BakerSFIeld: Bringing software fault isolation to x64

Joshua A. Kroll
Computer Science Department
Princeton University
Princeton, NJ, USA
kroll@cs.princeton.edu

Drew Dean
Computer Science Lab
SRI International
Menlo Park, CA, USA
ddean@csl.sri.com

***Abstract*—We present BakerSFIeld, a Software Fault Isolation (SFI) framework for the x64 instruction set architecture. BakerSFIeld is derived from PittSFIeld, and retains the assembly language rewriting style of sandboxing, but has been substantially re-architected for greater clarity, separating the concerns of rewriting the program for memory safety, and re-aligning the code for control-flow safety. BakerSFIeld also corrects several significant security flaws we found in PittSFIeld. We measure the performance of BakerSFIeld on a modern 64-bit desktop machine and determine that it has significantly lower overhead than its 32-bit cousin.**

I. Introduction

As widely distributed computing applications become more decoupled and are used in ways that can vary widely from day to day, trust relationships become increasingly difficult to establish. For systems with rapidly changing and mobile code, confinement of untrusted code modules is a tenable alternative to trust. Confinement is useful in secure systems design when a component cannot be trusted or easily verified to a high level of assurance. It also helps to place the security burden on the system's designers rather than on users. Software Fault Isolation (SFI) [1], also called sandboxing, implements memory and control-flow confinement for untrusted code modules loaded inside a trusted process.

The most common way to provide isolation of untrusted code is via an operating system process, which makes use of hardware memory protection to separate zones of trust into distinct address spaces. Efficiency in this world is gained by the use of special hardware (e.g. MMUs and TLBs). However, these methods are limited by the presence of hardware to support them and interaction across address spaces can be very expensive. Additionally, security policies are generally coarse-grained, with all processes confined to a pre-defined system call interface.

In SFI, we require all memory references to lie within specific, pre-defined bounds. We further require that all control flow transfer only allow instructions in a specific, pre-defined memory region to be executed. These properties can be checked statically for absolute references. For indirect references (e.g. references to memory or control transfer through a register), we require that the code include dynamic checks of safety. Code is verified at load time to ensure that all memory references and control transfers are either (1) absolutely safe or (2) include a valid dynamic check.

The original SFI techniques of Wahbe, et al. were only applicable to RISC architectures in which instruction words have a constant length and consistent alignment. Improved techniques from McCamant and Morrisett [2] allow sandboxing in CISC architectures by simulating constant-length instruction words through careful code re-alignment. McCamant and Morrisett also provide an implementation of their technique for the x86, which they call PittSFIeld. Most recently, Google has presented Native Client [3], a re-implementation of PittSFIeld which makes use of x86 segment registers for efficiency. Here, we present several improvements to the PittSFIeld sandboxing technique, fixing several vulnerabilities and improving end-to-end reliability. Alongside these, we present BakerSFIeld, an extension of the PittSFIeld tool which supports the x64 instruction set.[1]

To our knowledge, BakerSFIeld is the first implementation of SFI for the x64 instruction set. x64

[1]Throughout the paper, we will use x64 to denote the 64-bit extensions to the x86 instruction set variously referred to as 64-bit x86, AMD64, and x86-64.

brings several advantages over standard x86. First, in the x64 architecture there are 16 general-purpose registers, rather than the 8 in x86. This can allow for more aggressive compiler optimization. Second, 64-bit integer arithmetic is relevant in many use cases, such as cryptographic computations. Finally, modern x86 processors make good use of advanced hardware techniques such as superscalar execution and instruction-level parallelism. As 64-bit chips make their way out of the datacenter and into even low-end laptops, support for the x64 instruction set becomes increasingly important. Finally, because x86 memory segmentation is largely unavailable in x64, optimizations which rely on it are no longer valid. Thus, producing high-performance sandboxed code is a challenge.

This paper:

1) Describes vulnerabilities and improvements in the PittSFIeld SFI tool for the x86, improving security and reliability.
2) Provides a novel SFI implementation based on PittSFIeld which supports the x64 instruction set.
3) Provides a platform-independent code rewriting toolkit that acts on assembly source generated by an unmodified compiler and which, after transformation, can be built using a standard assembler.
4) Analyzes in detail the performance of this method on standard benchmarks. In particular, we consider the performance benefit of using 64-bit arithmetic.

The paper is laid out as follows. Section II describes the development of prior approaches to SFI and related low-level isolation technologies. Section II provides an overview of the SFI techniques used in BakerSFIeld. Section IV describes the BakerSFIeld architecture and code modification strategy. Section V describes the optimizations present in BakerSFIeld, while Section VI describes the results of running our tool in various circumstances on a standard set of SFI benchmarks taken from the PittSFIeld project. Finally, Section VII concludes.

II. Background and Related Work

A. SFI: RISC and CISC

The original concept of Software Fault Isolation came from Wahbe, Lucco et al. [1]. The key idea was simple: adjust all memory accesses (including control flow references) to go through a dedicated register. Then by choosing a memory region whose size and alignment are the same power of two, it is very easy to guarantee that any memory access will lie within that region. Simply by requiring that certain bits of that register always remain constant, it is possible to ensure that memory accesses are safe. For static references, this can be checked offline. For indirect references, it is necessary to insert some instructions to test and set these bits. The calculations to ensure this property require only bit operations, generally one `and` and one `or`.

In addition to developing the concept of SFI, Wahbe et al. provide a prototype implementation for the MIPS and Alpha architectures and measure a protection overhead of about 20% for both architectures. They also implemented a low latency system for cross-fault-domain RPC. They also include several important optimizations: they use unmapped guard pages on either side of a fault domain in order to allow for offsets from a register without needing to compute the effective address; they treat the stack pointer as a dedicated sandbox register because it is heavily used and never needs to leave the fault domain; finally, they also avoid always sandboxing the stack pointer by counting references to it.

The techniques of Wahbe et al. do not, however, work for all architectures. Implicit in their technique is a reliance on the fact that the instructions read and verified are also the instructions available on any execution path. This is naturally true for RISC architectures with constant-length instruction words. However, x86 has variable-length instruction words, and any byte in instruction memory might constitute the start of an instruction. In order to apply the Wahbe et al. techniques directly, it would be necessary to check all alternate parsings of the instruction stream that could be reached on some execution path. To solve this, McCamant and Morrisett proposed artificially imposing alignment constraints, choosing a power-of-two number of bytes and padding groups of instructions such that no instruction bridges two aligned blocks. In this way, jumps can be checked to ensure that they target the beginning of an aligned instruction group, and reliable disassembly can be assured. Their implementation, PittSFIeld, is the basis of our BakerSFIeld tool, and it will be described in greater detail below.

Wahbe et al. thought of their mechanism as applying

primarily to untrusted extension code. Today, as Internet applications become ever-more full-featured, it is only natural to ask how to isolate code which shares an address space with trusted components. Browsers, for example, are still in large part single-process applications. Yee, Sehr, et al. [3] provide a novel implementation of the PittSFIeld techniques which is geared towards the support of mobile native code modules that can be embedded into web sites. Native Client also supports a rich communication framework for inter-fault-domain communication and has a rather advanced trusted runtime to manage fault domains and their interaction with the underlying system. Like [1], Native Client uses a modified compiler tool chain to generate sandbox-compliant code, requiring developers to maintain separate infrastructure to produce sandboxed and non-sandboxed modules. Additionally, Native Client uses x86 memory segmentation to provide data integrity, a technique which is not applicable in x64.

B. Proof-Carrying Code

Three years after the publication of the first SFI paper, Proof-Carrying Code (PCC) was introduced by Necula and Lee.[4] PCC generalized SFI to verification of proofs of arbitrary properties of code, rather than just memory safety. PCC also implemented a more sophisticated proof checker. SFI makes the memory safety proof easy to check by explicitly inserting address masking operations immediately before using a memory address, much the same way that Scheme, Python, or other dynamically-typed, but type-safe, languages have runtime type checks. In this analogy, PCC is more like a statically-typed language (e.g., Standard ML, Java): the safety checks (unless the proof checker needs an explicit hint in the code) are done once, statically, making for a theoretically more efficient system — at the cost of proving a type soundness theorem, which may require non-trivial effort.

The key question is whether the additional expressiveness of PCC over SFI is actually useful in today's systems: is memory-safety enough? Insofar as few security policies are sufficiently formalized to be expressible in logic (and hence directly expressible in PCC), the answer may well be that memory safety is the necessary fundamental building block. The other issue with PCC is that the proofs tend to be quite large, so that the overhead of acquiring (in a mobile code setting) and checking the proofs can be significant. Both PCC and SFI may impose runtime overhead, if the proof checkers for the respective systems get in the way of certain compiler optimizations.

Typed assembly language, TAL [5] is relevant to SFI in much the same way. Here, invariants are expressed as types, and proofs of invariants correspond to inhabitants of those types. In this way, TAL might be seen as an intermediate position: the type system provides added high-level expressiveness, but might incur some additional effort on the part of the programmer.

C. Securing Untrusted Binaries

After the initial SFI work, much work has been done on a variety of methods to secure systems against untrusted binaries via confinement. We review some of this work here.

MiSFIT [6] combined the techniques of Wahbe et al. with static control flow analysis, collecting a list of legal jump targets and redirecting indirect jumps through code that checks explicitly that the target exists in a hash table. It also requires code producers and code consumers to share a secret, which is used to sign generated code.

Nooks [7] isolates trusted kernel code from untrusted device drivers using memory protection via a private page table. Device drivers are managed by a trusted runtime.

CFI [8] and XFI [9] use much finer grained tracking of control flow to restrict control flow to a nearly-arbitrary set of known targets. Additionally, XFI adds SFI for data integrity.

Vx32 [10] mixes instruction translation for control-flow integrity with x86 memory segmentation for data integrity. It is exported not as a stand-alone runtime, but as a library which can be linked with other applications. VxA [11] is a secure self-extracting archive tool built on Vx32.

Xax [12] is a model for a browser plugin using hardware memory isolation to create a kind of hardware virtual machine, isolated behind a narrow system call interface. Like Native Client, it is designed with mobile native code in mind, exporting a platform-independent Application Binary Interface and hooks for browser and system services.

Virtualization systems such as Xen [13] and managed code systems such as the Java Virtual Machine [14] or CLR [15] also have strong isolation and control

flow properties, but the techniques employed and applications relevant to these technologies are significantly different from SFI.

D. Optimizations: The Rise and Fall of Segmentation

x86 memory segmentation has long been a useful technique for improving the performance of single-address-space protection mechanisms. The idea is that since the processor's MMU is already doing segmentation calculations even if segments are not explicitly being used, by setting up data in different segments, it is possible to isolate accesses by loading special values into the segment registers and disallowing instructions which modify these registers.

The use of x86 memory segmentation [16] for efficiently multiplexing a virtual address space for security dates back at least to interprocess communication primitives in the L3 microkernel [17], although the techniques are very crude and the goal is to create a flat address space whenever possible. In [18], the technique of using segment registers to create multiple protection domains in a single address space is explicitly addressed. Its effects on building efficient data integrity domains are discussed in Liedtke's later paper, [19]. The use of segment registers for this purpose in L4 is discussed in depth in [20], along with their impact on performance.

Outside of the L^{4}Linux project, segment registers were used in [21] to implement efficient address space multiplexing for fault domain isolation. Their tool, Palladium, used this method for user-level data integrity and demonstrated the performance effectiveness by building a CGI handler into a web server that runs each CGI script in its own fault domain.

Finally, Native Client [3] makes use of segment registers to absolve itself of sandboxing checks, which allows for major performance improvements over the standard PittSFIeld techniques (approximately 5% vs. 20%).

E. The x64 Architecture

The x64 architecture [22], [23] might well be described as "x86 with more virtual memory addresses" as the instruction sets are very similar. However, there are a few noteworthy architectural differences:

1) Because memory addresses are 64 bits wide, x64 has architectural support for 64-bit integer arithmetic.
2) x64 has 16 general-purpose registers, each of which is 64 bits wide. x86 had only 8 general purpose registers, each 32 bits wide. The first 8 general-purpose registers in x64 are just extensions of the x86 registers. The others, called `%r8-%r15` are new. The added general-purpose registers make the idea of dedicated-register sandboxing more palatable than it was on the x86.
3) x86 memory segmentation is no longer used in x64. Segment registers are treated as though their value is always 0, even if values are assigned to them. The exceptions to this rule are the `%FS` and `%GS` segment registers, which play an important role in Microsoft Windows' kernel threads implementation.[2] However, the behavior of these registers is altered somewhat - values in these registers are treated as base values for address calculations, but no limit checking is performed. As such, the use of x86 memory segmentation as a shortcut to data integrity will not work in the x64 world.
4) x64 introduces a new addressing mode, allowing addresses to be specified with respect to the instruction pointer. This mode is referred to as `%rip`-relative addressing.

Modern processors implementing x64 also implement standard 32-bit x86 protected mode. The processors can freely switch between modes, and modes are set per-process.

III. Problems in PittSFIeld

In the course of analyzing and extending the PittSFIeld techniques and tools [24], we discovered a number of weaknesses in both the general CISC SFI methods and in the PittSFIeld tool itself. We describe each of these here. In section IV, we describe the architecture of BakerSFIeld and argue that it remediates each of these issues.

A. Instruction Encoding Length isn't Constant

In order to satisfy chunk alignment requirements, it is necessary for the rewriting tool to have a good understanding of the lengths of instructions as they will be encoded by the assembler. The PittSFIeld tool

[2]See `http://en.wikipedia.org/wiki/X86-64` and `http://msdn.microsoft.com/en-us/magazine/cc300794.aspx`

enumerates the lengths of all allowed instructions in a table, allowing these lengths to be looked up. In addition to a lack of generality, this solution can produce incorrect results.

In PittSFIeld, both the rewriting of assembly code to include sandboxing checks and the alignment of code are done in a single pass from top to bottom which considers only one instruction plus any state accumulated from prior instructions. This ensures that rewriting scales well with program size; this method runs asymptotically in linear time. Unfortunately, it isn't possible to make a complete determination of instruction size in this way. For example, instructions that include references to memory offset by the current instruction can have their offsets represented in different sizes (x86 supports 8-, 16-, and 32-bit offsets, although PittSFIeld bounds allowable offsets in all cases to less than 2^{16}). It is not possible to know the required size of such an offset if the offset would point to an instruction that the rewriter has not yet reached. Additionally, PittSFIeld does not check whether the increased code size due to the insertion of additional dynamic check instructions will change the size of instructions it has already processed or the size of a backwards offset (i.e. one pointing to an instruction it has already processed).

This is particularly important for branches, which are often addressed in just such a relative manner. Because proper code alignment assumes a particular size for relative branches, improperly aligned code can result when the insertion of extra code or alignment directives cause the target of a branch to move away from the branch itself, thus causing the encoding of the branch to grow.

B. Alignment isn't Top-to-Bottom

It is tempting to fix the above problem by separating the insertion of sandboxing checks from the alignment of the final, rewritten assembly code. However, alignment itself can also cause code size to increase because branch offsets may grow. One could repeat the alignment stage, noting that, in the worst case, one would only have to re-align the file as many times as there are ambiguously-sized instructions.

We thus come to the conclusion that the problem of aligning x86 instructions according to the CISC SFI rules does not admit a straightforward top-to-bottom solution. Either it is necessary to look ahead and behind when computing alignment, or it is necessary to take multiple passes. BakerSFIeld takes the second approach, adjusting alignment to be an idempotent function and iterating until a fixed point is reached. This solution is described in Section IV-C.

C. Writes Can Escape Dynamic Checks

During the verification stage, operations that might weaken the sandboxing invariants trigger conditions which propagate forward and are not allowed to reach a jump instruction. However, this solution is not fully context-sensitive. A value which is safe in one context may not be in another. Thus, in order to assure that sandboxing checks actually apply to the instructions that they are intended for, it is necessary either to strengthen the required sandboxing invariants or to strengthen the verification analysis.

For example, it is possible in the PittSFIeld model to jump via the reserved register `%ebx` directly to an instruction which performs an indirect memory write, also through `%ebx`. Since the value in `%ebx` is an address in the code section, it is out of bounds for the write. If there is no hardware memory protection in place, the write will succeed, allowing self-modifying code, which violates the SFI model's security. PittSFIeld solves this problem by attempting to keep track of the strength of various sub-invariants in the verifier. However, this makes the verifier brittle with respect to the tracking of these invariants, the semantics of the underlying instruction set, and the state of memory protection in the virtual memory subsystem prior to loading. It is notable that [1] avoid this issue by using separate dedicated registers for code and data sandboxing, which is unappealing on the x86 due to the paucity of general-purpose registers.

This method could also be used to subvert the `ret` instruction, allowing an arbitrary transfer of control to escape the sandbox. Such a case is shown in Figure 1. The more general problem is exhibited in Figure 2.

D. Verification

The PittSFIeld verifier is implemented essentially as a finite-state machine which carries information about the safety of particular instructions forward, failing either if an unsafe combination of states is reached or if a safe combination of states would unsafely be carried into a new context, such as via a jump. This design has several advantages: it's very simple to

```
     andl 0x10fffff0, (%esp)
.p2align
.Ltarget:
     ret
     ...
     pushl $outside_address
     jmp .Ltarget
```

Figure 1. An example of a dangerous instruction sequence allowed by the PittSFIeld model. In order to fix this, it is necessary to ensure that sandboxing checks and the instructions they protect inhabit the same chunk. The "`.Ltarget:`" label is a notational convenience.

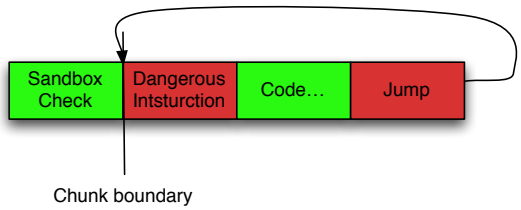

Figure 2. A code layout that PittSFIeld tries to prevent, but fails to implement correctly.

implement and analyze and it fails fast when presented with unknown states (such as unknown instructions or unknown argument configurations for a given instruction). However, despite the fact that [25] provide a formal proof of the validity of their method, we found a number of alarming implementation vulnerabilities in the verifier. For example, we discovered that the PittSFIeld verifier was decoding 3-argument operands not as `operand1`, `operand2`, `operand3`, but as `operand1`, `operand2`, `operand1`.

This is a classic example of a specification-implementation correspondence gap. We note that it would be useful to have a formal proof of the validity of an SFI verifier. However, if such a proof were to avoid issues such as these, it would require at least the formal semantics of all memory and control-flow related instructions in the x86 architecture, and the formal verification should correspond to the actual verifier implementation (e.g., by using a model checker for C programs or by rewriting the core verification functions in a language more suitable for reasoning, such as Coq).

IV. System Architecture

BakerSFIeld consists of two major components: a code generation tool chain that generates code compliant with our model and a special loader and load-time verifier for checking our safety invariants. The first of these takes in the assembly code generated by an unmodified compiler and rewrites it to comply with our invariants before assembling it with an unmodified assembler. Although its correctness is important for generating valid, usable sandboxed modules, it is not critical to the security of the system. The second of these constitutes a runtime environment for sandboxed modules. It constitutes the whole of the trusted computing base in our architecture.

The BakerSFIeld system supports confinement of memory writes and control transfers, both direct and indirect. It does not, however, sandbox memory reads, which would impose significantly higher overhead. Because reads are far more common and far less dangerous than either writes or control flow instructions, most other work on sandboxing has also made this trade-off [1], [2], [3]. The code transformation tool chain will accept valid AT&T/GNU Assembler syntax x86 or x64 assembly language and can in principle be run on any platform which supports Perl. A major advantage of the assembly language rewriting paradigm for sandboxing is that any source of x64 assembly language can be sandboxed, not just the output of GCC — unlike competing approaches that require the use of a specific tool chain. However, a significant portion of our code building infrastructure is suited specifically to Linux or depends on the GNU Binutils packages.

Figure 3 gives an overview of the BakerSFIeld architecture.

A. *Confinement Architecture*

Sandboxed modules are confined to two regions in memory: the code sandbox and the data sandbox. We follow [1], [24], and [3] in choosing regions whose size is a power of two and whose start address is aligned to that same power, which allows addresses to be checked for compliance using only bit operations, generally one `and` and one `or` instruction.

In the current implementation, we have performed all our work with 16 megabyte sandboxes stretching from `0x10000000-0x10ffffff` (Code) and `0x20000000-0x20ffffff`, though this choice was somewhat arbitrarily made, and the system supports easily choosing the sandbox size to support much larger memory regions.

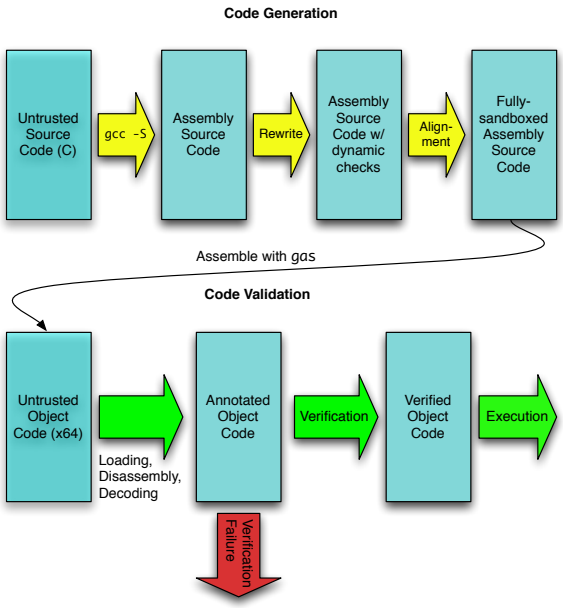

Figure 3. The BakerSFIeld system architecture

We use guard pages both above and below our code and data regions. This enables us to avoid some memory reference checking. Any register which is known to hold a value which is inside the sandbox can be referenced safely via an offset, so long as the offset is not larger than the size of the guard region. Guard regions are implemented using operating system memory protections, so any incorrect reference to memory outside the sandbox will generate a memory protection in the operating system, causing the process to be terminated.

The key insight in PittSFIeld is that it is possible to sandbox instructions in architectures with variable-length instructions by artificially imposing alignment constraints. To do this, PittSFIeld groups instructions into *chunks* with size (in bytes) and starting address alignment equal to a fixed power of two. BakerSFIeld also employs this method. Chunks which cannot be filled entirely with instructions are padded with no-op instructions so that no instruction crosses a chunk boundary. Then we require that all control flow target the beginning of some chunk, a property which can be checked with bit operations as with addresses above. This requires both checking control flow instructions for compliance and aligning instructions which are or might be the target of a control flow instruction.

`call` instructions are moved by the same no-op padding so that they occur only at the end of a chunk, ensuring that the return address from a call is aligned to the start of the next chunk. Unconditional jumps must be the last real instruction in their chunk, meaning they must either end their chunk or only be followed by no-ops inserted during alignment. Otherwise, the alignment rules could hide certain instructions in a chunk behind a jump, causing those instructions to be *dark* (i.e. never executable). We assume as an optimization that most jump targets will be identified by a label, and as such do not start new chunks for conditional branches.

For efficiency, padding is done via variable-length no-op instructions, so only a single no-op need be inserted in any given chunk. McCamant and Morrisett [2] found that no-op overhead from padding represented a significant part of the overall sandboxing overhead.

The layout of memory in our system is illustrated in Figure 4.

Notably, we must be careful that the set of instructions validated and the set of reachable instructions in execution are the same. These conditions imply that the provided binary can be statically disassembled (more specifically, they imply that execution is constrained to the static disassembly of the binary which is produced during analysis). This is important: a common code obfuscation technique is to cause a jump into what

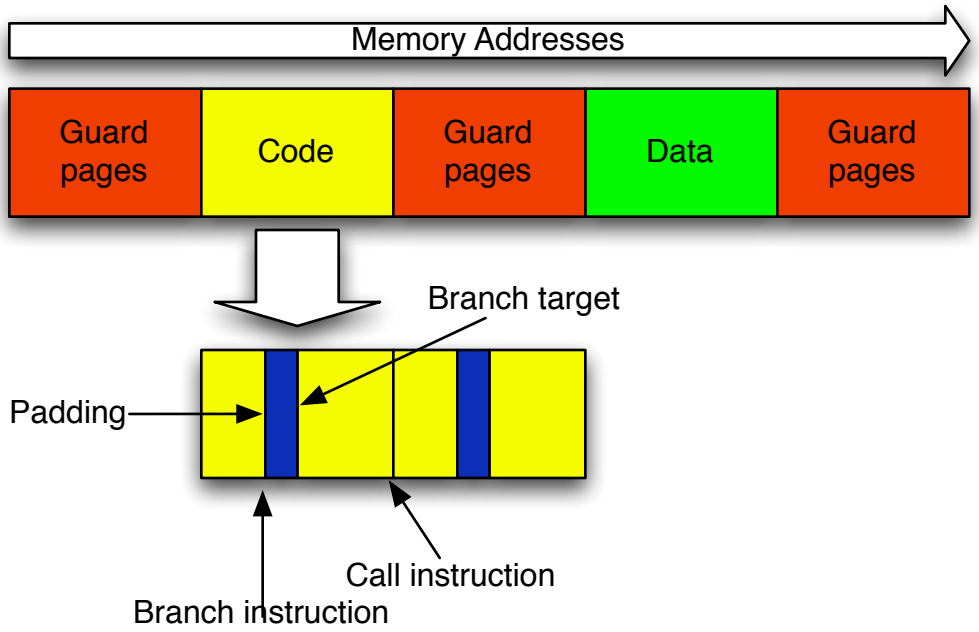

Figure 4. The organization of memory in BakerSFIeld. Execution is constrained to code in the ”Code” section while memory writes are constrained to the ”Data” section. Guard pages are used to enable optimizations for fuzzy checking of memory offsets.

appears to be the middle of another instruction in order to cause unexpected behavior. However, because disassembly of each instruction chunk always proceeds from the start of the chunk to the end, because no instruction bridges a chunk boundary, and because all control flow can only target the beginning of any chunk, we know that the set of validated chunks is also the set of reachable chunks in execution.

Static checking is straightforward, except for the caveat given above. More difficult is dynamic checking of indirect memory references. BakerSFIeld uses PittSFIeld-style dynamic checks, reserving a register (`%rbx`) for sandboxing. The effective address of any potentially dangerous memory reference, whether a write or a jump, is loaded in to the dedicated sandboxing register using a `lea` instruction. Then the address is cleaned, using an `and` instruction to clear high-order address bits that do not match the address of the relevant section and an `or` instruction to set the high order bits of the section address.

Here, we use the principle of “ensure, don’t check” [2], which says that instead of terminating a program with an error report if non-compliant behavior is detected, we can simply force an unsafe action to become an action which complies with the security policy. It is much more efficient simply to force memory addresses to be compliant than it is to determine whether they are compliant and generate an explicit error if they are not. In this way, correct programs are unaffected by the checks, but incorrect, non-compliant, or generally malicious programs may behave in unusual or unexpected ways. Because the programs are confined, however, this behavior cannot have side effects on the system, and can only negatively affect the bad code which generated it. This optimization is also included in [1] and [3].

As an optimization, we also follow PittSFIeld in treating the registers `%rsp` and `%rbp` as usually sandboxed. These registers may be copied from one to another without triggering additional dynamic checks. Small offsets from these registers are also allowed without checks. `%rbp` is re-checked whenever it is rewritten (i.e. a value is loaded from memory or another register) and before jumps if it is considered dirty. `%rsp` is treated similarly. However, because `%rsp` is frequently modified, small modifications to this register (less than a threshold depending on sandbox size and guard page size) are counted, and a fixed number of small changes are allowed before new dynamic checks are triggered. `%rsp` is also checked when it is re-

loaded or before a jump if it has been modified since the last check and is dirty.

We alter the PittSFIeld structure slightly by requiring that dynamic check instructions be placed in the same chunk as the potentially dangerous instruction they are meant to sanitize. At the cost of some minimal code size overhead, this technique allows us to relax conditions in the PittSFIeld verifier which require that bad state (e.g. the modification of a value in `rbp`) be remediated prior to a jump. In this case, unsafe instructions cannot be targeted directly by jumps, since any instruction which must be preceded by a check will be after the check in any given chunk.

In order to provide the functionality of the C library within the sandbox, we provide the modified sandbox-friendly library from the PittSFIeld project, with minor updates to retain compatibility with the newer Linux build tool chain. Currently, stub functions containing trampolines to outside helper function are installed for each function in the library at the base of the sandbox's code region, just above a small entry function. These functions can be called to provide access to the real C library functions outside the sandbox. The set of available functions can be tailored to the needs of the confined application and the security policy required. In accordance with the principle of least privilege, it is also advisable to only install those stub functions in the intersection of the set of functions allowed by the security policy and the set of functions actually called by the confined code. This determination could easily be made at compile time, and the BakerSFIeld tool has full support for installing only those stub functions which are relevant to a particular guest module.

B. Threat Model and SFI Invariants

We adopt the standard SFI threat model used by [1] but more carefully articulated in [2]. In general, compilation and code rewriting are done by an untrusted code producer. The safety policy is enforced by a separate verifier which is part of the runtime environment. In this way, the verifier can ensure:

- The code that is loaded and verified is the code that is executed.
- Only code which is properly verified is allowed to execute.

We expect the verifier to be able to take in arbitrary binary object files and only to allow execution if the input binary complies with the safety policy. Further, so long as they pass verification, sandboxed modules may exhibit arbitrary behavior, including executing any reachable instruction block in the validated code segment, using available system calls, and providing bad or incorrectly formatted instructions.

Our safety policies are simple and verification can be inferred directly from assembled code without the need for debugging or symbol-table information. We take inspiration from the safety policies in [2] and [3], but we modify some constraints as described above. The verifier checks the following invariants:

1) All direct memory references lie within the bounds of the data section.
2) All direct control transfers target properly aligned addresses within the code section.
3) All indirect writes and indirect control transfers are grouped with a valid dynamic safety check which tests:
   - For indirect writes, whether the write lies within the data section.
   - For indirect control transfer, whether the transfer targets an address in the code section and whether that address is aligned to the start of an instruction chunk.
4) No instruction or instruction group crosses a chunk boundary.
5) All control transfer instructions are properly aligned:
   - All `call` instructions occur at the end of their respective chunks.
   - All unconditional jumps either end their chunks or are only followed by no-op instructions, to prevent the possibility of alignment rules creating "dark" instructions.
   - All jump targets are aligned to the beginning of a chunk.

In general, these safety properties are not tractable to compute. Instead, BakerSFIeld and other SFI systems re-write code to make verification simple by making these properties manifest. We trade tractable verification for one-sided error: if a program is accepted, then it preserves these invariants and is safe to run. However, we may reject some programs unnecessarily if their safety is not sufficiently easy to determine by our simple finite-state verification technique, described below. Still, given that completeness in deciding such a property is impossible, trading some completeness for

efficiency while preserving soundness is a reasonable compromise.

Note too that these invariants explicitly disallow self-modifying code, and in fact also guarantee the property that the instructions disassembled and verified are the same instructions available on any execution path.

C. Compliant Code Generation

BakerSFIeld is capable of generating object code which complies with the above safety invariants by interposing a code transformation tool in the standard GNU code generation tool chain. Figure 3 shows the BakerSFIeld compiler workflow.

Code is first compiled to assembly source by a standard, unmodified compiler. We used GNU GCC, but in principle this is not necessary. At present, we support only AT&T/GNU Assembler syntax in our code transformation tool. The rewriting tool is implemented in Perl, and is thus highly portable. It consists of three phases: a simple pass which unrolls any string instructions present in the code, a rewriting tool which inserts the required sandboxing checks, and an alignment tool which generates code satisfying the BakerSFIeld alignment rules.

Code is then fed in to a rewriting tool, implemented as approximately 300 lines of Perl. This rewriting tool inserts all of the dynamic checks necessary for sandboxing as well as hints to the alignment tool about how to group instructions when instructions and their checks need to inhabit the same chunks. All of the optimizations which involve changing runtime checks are also implemented in this phase.

Next, the code is fed into an alignment tool, which generates code that satisfies the BakerSFIeld alignment rules. This code constitutes BakerSFIeld-compliant assembly source, ready to be assembled by a standard assembler. The alignment tool is implemented as approximately 500 lines of Perl.

Alignment of x86 and x64 code is a finicky problem. In order to properly determine when to start a new chunk or how to place a `call` exactly at the end of a chunk, it is necessary to know with precision the length of every instruction's encoding. x86 and x64 encodings are extremely complicated, with single instructions ranging from 1 to 15 bytes in length. The same instruction may be valid with and without certain optional prefixes, and the processor may ignore redundant optional prefixes. It may also be impossible to know in advance how large an instruction is. For example, branches which target labeled offsets from a register (rather than absolute offsets) may be encoded in 2 bytes: a 1-byte opcode plus a 1-byte displacement if the displacement is absolutely less than 128 bytes (displacements in x86/x64 are signed). But it may also be encoded as a 5-byte object: a 1 byte opcode plus a 4-byte displacement (there are no 64-bit displacements in x64). To complicate the situation even further, alignment may cause the target of a branch to move beyond the magical 128-byte limit, so it is not possible simply to compute instruction lengths prior to alignment.

BakerSFIeld uses a simple, highly general solution to this problem which leverages standard tools. We have implemented alignment as an idempotent function.. We can iteratively re-align the code if changes might have changed our view of the information necessary to compute the alignment. However, if applying our alignment twice produces the same output as applying it once, we know we have successfully computed a fixed point of the alignment function. Thus, we perform the following loop. Say the input is $Code_{A0}$ and the alignment approximation function is $A(Code_{AN}, Listing_N)$. Then beginning with $N = 0$:

1) Compute the lengths of all instructions and the result of any assembler directives which could affect alignment. This is $Listing_N$.
2) Compute the code alignment based on this information, $Code_{A(N+1)} = A(Code_{AN}, Listing_N)$.
3) Check if the computed alignment matches the input, $Code_{A(N+1)} = Code_{AN}$. If it does, the function A had the effect of the identity, and the code must be properly aligned. If it does not, start over at step 1 with the input $Code_{A(N+1)}$.

This leaves us with the problem of computing each $Listing_N$, which describes the correspondence between the input file and the length of the encoding of each instruction and alignment directive. Accurately computing this information in full generality is approximately as complex as writing an assembler. In order to solve this problem efficiently, we leveraged existing tools. GAS will generate a listing of its output, which describes exactly the correspondence on a line-for-line basis between input and output. As an added benefit, the listing will provide extra information, such as the address of each instruction (from which we compute its length) and the encoding itself, which is useful for

debugging.

Any given input file may have many valid alignments. In practice, our fixed point technique computes a reasonable alignment. To guarantee termination, we never remove no-ops, so instructions can only ever grow in size. We are also aggressive about aligning to new chunks. However, these issues do not seem to affect code size adversely and we usually converge to the idempotent alignment function within about 4 rounds of approximation. Because each pass of the alignment tool is implemented as a top-to-bottom pass, it is possible that sub-optimal choices are taken early in the approximation. There is currently no method to correct these once they happen. One improvement is to provide the compiler with some alignment directives so that the distance between the raw, rewritten code and the compliant aligned code is minimized.

Once the code has been properly aligned, it can be assembled with the standard GNU Assembler for x64.

D. Code Validation

Validation of code in the BakerSFIeld system is done at load-time. The basic workflow of loading, validation, and execution is shown in Figure 3. The trusted loader-verifier constitutes the entire runtime environment for a sandboxed module and thus the entire trusted computing base for the system. It is implemented in approximately 2800 lines of C, of which approximately 2000 are devoted to verification. Of those, most are simply involved in instruction decoding; the core verification section is implemented in approximately 600 lines of C.

The verifier is implemented essentially as a finite state machine, where states are labeled by a combination of conditions about the SFI invariants described above. This state machine passes over the object code to be verified from top to bottom, computing for each instruction a conservative property which describes an estimate of the state that the processor will be in when it reaches that instruction. State transitions are per instruction and also depend on alignment. Additionally, conditions which would strengthen any of the SFI invariants described above (such as a sandboxing check instruction or group of instructions) persist to the end of a chunk boundary. Any conditions which would weaken such an invariant persist across chunk boundaries until they are remedied. If an unsafe condition reaches a control flow instruction, such as a `jmp`, `Jcc`, `call` or `ret`, verification will fail.

Verification relies upon disassembly. As mentioned above, the SFI invariants guarantee that the instructions which are disassembled and verified are the instructions which will be executed on any execution path. However, it is still necessary to reliably disassemble the instructions, at least enough to know whether they affect the status conditions used by the verifier. BakerSFIeld relies on an external disassembler, Udis86 [26]. PittSFIeld by contrast made use of a library called libdisasm [27], which did not support x64.

V. Optimizations

We describe here several optimizations implemented in our tool. Beyond those described in Section IV, PittSFIeld implements two sandboxing-check optimizations which are retained in BakerSFIeld: single instruction address operations and efficient returns.

Ordinarily, in sandboxing an address, it would be necessary to use an `and` instruction to clear some bits and an `or` instruction to set some others. However, we follow PittSFIeld's lead and cleverly arrange our sandbox regions so that we can reduce this calculation to a single instruction. We choose the code and data regions such that their base address has only a single bit set (as mentioned, the default code and data regions in BakerSFIeld are `0x10000000` and `0x20000000` respectively). Further, we reserve as a guard region the region starting at address `0x0` which is the same size as the code and data regions.[3] Then it is possible to use a single `and` instruction for sandboxing, since this instruction will clear all of the necessary bits, except possibly the one which needs to be set. Then either the sandboxed address was correct, that bit is set, and all continues normally, or the address was incorrect (if, for example, it was from incorrect or malicious code), and the memory reference will be trapped to the guard region, which will generate an error. McCamant and Morrisett [2] found that this optimization decreased their overhead by approximately 10%.

In order to take advantage of modern branch prediction hardware, it is useful to implement procedure returns not as indirect jumps through registers (as indirect `call` instructions are implemented in BakerSFIeld)

[3]There is no reason the code and data regions need to be the same size. If they are not, we simply reserve from zero the region which is the size of the larger sandbox.

but instead using the `ret` instruction. To accomplish this, BakerSFIeld modifies the return address on the top of the stack just before the `ret` instruction, as if that address were stored in a register. McCamant and Morrisett [2] found that this optimization reduced worse case overhead by over 50% and standard overhead by about 25%.

It is worth noting that a highly-efficient optimization is available for sandboxing in x64 entirely for free. Using the new `%rip`-relative addressing mode, it is possible to get integrity for many memory references, both for control flow and data writes. Because the verifier can resolve `%rip`-relative addresses fully statically (once the file is loaded, the verifier knows exactly the value of `%rip` for every instruction), it is not necessary to provide dynamic checks for such instructions. Using the `-fpic` option to GCC, it is possible to make a large number of the memory references in a program obey this paradigm. Optimized code from GCC at levels above `-O1` also make use of `%rip`-relative addressing.

VI. Performance Results

We ran our tool on the microbenchmark suite designed for PittSFIeld. It does very well, with performance overheads between 2-75%. Overall, our average overhead was 20.4%, which compares favorably with the 21% average overhead in PittSFIeld, particularly when one considers that our average is significantly worsened by poor performance in the `gzip` benchmark and that the PittSFIeld benchmarks were taken at GCC optimization level `-O3` while the BakerSFIeld benchmarks were built at `-O2` due to a compatibility issue unresolved at the time of this writing.

All benchmarks were run on a Dell Optiplex GX620, with a 3.2 GHz Pentium 4 CPU, 1 GB of RAM, 80 GB disk, running Ubuntu Linux 8.04, with the Linux 2.6.24 kernel, and GCC 4.2.4 The benchmarks were compiled with `-g -ffast-math -fno-stack-protector -O2 -fno-schedule-insn2 --fixed-rbx -fno-omit-frame-pointer`, and the C library (from PittSFIeld) compiled with `-O3 -fno-stack-protector -fno-schedule-insns2 --fixed-rbx -fno-omit-frame-pointer`. The benchmark programs are as follows:

- λ-42 simply returns the constant value 42. It's primary use is to measure the size of the C library, and its sandboxing overhead. The runtime performance of it is dominated by the load-time verification of the sandboxed C library.
- fib naïvely (i.e., via recursion) computes the 34^{th} number in the Fibonacci sequence. This is a good "worst-case" for indirect control flow, as the recursive function requires a significant number of `ret` instructions.
- factor computes a nontrivial factorization of a 58-bit composite number using the wheel factorization algorithm. This benchmark is heavy on integer arithmetic.
- bzip2 is the an implementation of the Seward bzip2 compression algorithm, meant to stress memory performance.
- gzip is the standard GNU compression utility based on the DEFLATE algorithm.
- md5 computes the md5 hash of a pre-determined 16-byte buffer in memory.

A. Code Size

Code size benchmarks are shown in Table I.

Code size blowup for these microbenchmarks is strongly dominated by the inclusion of the entire replacement C library in the resultant object file. Thus, the λ-42 benchmark gives us a good idea of this overhead, about 10,500 bytes. For larger programs such as gzip, this overhead is dominated by the size overhead of the program itself. However, because the replacement C library consists mostly of calls to helper functions which wrap the real libc implementations outside the sandbox, the overhead is very high. For example, controlling for libc overhead in the gzip benchmark yields a code size blowup ratio of approximately 1.43.

Still, these ratios compare very favorably with those given in [2]. Since these files have not been modified from the PittSFIeld, it is most likely that the favorable size overhead gains are due to the more aggressive use of memory offset references via `%rip`-relative addressing, which lead to shorter instructions. Our worst-case blowup, gzip, has identical performance to the PittSFIeld version, however.

Program	lambda-42	fib	factor	bzip2	gzip	md5
Unsandboxed	18923	19039	19235	83452	64578	21804
Sandboxed	29664	29840	30112	137840	106740	33608
Ratio	1.57	1.57	1.57	1.65	1.65	1.54

Table I
CODE SIZE (IN BYTES; TEXT SEGMENT ONLY) FOR SANDBOXED VS. UNSANDBOXED BENCHMARKS

Program	lambda-42	fib	factor	bzip2	gzip	md5
Unsandboxed	0	10.04	1.71	7.64	1.86	5.15
Sandboxed	0.015	10.39	1.75	8.19	3.24	5.91
Ratio	∞	1.04	1.02	1.07	1.74	1.15

Table II
AVERAGE EXECUTION TIME IN SECONDS FOR SANDBOXED VS. UNSANDBOXED BENCHMARKS

B. Execution Overhead

Execution overhead benchmarks are shown in Table II.

Execution overhead was calculated by taking 10 runs of the code and measuring using the Linux `time` utility, then taking descriptive statistics. The standard deviation and variance were in all cases very small, only a few milliseconds, and are thus insignificant.

These numbers compare favorably on a pairwise basis with the PittSFIeld numbers with the exception of gzip, which we measured at 74% overhead and [2] measured at 17% overhead. Leaving out this benchmark yields an average overhead of only 7%, compared to an average of approximately 24% for the same benchmarks in [2].

VII. CONCLUSION

BakerSFIeld brings the advantages of Software Fault Isolation to the x64 instruction set architecture. It also offers a much cleaner architecture than PittSFIeld by separating code rewriting (for memory safety) and code re-alignment (for control flow safety). For most applications, the overhead is very reasonable. Indeed, the overhead of SFI on modern superscalar processors seems to be much lower than on older x86 processors.

Future work will proceed along three major axes:

- Engineering improvements to the system, to make it easier to use directly and to make it support the integration of SFI domains into actual programs. Additionally, it will be useful to explore the various benchmarks that have been and can be taken, to fully understand the reasons for the lower 64-bit overhead.
- Verification of the verifier. We wish to provide a provably-sound verifier which can be used for high-assurance confinement. Note that this provides a way to leverage verified codebases by providing safe (well, at least confined) unverified extensions. This allows for a system with composable trust properties, even if some components are not trustworthy.
- Novel applications of SFI. For example, SFI could be used to provide a user-thread system that includes memory safety guarantees. This could be useful in engineering large pieces of software, in which cheap cross-component communication is important but isolation is also important. An example might be browser tabs, which could be managed by a trusted core thread but could be isolated from each other in case one crashes.

ACKNOWLEDGMENT

This research was supported by the National Science Foundation under Grant Number CNS-0524111.

REFERENCES

[1] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*. ACM, 1994, p. 216.

[2] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *15th USENIX Security Symposium*, 2006, pp. 209–224.

[3] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[4] G. Necula and P. Lee, "Safe kernel extensions without run-time checking," *ACM SIGOPS Operating Systems Review*, vol. 30, no. si, pp. 229–243, 1996.

[5] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, pp. 527–568, May 1999. [Online]. Available: http://www.acm.org/pubs/articles/journals/toplas/1999-21-3/p527-morrisett/p527-morrisett.pdf;http://www.acm.org/pubs/citations/journals/toplas/1999-21-3/p527-morrisett/

[6] C. Small and M. Seltzer, "MiSFIT: Constructing safe extensible systems," *IEEE concurrency*, vol. 6, no. 3, pp. 34–41, 1998.

[7] M. Swift, S. Martin, H. Levy, and S. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, p. 107.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications." in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM New York, NY, USA, 2005, pp. 340–353.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, p. 88.

[10] B. Ford and R. Cox, "Vx32: Lightweight, User-level Sandboxing on the x86," in *2008 USENIX Annual Technical Converence*, June 2008.

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

[12] J. Douceur, J. Elson, J. Howell, and J. 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.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003, p. 177.

[14] T. Lindholm and F. Yellin, *Java(tm) Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[15] A. Kennedy and D. Syme, "Design and implementation of generics for the .NET Common Language Runtime," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM New York, NY, USA, 2001, pp. 1–12.

[16] Crawford, J. and Gelsinger, P., "Programming the 80386," SYBEX Inc., 1987.

[17] J. Liedtke, "Improving IPC by kernel design," in *ACM Symposium on Operating Systems Principles: Proceedings of the fourteenth ACM symposium on Operating systems principles*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1993.

[18] ——, "Improved address-space switching on Pentium processors by transparently multiplexing user address spaces." GMDGerman National Research Center for Information Technology, Sankt Augustin, Germany, Tech. Rep. 933, Sept. 1995.

[19] ——, "Onμ-kernel construction," in *15th ACM Symposium on Operating System Principles (SOSP)*. Citeseer, 1995.

[20] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of μ-kernel-based systems," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*. ACM New York, NY, USA, 1997, pp. 66–77.

[21] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *ACM Symposium on Operating Systems Principles: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 1999.

[22] *AMD64 Architecture Programmer's Manual*, Publication No. 24592-24594; Revision 3.14 ed., Advanced Micro Devices, September 2007.

[23] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Order Number 253665-253669 ed., Intel Corporation, September 2009.

[24] McCamant, S. and Morrisett, G., "Efficient, Verifiable Binary Sandboxing for a CISC Architecture," MIT Computer Science and Artificial Intelligence Laboratory, Tech. Rep. MIT-CSAIL-TR-30, 2006.

[25] McCamant, S., "A Machine-Checked Saftety Proof for a CISC-Compatible SFI Technique," MIT Computer Science and Artificial Intelligence Laboratory, Tech. Rep. MIT-CSAIL-TR-2006-035, 2006.

[26] Thampi, V. (2009) The Udis86 Disassembler Library for x86 and x86-64. [Online]. Available: {http://udis86.sourceforge.net/}

[27] (2002) Libdisasm: x86 Disassembler Library. HCU Linux Forum. [Online]. Available: {http://bastard.sourceforge.net/libdisasm.html}