaCl sandboxing slowdown, as a percentage (in parentheses).

constraints.

Overall, these measurements suggest that the relative cost of language-independent NaCl sandboxing can be small compared to the overhead of a dynamic language runtime. The performance difference between the C and Mono benchmarks can be seen as the cost of using a managed-language framework. For the native, unsafe benchmark executions, the cost of executing with Mono ranges from 84% to 141% overhead, while that overhead ranges from 67% to 99% under sandboxed execution.

5.4 NOP Optimizations

We were motivated to optimize the execution of NOP padding, because it is the single largest contributor to overall NaCl sandboxing overhead. To our surprise, we found that choice of NOP padding sequences had a large impact on performance, even when the chosen NOPs were never executed at runtime. For the V8 Benchmark Suite on x86-32, different NOP padding choices reduced sandboxing overheads from a factor of 1.62x to 1.28x.

Figure 6 shows a sampling of the search space of possible NOP paddings and the resulting performance on the V8 benchmark suite. It shows three different strategies for generating NOPs:

- *Shortest NOPs* generates long sequences of the single-byte NOP instruction (`0x90`).
- *Longest NOPs* uses a greedy algorithm to generate the longest NOP instructions allowed. (Using the prefix/addressing mode variations of the `0x0f1f` multi-byte NOP opcode listed in the AMD64 Optimization Guide [24].)
- *Optimized NOPs* uses a lookup table containing optimized NOP sequences from 1 to 31 bytes in length. The table was generated automatically, in about one minute, using a microbenchmark that exhaustively searches the space of different combinations of instructions that have no effect. The individual NOP instructions are drawn from the many different inert variations of the NOP, XCHG, MOV, and LEA opcodes, using different registers and addressing modes.

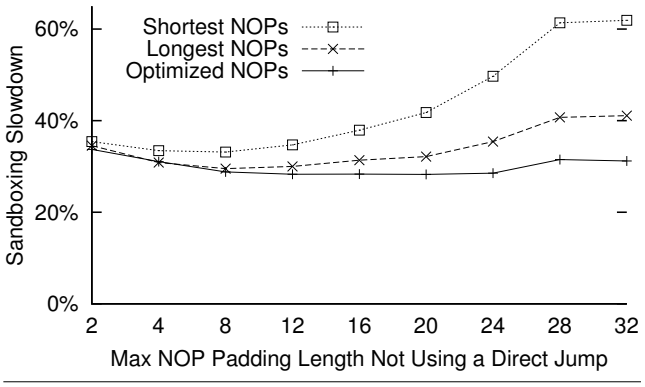


Figure 6. Sandboxing slowdown (lower is better) for different strategies of generating NOP padding for NaCl instruction alignment. The slowdown is measured from the overall GMean of the V8 Benchmark Suite executing on x86-32.

Looking at the tables generated for optimized NOPs, the trend is that variety improves performance: NOP padding sequences that vary the opcodes and registers between all the different NOPs execute much faster than NOP padding sequences that are more homogeneous, e.g., repeating the same instruction over and over. The fastest measured NOP sequences tend to contain one NOP of each possible type. This is possibly a result of more possibilities for instruction level parallelism when a mix of different NOP types is issued on modern microarchitectures.

For long NOP padding, the fastest NOP sequences contain a direct jump instruction that jumps to the end of the padding. Somewhat surprisingly, the choice of the unexecuted NOP instructions immediately after the direct jump has an effect on performance, despite being unreachable. We observed, on different architectures, between a 1.5% and a 1.8% slowdown when we changed these unreachable NOP instructions from optimized sequences of NOPs to sequences of the shortest, single-byte NOPs. We assume that those unexecuted NOP instructions impact some microarchitectural bottleneck in instruction fetching and decoding.

NOP padding performance will vary with microarchitecture, but optimized tables can be generated on each platform. Figure 6 shows the performance effects of different choices for directly jumping to the end of NOP padding sequences on the overall score of the V8 benchmark suite. The left side of the graph ("2" on the x-axis) shows V8-32 performance measured when all NOP padding sequences start with a direct jump (that jump occupies two bytes). The right side of the graph ("32" on the x-axis) shows performance when NOP padding never starts with a direct jump. Notably, starting with a direct jump is not the fastest strategy for all NOP padding lengths. For the optimized NOP selection strategy, a direct jump is faster only when NOP padding is longer than about 24 bytes.

A related optimization is to avoid execution of padding before a CALL instruction by using an explicit PUSH of the would-be return address followed by a JMP to the function entrypoint address. This optimization replaces a call of the form

```
1  ...padding...
2  call FOO
```

with the sequence

	x86-32	x86-64	x86-32	x86-64
Intel Xeon X5550 Nehalem 2.7GHz	7528	7059	5869 (28%)	4683 (51%)
Intel Core2 Quad Q6600 2.4GHz	5612	5128	4535 (24%)	3296 (56%)
AMD Phenom II X4 905E 2.5GHz	5030	4793	4026 (25%)	3390 (41%)
AMD Athlon 4450E 2.3GHz	3853	3447	2856 (35%)	2385 (45%)
AMD Opteron 8214 HE 2.2GHz	3633	3224	2701 (35%)	2226 (45%)
Intel Atom N450 1.7GHz	1395	1176	1041 (34%)	589 (100%)
	(a) Unsafe		**(b)** Sandboxed	

Table 6. Performance scores (higher is better) on different microarchitectures for the V8 Benchmark Suite running natively (Unsafe) and in our language-independent sandbox. Sandboxing slowdown is shown in in parentheses.

```
1  push retloc
2  jmp FOO
3  ...padding...
4  retloc:
```

avoiding the need to execute the padding NOPs. This optimization provides modest performance gains, of about 3 percent, and is included in the measurements of Figure 6. Similar to direct jumps over padding, this technique provides largest benefit when only applied when skipping NOP padding sequences of length 20 bytes or more.

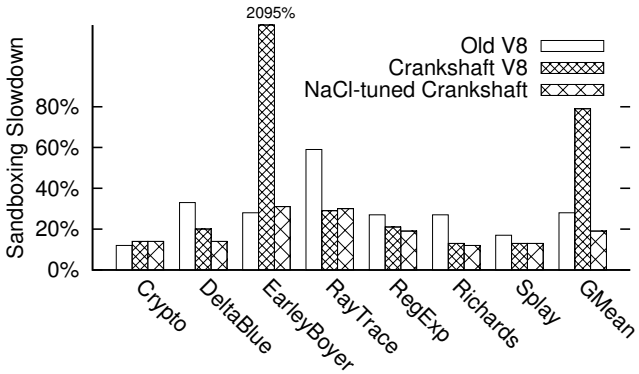
5.5 Overhead Variability between CPU Architectures

Table 6 compares the overhead of our language-independent sandboxing when running V8 benchmarks on several modern processors, implementing different microarchitectures. For V8 on x86-32 overheads are consistent, ranging from 28% to 34%. For x86-64 overheads range from 41% to 56%, and the numbers seem to suggest consistently less relative overhead on AMD processors than on the fast Intel processors. The Intel Atom is a notable outlier, with 100% sandboxing slowdown measured for V8-64. While we haven't yet fully explored the Atom's outlier performance, we note that these results match those previously reported for the x86-64 NaCl sandbox [44]. Interestingly, raw performance is better on x86-32 for both the unsafe and sandboxed versions, most likely because V8 has been more heavily optimized for x86-32.

5.6 Application to the new Crankshaft V8

Concurrently to our efforts described so far in porting a fork of the V8 code base to Native Client, the developers of V8 substantially extended the V8 JavaScript platform to use highly-dynamic, profile-driven optimizations. This "Crankshaft" version of V8 has since been released as part of the Chrome Web browser for the x86-32 architecture [34]. Crankshaft greatly improves the performance of V8, through use of SSA-based optimizations, loop-invariant code motion, better register allocation, and function inlining. Also, Crankshaft allows fast, unboxed use of all small integers less than 2^{32} (cf. the old V8, as discussed on page 6).

To gain further insights into the applicability of our work, we ported the x86-32 Crankshaft V8 (version 3.1.4) to use our code-modification primitives. We were pleased to see that our existing NaCl sandboxing port was insensitive to the higher-level changes to V8, such as the mechanisms for SSA-based optimizations. Also, the low-level x86-32 code-emission in the V8 backend remained mostly unchanged, allowing us to reuse a majority of our earlier porting work. All in all, we modified 3,483 lines in the x86-32



	Old V8	Crankshaft	Crankshaft′
Crypto	3910 (12%)	12521 (14%)	12583 (14%)
DeltaBlue	4921 (33%)	13718 (20%)	14413 (14%)
EarleyBoyer	15827 (28%)	1030 (2095%)	17294 (31%)
RayTrace	5849 (59%)	8923 (29%)	8908 (30%)
RegExp	2660 (27%)	2274 (21%)	2298 (19%)
Richards	3864 (27%)	10862 (13%)	10955 (12%)
Splay	13098 (17%)	3704 (13%)	3699 (13%)
GMean	5868 (28%)	5459 (79%)	8250 (19%)

Figure 7. A chart showing NaCl sandboxing slowdown (lower is better) for the old V8 and Crankshaft V8 platforms running the V8 JavaScript Benchmark Suite. A table of the chart's underlying raw benchmark scores (where higher is better) for NaCl sandboxed execution, with sandboxing slowdown in parentheses. Crankshaft′ denotes the NaCl-tuned version of Crankshaft.

Crankshaft V8 codebase (cf. Table 3).

We also made two small tweaks to our port, to tune Crankshaft's mechanisms to the characteristics of the NaCl sandbox. First, we increased the threshold for the function-invocation count at which Crankshaft will specialize a function to its arguments and modify the function's code. Thus, we reduced code modification rates to account for the increased cost of code modification (V8 is tuned for faster, single threaded code modification). Second, we saw that a lot of NOP padding was needed to align indirect jump targets in the middle of the code for general-purpose functions; specialized functions jump to this code if they detect that their specialization does not apply. To avoid the NOP overhead, we used a small table of direct-jump trampolines, placed after the general-purpose function code, to implement these calls.

We re-ran the benchmarking experiments on both versions of our NaCl port of the Crankshaft V8 platform. The results can be seen in Figure 7. After tuning, the absolute performance of the NaCl sandboxed Crankshaft language runtime beats that of the *unmodified, unsafe* x86-32 V8 version we used previously (see Figure 3). It is particularly gratifying that in a span of a few months, the original overhead of our language-independent sandboxing has been more than offset by the independent optimization of the language runtime.

As Figure 7 shows, the relative overhead of NaCl sandboxing is also reduced by almost a third, going from 28% to 19% for the tuned Crankshaft benchmarks. We believe this decrease in relative overhead is primarily due to Crankshaft's function inlining, which reduces both NaCl NOP padding as well as function-return branch mispredictions. The SunSpider100 benchmarks exhibited a similar absolute performance improvement, and also saw sandboxing slowdown fall from 32% to 24%.

6. Discussion and Related Work

In this paper, we have presented our techniques for safe runtime code generation and modification using the terminology of the Native Client platform upon which our implementation is based. Even so, we are confident that our techniques apply more widely. In particular, we believe that the safety constraints of Table 2 could be easily reformulated for other SFI-based platforms (e.g., XFI [20]), and still provide the immutability guarantees necessary for safe runtime code modification. Our confidence derives from the fundamental basis of our work, which lies not in NaCl-specific properties, but in the use of local code inspection to inductively establish global execution invariants—a common characteristic of many execution monitoring mechanisms.

Our experience porting JIT-based runtime platforms to our language-independent sandbox suggest that it is possible to combine SFI-based sandboxing with dynamic code generation, yielding a system that combines benefits from both techniques. Although the performance impact of sandboxing is not negligible, we believe the resulting system is still viable for a large set of practical use cases, especially considering that some of the relevant languages are still commonly implemented with interpreters. Some will argue that a JIT-based language runtime is safe enough without an SFI sandbox. We note that the verity of this statement relies on the quality of the language implementation, including the language runtime and any extensions.

In all cases, it can be expected that a fully-featured, advanced language runtime will comprise a significant amount of complex, trusted code—even when legacy libraries are *not* considered. Note the total line counts for V8 and Mono in Table 3. Furthermore, in cases where a language implementation is used as a scripting engine for a larger system, such as a Web browser, it is often desirable to sandbox the entire composed system [4, 42].

Like Native Client and other SFI-based mechanisms, our language-independent sandboxing is designed to provide high-assurance guarantees of clearly-defined safety properties, even in the face of malicious software crafted by an adaptive attacker. In comparison, many other software protection systems—such as Nooks [51], to name just one—provide a weaker form of safety by making only a best-effort attempt at containing faults.

Many software protection systems have been implemented using a combination of static analysis of machine code and inline, machine-code software guards that perform runtime checks or sandboxing operations. The first use of these techniques may have been in the late 60's, to allow the kernel execution of untrusted code in a profiling system for the SDS-940 [14]. Since then, these techniques have been used in the original work on SFI [53], subsequent SFI implementations such as MiSFIT [46], SASI x86 [18], XFI [20], and PittSFIeld [32], and in a number of other systems. Compared to Native Client, some of these systems offer finer-granularity safety guarantees for certain aspects of software execution—albeit typically at the cost of higher enforcement overhead. For example, CFI [1] can guarantee that machine-code execution follows a permitted control-flow graph, XFI [20] can enforce integrity properties for the runtime stack, and DFI [11] and its successors like BGI [12] can maintain data-flow integrity properties for even values in heap memory. However, it remains unclear if the techniques of these systems can be used to implement a production-quality, practical execution platform like Native Client, that is portable across both operating systems and hardware architectures with good performance. Furthermore, we know of no such system that provides safety guarantees in the face of runtime code generation and modification.

Programming languages and runtime software mechanisms are recognized as an effective approach to providing safety guarantees and enforcing security policies. Software isolation on the Burroughs B-5000 system depended on applications being written in the Algol high-level language [5] and a similar approach has been taken in later, experimental operating systems such as SPIN [7].

Commonly, language-based isolation is enforced through execution on top of a virtual machine, which is implemented using a trusted compiler or interpreter, and typically makes use of extensive libraries of trusted support routines based on (legacy) native code [45]. In comparison, approaches like Typed Assembly Language (TAL) [35] and Proof-Carrying Code (PCC) [37] provide guarantees about the machine code that is executed on actual hardware machines. SFI, like TAL and PCC, has the attractive characteristic of allowing independent, static safety verification of the machine code to be executed; this not only decouples the execution platform from the language toolchain, but also reduces the size of the trusted computing base [40].

A long line of research aims to preserve the semantic properties of high-level programming languages through translation to lower-level languages. Typed assembly language derives from this field and, in that context, Smith et al. have considered restricted forms of runtime code generation [47]. Recently, this field has seen great progress, starting with Xavier Leroy's work on certifying compilation from C-like languages to PowerPC machine code with full, formal proofs of semantic preservation. This work has been extended to handle incremental runtime code generation, and even self-modifying code, first by Cai et al. [8] and then, most recently, by Magnus O. Myrren [36]—in this later work formally certifying the correctness of a JIT compiler from a small bytecode language to x86 machine code. Our mechanisms also give strong safety guarantees and use techniques amenable to formal verification [32]. However, our language-independent sandboxing does not aim to preserve high-level language semantics, and the implementation of our mechanisms is unencumbered by those semantics.

7. Conclusions

Through use of language-independent, software-based fault isolation, it is possible to safely and efficiently sandbox programs that make use of self-modifying machine code. It suffices to extend traditional SFI techniques with a few new features, including new safety constraints that apply inductively on the structure of machine code, even across code modification. These new safety features are not difficult to implement in practice, e.g., as part of Native Client, an existing, production-quality SFI-based sandboxing platform. It is also rather straightforward to port V8 and Mono, two disparate, modern JIT-compiled languages, to run within a thus extended sandboxing platform. Such language-independent sandboxing holds the promise of facilitating the deployment of new language and technology options for the development of untrusted software, in particular, on the Web.

Acknowledgments

We thank Seth Abraham and Richard Winterton at Intel for their helpful discussions about Intel processor behavior in the presence of cross modifying code.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, implementations, and applications. *TISSEC*, 2009.

[2] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, 2001.

[3] John Aycock. A brief history of just-in-time. *CSUR*, 2003.

[4] A. Barth, C. Jackson, C. Reis, and Google Chrome Team. The security architecture of the chromium browser. Technical report, Stanford

University, 2008. URL http://crypto.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[5] R. S. Barton. A new approach to the functional design of a digital computer. In *IRE-AIEE-ACM (Western)*, 1961.

[6] Mark Barwinski, Cynthia Irvine, and Tim Levin. Empirical study of drive-by-download spyware. Technical report, Naval Postgraduate School, 2006.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP*, 1995.

[8] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proc. PLDI '07*, 2007.

[9] Benjamin Canou, Vincent Balat, and Emmanuel Chailloux. O'Browser: Objective Caml on browsers. In *Workshop on ML*, 2008.

[10] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATC*, 2004.

[11] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI*, 2006.

[12] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009.

[13] Blazakis D. Interpreter exploitation: Pointer inference and JIT spraying. In *Black Hat DC*, 2010.

[14] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *IFIP*, 1971.

[15] Peter Deutsch and Allan Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, 1984.

[16] ECMA, 2001. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-327.pdf.

[17] Manuel Egele, Engin Kirda, and Christopher Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *iNetSec 2009 Open Research Problems in Network Security*, 2009.

[18] Ú. Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *NSPW*, 1999.

[19] Úlfar Erlingsson. High-performance binary applets, 1997. URL http://www.cs.cornell.edu/home/ulfar/cuba/paper.

[20] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In *OSDI*, 2006.

[21] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen: a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 2002.

[22] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *CACM*, 1986.

[23] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC*, 2005.

[24] Advance Micro Devices Inc. *Software Optimization Guide for AMD64 Processors*, 2005.

[25] Advance Micro Devices Inc. *AMD64 Architecture Programmers Manual Volume 1: Application Programming*, 2009.

[26] Google Inc. Google web toolkit, . URL http://code.google.com/webtoolkit.

[27] Google Inc. The V8 JavaScript engine, . URL http://code.google.com/p/v8.

[28] Intel Inc. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1*, 2010.

[29] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC*, 2003.

[30] T. Lindholm and F Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[32] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Usenix Security*, 2006.

[33] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *CACM*, 1960.

[34] Kevin Millikin and Florian Schneider, 2010. URL http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html.

[35] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *POPL*, 1998.

[36] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proc. of POPL'10*, 2010.

[37] George C. Necula. Proof-carrying code. In *Proc. of POPL'97*, 1997.

[38] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[39] Thi Viet Nga Nguyen and François Irigoin. Efficient and effective array bound checking. *TOPLAS*, 2005.

[40] Department of Defense. Trusted computer system evaluation criteria (orange book), 1985.

[41] The Mono Project. The Mono language runtime. URL http://www.mono-project.com.

[42] Charles Reis and Steven D. Gribble. Isolating web programs in modern browser architectures. In *Proc. 4th ACM European conf. on Computer systems*, EuroSys '09, pages 219–232, New York, NY, USA, 2009. ACM. doi: http://doi.acm.org/10.1145/1519065.1519090.

[43] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *ASPLOS*, 1996.

[44] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[45] Joseph Siefers, Gang Tan, and Greg Morrisett. Robusta: taming the native beast of the jvm. In *CCS*, 2010.

[46] C. Small and M. I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency: Parallel, Distributed and Mobile Computing*, 1998.

[47] Fred Smith, Dan Grossman, Greg Morrisett, Luke Hornoff, and Trevor Jim. Compiling for template-based runtime code generation. *J. of Functional Programming*, 13(3), 2003.

[48] Michael F. Spear. Lightweight, robust adaptivity for software transactional memory. In *SPAA*, 2010.

[49] Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *CGO*, 2006.

[50] SunSpider Benchmark Suite. URL http://www2.webkit.org/perf/sunspider/sunspider.html.

[51] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *TOCS*, 2005.

[52] The perfmon2 hardware-based performance monitoring interface for Linux. URL http://perfmon2.sourceforge.net.

[53] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[54] Robert Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for unix. In *USENIX Security*, 2010.

[55] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. Secure dynamic code generation against spraying. In *CCS*, 2010.

[56] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.